NIELS BOHR INSTITUTE
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN
DENMARK

A High Performance Backend for Array-Oriented Programming on Next-Generation Processing Units

Simon Andreas Frimann Lund

Academic advisor: Brian Vinter

September 2015

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Submitted: September 2015
Defended:
Final version:

Advisor: Brian Vinter, University of Copenhagen, Denmark

Cover art: Photo from the city archives photographic atelier (starbas.net Ref. 2134/TMF07209) of the Copenhagen harbor and the Langebro bridge Anno 1954. The bridge then carried commuters and freight trains to and from the factories along the docks of the harbor. The bridge opened several times an hour for ships supplying the factories with coal. The freight train bridge was later converted to carry pedestrians and bicyclist. Today the bridge opens a couple of times a week.

Abstract

The financial crisis, which started in 2008, spawned the HIPERFIT research center as a preventive measure against future financial crises. The goal of prevention is to be met by improving mathematical models for finance, the verifiable description of them in domain-specific languages and the efficient execution of them on high performance systems.

This work investigates the requirements for, and the implementation of, a high performance backend supporting these goals. This involves an outline of the hardware available today, in the near future and how to program it for high performance. The main challenge is to bridge the gaps between performance, productivity and portability.

A declarative high-level array-oriented programming model is explored to achieve this goal and a backend implemented to support it. Different strategies to the backend design and application of optimizations are analyzed and experimentally tested. Resulting in the design and implementation of Bohrium a runtime-system for transforming, scheduling and executing array-oriented programs.

Multiple interfaces for existing languages such as Python, C++, C#, and F# have been built which utilize the backend. A suite of benchmarks applications, implemented in these languages, demonstrate the high-level declarative form of the programming model. Performance studies show that the high-level declarative programming model can be used to not only match but also exceed the performance of hand-coded implementations in low-level languages.

Acknowledgements

With this thesis, I end my studies in computer science. Although, come to think it, I probably won't. The curiosity that was awakened when I as a kid peered over my eldest brother Benjamin's shoulder and saw the wonder of an i486DX is not going away anytime soon. However, what will end are my studies as a Ph.D. student thereby concluding a decade spent as a student at the University of Copenhagen. So for me this is the end of an era, a time that I am thankful for and have many to thank for.

First and foremost then I would like to thank my immediate family. Mor, Far, Benjamin, Zanne, Tobias, Heidi, my awesome nephews Sebastian, Valdemar, and Alexander: Thank you for your encouragement, your unconditional love, your support when challenges were plenty, for your patience and understanding when I was in wired into another world, and for being there when I got back out. Also, thanks to my uncles, aunts, and cousins for following my process a bit. And even though you probably cannot read this Søren: Thanks for being my uncle and for the positive impact you have had, and continue to have on my life.

Thanks also, to my friends A.C., Johannes, Laura, Marko, Matias and Phie for reminding me to go outside once in a while. Thanks to Astrid for her encouragement in embarking on this voyage. Thanks to Frederik in all his capacities as a colleague, friend, and of course proofrederr. A special thanks to Lene for turning her head just when she did and let her eyes meet mine.

The decade of studies has gone by fast; I have had the opportunity to swim in a sea of knowledge taking to shore different topics of interest. I have with my Ph.D. had the pleasure of diving deep into a topic, meet very talented researchers, work with bright minds and explore interesting subjects. For all this, I can thank my advisor Brian Vinter.

Thanks to all the members of the distlab, image, and eScience groups for the entertainment and great work environment. Thanks to those I have worked closely with over the years: James Avery, Kenneth Skovhede, Klaus Birkelund Jensen, Mads R. B. Kristensen, Troels Blum, and Weifeng Lui. Although he puts curly brackets in the wrong places, I especially want to thank Mads for his insight, approach and letting me pick his mind.

Thanks to Bradford L. Chamberlain for hosting my visit to Cray Inc. and for his inspiring research on ZPL and Chapel. Thanks to the Chapel Team: Ben Harshbarger, David Iten, Elliot Ronaghan, Greg Titus, Kyle Brady, Lydia Duncan, Mike Noakes, Sung-Eun Choi, Thomas Van Doren, Tim Tzakian, Tom Hildebrandt, Tom MacDonald, Vassily Litvinov and thanks to the fascinating people of Seattle that I crossed paths with. Especially Erika, Chris, David, and Monique.

Thanks to the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (hiperfit.dk) under contract number 10-092299 which has partially funded me. And thanks to the University of Copenhagen and the Niels Bohr Institute. And just because I can write anything in acknowledgements, I would like to thank music for keeping a link to the surface as I dove deep down in my bubble. Especially the sounds of: A-Trak, Air, Alt-J, Amon Tobin, Apparat, Arcade Fire, Basement Jaxx, Beach House, Beirut, Birdy Nam Nam, Bjørn Svin, the Black Keys, Bloc Party, Blur, Boards of Canada, Bon Iver, Bonobo, Boys Noize, Caribou, the Chemical Brothers, Com Truise, the Cranberries, Crystal Castles, Daft Punk, Digitalism, Diplo, Dirty Projectors, Disclosure, DJ Shadow, Fever Ray, FKA Twigs, Florence + The Machine, Foo Fighters, Four Tet, Friendly Fires, the Fugees, the Glitch Mob, Gold Panda, Groundislava, Hawksley Workman, Hercules and Love Affair, Hot Chip, Interpol, Jamie Woon, Jamie XX, Jimi Hendrix, Joy Division, Justice, Kashmir, Kasper Bjørke, Keane, Kent, Kings of Convenience, Klaxons, the Knife, La Roux, LCD Soundsystem, Leonard Cohen, Little Dragon, the Low Frequency in Stereo, M.I.A., Malk De Koijn, Manic Street Preachers, Massive Attack, Metric, MGMT, Moderat, Modeselektor, Modest Mouse, Mogwai, Morcheeba, MSTRKRFT, Muse, Mø, the National, New Order, Nick Cave and The Bad Seeds, Nicolas Jaar, Nils Frahm, Nirvana, Nosaj Thing, Oasis, Orbital, the Pixies, PJ Harvey, Placebo, Portishead, Pretty Lights, Pulp, Queen, Queens of the Stone Age, the Raconteurs, Radiohead, Ratatat, RJD2, Royksopp, Santigold, SBTRKT, Schoolboy Q, Sia, Simian Mobile Disco, Slow Magic, the Smashing Pumpkins, Snow Patrol, Sort Sol, Soulwax, the Strokes, Suspekt, Tiga, Todd Terje, Tool, Travis, Trolle Siebenhaar, Turboweekend, UNKLE, the Verve, When Saints Go Machine, White Lies, the White Stripes, the Whitest Boy Alive, Yeah Yeah Yeahs, Yeasayer, Yuna and Kate Bush for making me keep running up the hill.

Contents

Part I

Extended Abstract

Chapter 1

Context

The financial crisis, which started in 2008, created a recession in the world economy which resulted in unemployment and instability. One of the triggering factors in the onset of this crisis was the poor performance of the banks in terms of pricing their financial products and assessing the risk of their financial transactions. National and international authorities have consequently decided to tighten regulations in the financial sector for example requiring that banks document the risk of every loan. Regulation and legislation are means of trying to prevent a new crisis, but it does not solve the problem that banks are performing poorly. Banks must directly address the problem at hand by strengthening their capabilities.

A Danish initiative, which focuses on this approach, is the academic research center HIPERFIT. It is the goal of HIPERFIT to come up with new mathematical models within financial mathematics and develop powerful and safe tools to evaluate them; thereby strengthening the banks' capabilities. The research areas of HIPERFIT are cross-discipline as figure 1.1 illustrates.

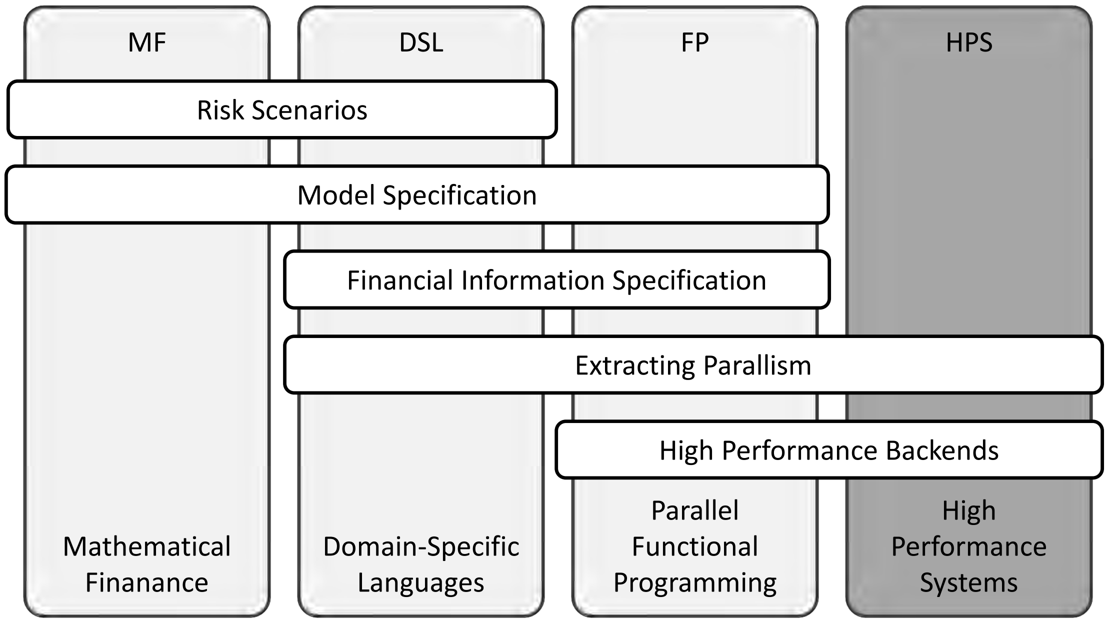

MF
DSL
FP
HPS
Risk Scenarios
Model Specification
Financial Information Specification
Extracting Parallism
High Performance Backends
Mathematical Finanance
Domain-Specific Languages
Parallel Functional Programming
High Performance Systems

Figure 1.1: Research areas of the HIPERFIT research center.

The area of Mathematical Finance explores financial models and methods with a focus on increasing accuracy of models of financial phenomena and determining risks with greater precision. Increasing accuracy and precision imply increasing parameters for a model thereby increasing compute-time and complexity of the implementation. Domain-specific languages aid this description by providing high-level language constructs that are close to the domain of the mathematical models. Basing these languages on functional language design provides a means for formally verifying the implementation thereby lowering the possibility of introducing errors at the implementation level. The last pillar, high performance systems, researches ways to support the high-level constructs and obtain efficient execution of mathematical models.

This manuscript documents the work that I have done and collaborated upon during my Ph.D. studies within the area of high performance systems. The following section provides an overview of the project itself along with an outline of the remainder of the thesis.

1.1 Structure of the Thesis

In the thesis and the publications, terms such as high-level, low-level, portability, productivity, and performance are used. These terms can be perceived differently based on the experiences and perspective of the reader. So a default context for them will be established here. When referring to high-level languages, then the reader should think of languages such as domain-specific, scripting, functional and, in general, *declarative* languages. Or in others words, languages that focus on expressing *what* the machine should compute and not *how* it computes it.

High-level has historically been a fitting label for languages such as C, C++, and Fortran due to the abstractions they provide over machine code and assembly languages. These languages, however, provide capabilities of explicit control of hardware details and often require the programmer to use them and are for this reason referred to as low-level.

Portability is a broad term that can refer to concepts such as platform portability where a given program can run on different platforms such as Unix, Linux, MacOSX and Windows. Another usage is machine portability, that is, a given program can run on different hardware architectures. Platform portability is of interest; however, the default context and primary focus is on machine portability.

Productivity and performance, commonly perceived as two opposite and conflicting concepts, introduce a tradeoff – the more you have of one, the less you have of the other. High-level languages are associated with productivity as they abstract away details of the underlying machine and how to program it. Arguably increasing productivity of the programmer since only the domain of interest needs to be expressed and not the concerns of mapping that domain to a specific machine architecture. Mapping the domain to a given machine thus becomes the responsibility of the language interpreter, compiler, runtime or, generally, the backend of the language.

Performance is an abstract term but commonly associated with the efficient utilization of hardware. As the backends for high-level languages fail to deliver this, responsibility is then put in the hands of the programmer and the use of low-level languages for instrumentation of hardware. Consequently, programs become an entanglement of the application domain and hardware instrumentation requiring that the programmer has a full understanding of both.

This thesis explores exactly the tension between these concepts, specifically on bridging the gap between performance and productivity. How can the abstractions of high-level languages be maintained and the need for low-level languages avoided? How can a language backend exploit information from high-level abstractions as an aid to efficiently utilize hardware?

My primary contribution to the study of these problems lies with the publications in part II. This part of the thesis serves the purpose of putting those publications into context, providing the background for the work, the approach to studying it and outlining directions for future work. The thesis is organized as follows.

Sections 1.2 and 1.3 provide information about architectural traits of current, and next-generation processing units as well as an overview of the tools and challenges for programming them. Section 1.4 describes the approach I have taken, which in short evolves around the design and implementation of Bohrium a backend for array-oriented programming. The information from sections 1.2 and 1.3 serve as motivation of several choices in the design and implementation of Bohrium.

Bohrium is a collaborative effort, section 1.5 therefore provide an overview of my contributions to Bohrium as well as my contributions outside the context of Bohrium. Bohrium is language agnostic in the sense that it supports a programming model instead of a specific language. Chapter 2 describes this model. The programming model is array-oriented for which an essential implementation concern is the supported data-representation. Chapter 2 section 2.2 provides a brief overview of the data-representations that I have focused on.

Chapter 3 provides the most recent description of the state of Bohrium. Chapter 4 describes ongoing and future work on Bohrium as well as another research direction to bridging the gap between performance and productivity. The final chapter 5 concludes upon the work.

1.2 Computing Platforms and Architectures

Computing systems have evolved in utilization, efficiency and availability. The results of the early advances can be summarized in the concept of the Turing machine[68], the halting problem[68] and the von Neumann architecture[73]. These concepts provide the fundamental architecture of computing systems today, whether they are supercomputers or workstations.

The computing industry produces microprocessors under the driver that the amount of transistors/components on a cheap integrated circuit doubles somewhere between every 12[49] to 24[50] months. This trend in the fundamental building block for microprocessors has for a period of years had an associated doubling in clock frequency for general-purpose microprocessors. The frequency-scaling of the general-purpose processor has provided scalable performance of applications. An application-developer could expect that application-performance would scale linearly with the frequency of the next generation of a general-purpose processor.

The power and memory wall are two factors breaking this convenient expectation. As the frequency increases so does power consumption and the need for dissipating heat. The clock frequency of the general-purpose processor has peaked at around 4 GHz. Attempts at going above this boundary are mostly the domain of enthusiasts and involves clocking the processor to run above its specification and extreme cooling methods involving liquid nitrogen. Frequency scaling did have its issues prior to reaching the 4 GHz boundary due to a discrepancy between the speed of the processor and the speed of memory system. This is because production of processors has focused on increasing frequency, whereas the production of memory has focused on increasing capacity. The memory wall or von Neumann bottleneck refers to this discrepancy in the hardware design. General-purpose processors have, despite this, been providing scalable performance in concert with frequency scaling by integrating a complex multi-layered cache-hierarchy within the processor. The cache hierarchy also relies on complex on-chip units preloading data into cache thereby lowering the cycles wasted when waiting for data from main memory.

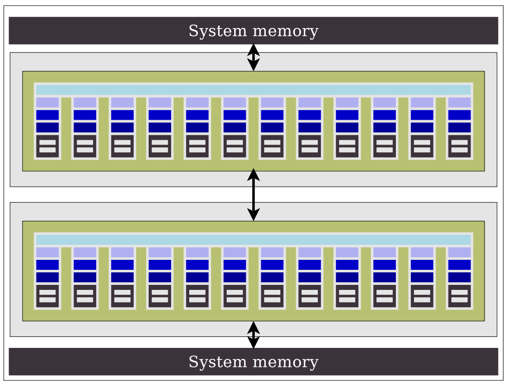
The following sections will describe the current trends in the design of processing units. Starting with the current design of the latest generation of Central Processing Units (CPUs) in subsection 1.2.1, followed by the currently very popular graphics processing unit (GPU) in section 1.2.2. These two processing units designs are currently the most widespread. Subsection 1.2.3 and 1.2.4 describe emerging architectures of the Accelerated Processing Unit (APU) and the Many Integrated Cores (MIC).

1.2.1 Multi-core CPUs and ccNUMA

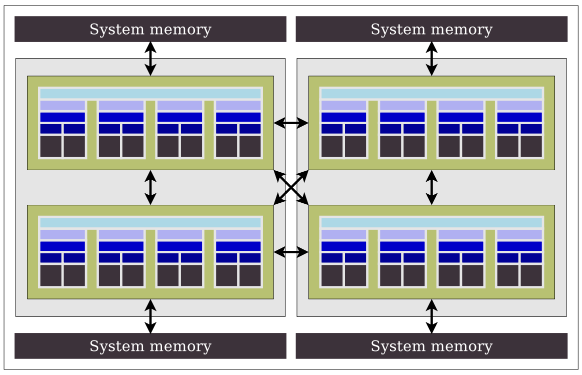
Vendors of general-purpose processors are using the growing amount of transistors available to cram multiple cores onto the same chip. That is, instead of scaling up the core frequency, they scale out the number of cores. Higher performance can then be achieved by having multiple processes or threads of execution thereby increasing the instruction throughput. Current architectures provide shared memory and symmetric multiprocessing (SMP) with cache coherence. Another similar model uses symmetric multithreading (SMT) where each core run multiple hardware threads to better utilize the core. The term multi-core processor covers both.

Shared memory architectures support a convenient programming model as it provides a single address space for all threads. This allows for efficient inter-process/thread communication.

A multi-core processor consists of homogenous cores in which part of the memory hierarchy is private. When threads running on different cores update the same values, private cached copies must be updated or invalidated, a choice left to the cache protocol of the processor. Scaling up the number of cores while maintaining cache coherence becomes increasingly difficult when expecting Uniform Memory Access (UMA). Non-Uniform (NUMA) is for this reason used in the latest generation of multi-core processors. Figure 1.2 illustrates two such designs.



(a) Two-socket configuration with two Intel Xeon E5-2650L. The processor package has a single die, the package is referred to as a NUMA node. Each node has twelve cores each with its own private L1 and L2 cache. L3 cache is shared with all cores on the same NUMA node. Each core can run two hyperthreads.



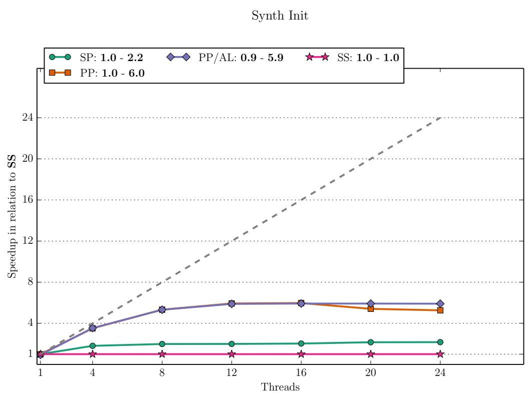
(b) Two-socket configuration with two AMD Opteron 6272 processors. The processor package has two dies, each die is referred to as a NUMA node and is comprised of four bulldozer modules. The bulldozer module has two cores each with its own private L1 data cache. L1 instruction cache and L2 cache is shared in the bulldozer module. L3 cache is shared with all cores on the same NUMA node.

Figure 1.2: Examples of multi-core processors with non-uniform memory access.

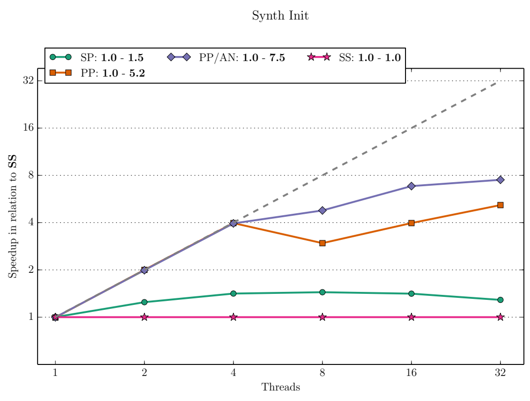
Obtaining efficiency from CPUs is not only about thread-level parallelism. For compute-oriented tasks CPUs rely on instruction level parallelism (ILP) by providing instruction set extensions for carrying out a single instruction on multiple data (SIMD). Technology pioneered by supercomputers was adopted and integrated into mainstream general-purpose processors. Different vendors like Intel, AMD, Cyrix provide different SIMD-like extensions as a competitive advantage to provide the most compute power of their x86-compatible processors.

Instruction set extensions to x86 have started out as MultiMedia eXtensions (MMX) by Intel, extended MMX (eMMX) by Cyrix which allowed a single instruction to be performed on multiple integers. AMD in response added the 3DNow! as an MMX extension for instructions on multiple single-precision floating point numbers. As increased floating-point compute capability became popular Intel replied with the Streaming SIMD Extensions (SSE) supporting floating-point operations. With each new generation, Intel has continued expanding their vector extension labeling them SSE/1/2/3/4/5 and now Advanced Vector Extensions (AVX/1/2). AMD have kept up and provide compatibility with the extensions but also introduce a revised version of SSE5 and their own extensions labeled XOP, FMA4, and CVT16. The vector-oriented extensions have been increasing the widths of FMA-instructions which execute two instructions in one clock-cycle such as multiply add and multiply subtract. The next generation of processors from Intel, Skylake and Cannonlake, will introduce 512-bit wide (AVX-512) instructions.

Thread-level and instruction-level parallelism are great advances that increase the compute-capability of CPUs. However, there is a downside as these advances on compute-capabilities lead to hitting the memory wall even harder. Current and next-generation multi-core processors are thus highly concerned with improving the memory subsystem. The performance consequences of the memory-wall are best described with an example.



(a) Results from running a synthetic memory-bound benchmark on an Intel 2650L with two NUMA nodes and a total of 24 cores.



(b) Results from running a synthetic memory-bound benchmark on an AMD Opteron 6274 with four NUMA nodes and a total of 8 cores.

Figure 1.3: Benchmark illustrating scalability challanges due to the memory wall and remote memory access on multi-core NUMA architectures. The graphs show speedup relative to the serial implementation labeled **SS.**

Figure 1.3 provide speedup graphs on a synthetic memory-bound benchmark on two different multi-core processors with NUMA. The benchmark initializes an array and then updates it. There are three implementations of the bencmark: serial initialization with serial access (**SS**), serial initialization with parallel access (**SP**), and parallel initialization with parallel access (**PP)**. The labels **PP/AN** and **PP/AL** denote two different strategies for controlling thread locality. Only the parallel access of the implementations are timed. The purpose of this experiment is to illustrate the scalability challenge of the memory wall and the effect of NUMA.

Comparing **SS** to **PP/AL** on figures 1.3a and **PP/AN** on figure 1.3b show how severely the memory wall limits scalability. On the 24-core system (figure 1.3a) a best-case speedup of 5.9 is obtained. On the 32-core system (figure 1.3b) a speedup of 7.5 is obtained. The memory wall is here and it is severe.

NUMA architectures aid scalability, however there are pitfalls a programmer must be aware of. Compare **SS** to **SP** on figures 1.3a, and 1.3b. Memory is initialized using `memset` which infers that memory will not be distributed among NUMA nodes. The consequence is that the majority of threads will suffer the negative effect of NUMA which is higher latency for retrieving data on another node. This completely thrashes performance on all the processors as the figures show.

A programmer can easily step into the NUMA pitfall, even when aware, and must explicitly manage thread locality. Figure 1.3b illustrates this most significantly. Compare **PP** to **PP/AN**, here two identifical implementations are executed, yet with different scalability. The difference is that threads are bound for the results labeled **PP/AN.** Bound in a manner that minimizes remote access/internode communication. Locality has become essential for multi-core performance due to NUMA.

1.2.2 General-Purpose Graphics Processing Units

Graphics Processing Units (GPUs) were historically devices dedicated to handle processing of the graphics intensive parts of an application such as rendering in Computer Aided Design (CAD) and real-time graphics for computer games. As the graphics processors evolved and became programmable through DirectX and OpenGL the raw performance of the GPU generated efforts in trying to use the GPU hardware for other areas than graphics. At the time the efforts were painstakingly hard as a computational science simulation must be expressed using the graphics pipeline and programmed with DirectX or OpenGL. This involved using pixel shaders as compute functions, setting up input as texture images and the output represented as a set of pixels generated by raster operations. The field was called GPGPU, for general-purpose computing on GPUs. With promising results within the field GPU vendors started to support it with the first major breakthrough being the Compute Unified Device Architecture (CUDA[55]) from NVIDIA.

Today the major vendors NVIDIA and AMD/ATI provide a broad range of support for GPGPU. NVIDIA continue to promote CUDA as the primary means for GPGPU on NVIDIA GPUs. AMD/ATI had previously promoted their own programming model STREAM[7] but has deprecated it in the favor of the open standard OpenCL[67] for general-purpose programming of their GPUs. Current GPU design still has hardware dedicated for graphics processing but the overall GPU architecture has unified into components which are usable for both graphics and general-purpose applications.

The latest shift from GPU vendor AMD was a complete change in GPU architecture. Previous generation have used VLIW4 architecture which is now replaced by what they label as the Graphics Compute Next (GCN) architecture. This was done to increase performance for non-graphics workloads, namely GPGPU.

General characteristics of current GPUs are that they contain small programmer controlled local memories and caches to boost memory throughput. GPUs are high-latency as they have simple control logic with no branch prediction or data forwarding. For compute power they contain a high amount of long latency, but heavily pipelined, ALUs for high throughput. The general design relies on a massive amount of lightweight threads to compensate for high latency.

A key trait of a GPU is that even though it is capable of performing branch instructions it is very poorly suited to do so due to high-latency and high costs incurred as a consequence of control divergence. The GPGPU term should therefore not be equated with the general-purpose programmability of a low-latency CPU. A key concern for efficiently utilizing a GPU is that it is connected with the CPU over the PCIe bus. The CPU and GPU are thus operating on physically separated memory systems. This is why GPUs are often referred to as *accelerators* or *coprocessors*, as they are used to offload specific tasks from the CPU to GPU. As mentioned the performance of the GPU is gained from massively parallel workloads which can be split into execution on a rich amount of SIMD units in the case of an AMD GPU and SIMT-based cores in the case a NVIDIA GPU.

1.2.3 Accelerated Processing Units

The design of multi-core CPUs struggle to increase performance by adding special-purpose instructions to increase compute capability of floating point operations and extending the memory system with the ccNUMA architecture. The vendor AMD introduced a different design labeled the Accelerated Processing Unit (APU) where a GPU is integrated into the same die as the CPU. One can see the APU as a CPU which has replaced its floating point unit with a GPU. This is not entirely accurate but the design idea is to gain compute performance from a GPU core dedicated to compute performance where a multi-core CPU obtains compute performance by having multiple cores with advanced floating point units. As described in the previous section the CPU and GPU operate on distinct memory spaces and one of the key challenges is to determine when to move data between them. The APU design solves this by providing a unified memory space for both the CPU and the GPU, hereby maintaining the convenient programming model of a shared memory space. This union of memory spaces has been labeled the heterogeneous Uniform Memory Access (hUMA). hUMA has the additional trait that memory access times are uniform in contrast to the NUMA memory system of the current multi-core CPUs. The devices themselves are programmed using OpenCL delivering the convenience of also having a single programming model regardless of whether the program is executed on the CPU or GPU core.

As I began my PhD studies no processing units existed using the APU architecture. The APU efforts of AMD reached the market in 2011 which in the first iteration was mostly just the successful engineering feat of allowing the CPU and GPU cores to co-exist on the same die. When programming them using OpenCL the memory spaces were still two distinct virtual memory addresses and the programmer had to manually map a pointer in CPU memory space to that of the GPU's memory space. This is however just a mapping of namespaces and does not require copying of data. The first APU (codename Kaveri) to take full advantage of the Heterogeneous System Architecture (HSA) was released in early 2014. As of 2015 the roadmaps of AMD are pointing in the direction of APUs in a wealth of configurations for different applications.

1.2.4 Many Integrated Cores and Network On Chip

The architecture of the Many Integrated Cores (MIC) and Network on Chip (NIC) is somewhere in between the design-space of CPUs and GPUs. The MIC architecture uses simpler processing cores, compared to the previously described multi-core processors, requiring fewer components and thereby allowing for a higher number of cores on the same die. The interconnect between the cores is the central focus of the architecture.

Xeon Phi is Intel's flagship product for their MIC-design. The design features a throughput-oriented coprocessor with its own memory connected to the system via the PCIe bus much like the GPUs. The instruction set is, unlike GPUs, based on x86. In this architecture, a high performance bidirectional ring network connects multiple in-order cores providing fully coherent L2 caches. Each core integrates a Vector Processing Unit (VPU) with 32-bit wide SIMD units supporting fused-multiply accumulate (FMA). Each core supports four threads using symmetric multithreading (SMT) model labeled Hyperthreading. They hide latency by switching execution when a cache-miss occurs, and a hardware-prefetcher to prefetch cache data to avoid cache-misses. The latest product the Xeon Phi 7120 scales the core-count up to 61 cores running at 1.238 GHz with access to 16 GB GDDR5 ram.

Tilera is the producer of multiple MIC-based processors. These processors are, unlike the Xeon Phi, not designed to be coprocessors but rather full-featured processors on their own, capable of efficiently running an operating system. Tilera provides their own instruction-set for their processors which are not x86 compatible. The Tilera design consists of identical cores placed in mesh-network with L1 cache and a fully coherent L3 cache. Tilera has a specialized technology named Dynamic Distributed cache (DDC) to facilitate the cache coherence. Tilera claims that it accelerates coherent cache performance by a factor of two compared with other multi-core interconnects. Their latest product the Tile-Gx8072 scales the core-count up to 72 cores running at 1.0-1.2 GHz with four integrated memory controllers capable of accessing up to 1 TB of DDR3 memory.

Epiphany is a MIC-architecture which was designed from the ground up by Adapteva as a crowd-sourced project with the goal of creating a non-legacy architecture with vast scalability. The Epiphany[33] architecture is a coprocessor design much like the Xeon Phi but features a mesh-based interconnect like the Tilera. In Epiphany, the mesh-network connects nodes. Each node contains a RISC CPU, DMA Engine, Local Memory and a Node Interconnect Interface. The interesting architectural difference is that the Epiphany provides one big flat shared memory space, but there is no cache-hierarchy and, therefore, no cache coherency to maintain. The first epiphany based product named the Parallela is a System on Chip (SOC) which became available October 2013. It features a dual-core ARM A9 and a 16-core Epiphany coprocessor with access to 1GB of DDR3 memory.

The MIC/NOC architectures do not distinguish between the chip as a coprocessor or a as full-featured processor. Their general trait is the network-oriented interconnect between the cores. A recent example of this trend is that Intel has announced that the next generation of Xeon Phi (Knights Landing) will be available as both a coprocessor and a standalone CPU.

1.2.5 Supercomputers

In the search for an answer as to what the best performing next generation processor will be, and thus the most interesting for the backend to target, one has historically been able to inspect the technological traits of the TOP500[1] list of supercomputer sites in the world.

As of June 2015 the first place on the TOP500 list is held by the Tianhe-2 (MilkyWay-2) supercomputer. It uses a x86-based multi-core processor, the Intel Xeon E5-2692, along with the Intel Xeon Phi 31S1P coprocessor. Following it, at second place, is the Cray XK7 based supercomputer Titan. Titan features an AMD Opteron 6274 multi-core processor along with an NVIDIA K20X GPU. The computation nodes of current supercomputers have thus adopted mainstream general-purpose commercial-of-the-shelf (COTS) components. While this does not provide much inspiration as to which features next generation processors can adapt, it does testify that the current mainstream ccNUMA architecture is useful for high performance computing tasks insofar as it is used for these tasks. The most inspiring asset of today's two fastest supercomputers is that they feature heterogeneous nodes. The Tianhe-2 uses the Xeon Phi coprocessor, and the Titan uses an NVIDIA based GPU. If today's supercomputers are an indication of tomorrow's processors, then it is certain that the next generation processors will be heterogeneous. One indication is that since today's top-performing super-computers are clusters of COTS components then the distributed shared memory programmed using MPI[66]/PGAS is an attractive means of obtaining scalable performance. Other approaches such as the SGI Altix machines have a completely different design which scale up into one huge machine programmable via shared memory although with multiple layers of NUMA. The SGI machines can scale up to 2048 cores and 16TB of memory. The core count is equivalent of 128 Cray XK7 nodes, and the memory of a single SGI machine is equivalent to 512 Cray XK7.

The primary barrier, regarding heterogeneity is the physical separation of memory spaces, the inconvenient handling of non-shared memory spaces and the performance penalty for transferring data between the device and main-memory. The latest generation of the Xeon Phi integrates the coprocessor on the same die as the CPU cores, turning the Xeon Phi into a full-fledged processor. It thus seems like the next generation processors will be heterogeneous, consisting of multiple coprocessors integrated on the same die.

1.2.6 Summary

The memory wall and effects of NUMA are the key motivations for hardware vendors exploring different architectures and designs. It is hard to predict the future, and the current divergence in processor design from the major vendors does not make it easier. Thus, designing a backend for data-centric applications on next-generation processing units, one cannot focus on targeting just a single architecture. The design of multi-core CPUs, APUs, and MICs all point in the direction of shared memory model featuring cache coherence with varying levels of non-uniform memory access.

The most opposing trend in this regard is the physically separated memory spaces of accelerators such as discrete GPUs. These drag in another direction, pulling with the force that is their superior throughput in terms of floating point operations. Regardless of whether the memory system is shared or distributed the common trend is that next-generation processing units and

[1]The list is updated every six months and available online at http://www.top500.org/

computing systems will be heterogeneous. It must be the focus of the backend to be able to handle the current heterogeneity of computing systems as it is a lasting trait of processing units and computing systems of today and the foreseeable future.

1.3 Backends and Languages

This section outlines current approaches to programming computing systems for high performance. This includes the tools available for programming the TOP500 supercomputers as well as high-end workstation.

The mainstream languages C, C++ have very limited support for expressing parallelism. One explanation for this is that the languages themselves predate thread-level parallel processing. Compilers and interpreters for these languages do to some extent provide ways of utilizing parallel hardware but mainly for instruction-level parallelism. These languages are regardlessly popular weapons of choice when it comes to programming parallel hardware through the use of language extensions, libraries and runtime systems.

1.3.1 Low-level APIs

pthreads: Posix Threads

Posix threads (Pthreads[53]) are a standardization of threading libraries for symmetric multiprocessors. Historically vendors have each provided proprietary threading-libraries for their own processing units which hindered portability from one CPU to the other and in some cases from one generation of a CPU to the other. The IEEE POSIX 1003.1c standard provides a standardized C language threads programming interface for Linux and other UNIX-like systems. Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads. Pthread compatible threading libraries provide the fundamental building block for parallel computation the library contains functions for creating/killing a thread and synchronization between threads of a process.

Pthreads portability breaks when it comes to using multiple platforms. Synchronization primitives such as barriers are defined as optional which leads to uncertainty as to whether or not synchronization barriers will be available when deploying an application based on Pthreads. The lack of operating system portability is due to tight bounds between the multi-tasking support and process/thread model of the operating system.

A widespread, well-supported, and highly portable alternative for multithreading is available and named Open Multi-Processing (OpenMP[22]). OpenMP provides a runtime API for encapsulating the concepts of multithreading but without the concerns of mapping to the exact vendor-provided threading-model. OpenMP is described in greater detail in section 1.3.2 on Compiler Directives.

Qthreads: An API for Programming with Millions of Lightweight Threads

Qthreads[75] provide abstraction that enable development of large-scale multithreading applications on commodity architectures. Qthreads was designed and implemented to unify threading models for emerging architectures with large scale hardware supported multithreading. It provides a lighter threading model compared to other threading models such as pthreads. A key feature of Qthreads is the ability to describe locality encapsulated in the concept of shepherds. Which as described previously is essential for performance on NUMA architectures.

CUDA: Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model introduced by NVIDIA for using their GPUs for GPGPU purposes. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. Using CUDA, the latest NVIDIA GPUs become accessible for computation like CPUs.

The CUDA platform is accessible to software developers through C, C++ and Fortran libraries. The CUDA programming model revolves around kernels which are self-contained functions implemented in CUDA C/C++ or CUDA Fortran. The CUDA-API provides the means for JIT-compiling kernels using one of the supported compilers. The implementation of the kernels themselves evolve around a programming model NVIDIA has labeled Single Instruction Multiple Threads (SIMT). The model is closely related to SIMD but has, due to the CUDA architecture, essential low-level differences in relation to thread-scheduling. The lowest scheduling unit in CUDA is a warp. A warp is a set of threads scheduled to execute in parallel and all threads within the warp are executing exactly the same instruction. It is thus not a single instruction mapped to multiple data elements (SIMD), but a single instruction mapped to multiple threads (SIMT); the threads themselves decide which data element to operate on. Each thread has a unique position (blockIdx, threadsIdx) in a three-dimensional organization partitioned into blocks of threads. In the SIMT-model, each thread uses its coordinate to map to the data-elements.

The warp-size is supposed to be an implementation detail that the programmer should not be aware of, but for many optimizations on the CUDA platform knowing the warp-size is essential for obtaining high performance. Another low-level concern is how and when to transport data between the main memory of the host and the device memory of the GPU.

OpenCL: Open Computing Language

Open Computing Language (OpenCL) can briskly be described as the open alternative to CUDA. OpenCL is the low-level API to use when targeting AMD GPUs. OpenCL includes a language (based on C99) for writing kernels (functions that execute on OpenCL devices), plus an API to declare which devices to use, allocate and copy data on and between devices, and execute kernel-functions on the compute devices. The goals of OpenCL do stretch quite a great deal further.

OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting not only of GPUs but also of CPUs, APUs, MICs, and Digital Signal Processors (DSP)s . The goal of OpenCL is to provide *the* platform for parallel computing using task-based and data-based parallelism. Major vendors including Intel, Qualcomm, AMD, NVIDIA, Altera, Samsung, Vivante, and ARM has adopted and actively support OpenCL. Although the support from NVIDIA seem to be faltering.

The previously mentioned families of processing units Opteron, Xeon Phi, Radeon HD, APU, and Epiphany are all programmable using OpenCL. However, since OpenCL is a low-level interface, the programmer must still explicitly control the hardware to the extent that OpenCL allows. Therefore, it follows that the performance of a kernel written in OpenCL is not portable across processing units. One example is a kernel written for a GPU which implies a kernel written to utilize around 1500 threads. Instantiating 1500 threads on a CPU which at the hardware level only supports 16 threads will only ensure horrifically slow execution. OpenCL is nonetheless a very promising and convenient low-level API for instrumenting parallel execution.

MPI: Message Passing Interface

The TOP500 super-computing sites have previously had a mixture of architectural differences for obtaining scalability. Previously, SGI-like architectures have dominated the TOP500 sites, but today the design have converged towards large-scale, custom-build systems based on the commercial-of-the-shelf components such as the systems designed by Cray, Inc. Along with this convergence the Message Passing Interface (MPI) has become the de-facto standard for distributing work-loads among the nodes of a cluster regardless of its size.

Message Passing is often used with the Single Program Multiple Data (SPMD) model. When using MPI a number of processes run and execute the same program. MPI / SPMD are thus a great deal coarser grained in its decomposition than the SIMT/SIMD models. The processes communicate by passing messages; there is no notion of shared memory. Each process has a rank which is much like the threadIdx/blockIdx of CUDA; it is a unique identifier of the process in the set of processes running the same program on multiple machines. The rank is thus the key used to direct the behavior of the program and decompose the problem domain.

PGAS: Partitioned Global Address Space

Partitioned Global Address Space (PGAS), also known as distributed shared memory, is a design around a communication primitive for maintaining a shared memory model in a distributed setting. This is in sharp contrast to MPI in which all communication is based on passing messages and shared memory is not possible. The PGAS model is used in the same setting as MPI for large-scale computational clusters. Accessing a value in distributed memory is significantly slower than access to a value in local memory. The PGAS model encapsulates this by provided one big global address space but conveniently provide constructs for determining whether a given operation will be accessing local or distributed memory.

The GASNet communication system is an example of a building block implementing this distributed memory model. It is however intended for use in compilers and runtime-system and not for end-users.

BLAS: Basic Linear Algebra Subprograms

BLAS[2] was originally a Fortran library containing 38 low-level subprograms for many of the basic operations within numerical linear algebra. The operations included dot product, elementary vector operation, Givens transformation, vector copy and swap, vector norm, vector scaling, and the determination of the index of the vector component of largest magnitude. BLAS has, since its first publication in 1979, gone from being just a Fortran library to become the defacto library interface for performing operations within the scope of linear algebra. The BLAS operations today consist mainly of three levels.

Level 1 Vector expression on the form: $y \leftarrow ax + y$ where x, y are vectors and a constant. As well as dot products and vector norms.

Level 2 Matrix/Vector expression on the form: $y \leftarrow \alpha Ax + \beta y$ where A is a matrix, x, y are vectors and α, β are constants.

Level 3 Matrix/Matrix operations on the form: $C \leftarrow \alpha AB + \beta C$ where A, B, C are matrices and α, β are constants.

The Netlib[2] provides a C-reference implementation of BLAS and there are optimized versions of the BLAS libraries for almost all hardware targets. Both enthusiasts and vendors supply the implementations of these libraries. To name a few Intel provides the Math Kernel Library (MKL[1]) targeting their multi-core CPUs and the MIC. AMD supplies Core Math Library (ACML) targeting their Opteron CPUs and Accelerated Parallel Processing Math Libraries (APPML) targeting their GPUs. NVIDIA provides cuBLAS[5] a complete BLAS implementation for dense matrices and a cuSPARSE[51] for a dedicated subset for sparse matrices, both targeting CUDA-based GPUs. GotoBLAS[32]/OpenBLAS[77]/GotoBLAS2 are open-source initiatives currently led by Texas

Advanced Computing Center (TACC) targeting Intel Nehalem, Intel Atom, VIA Nano, AMD Shanghai, and AMD Istanbul. There currently exists an abundance of BLAS-libraries targeted for virtually any architecture.

Another very popular BLAS approach, which has had significant impact on high performance computing, is Automatically Tuned Linear Algebra Software (ATLAS)[74]. Microbenchmarks help estimate attributes such as cache-sizes which can then be used to provide optimal values for block sizes for tiling. Additional micro-kernel benchmarks are executed to determine optimal values for each BLAS operation. The autotuning approach allows for highly tuned BLAS-libraries as even small variations within a CPU architecture can be measured, modeled and integrated with the optimized BLAS-operations.

Build To Order (BTO)[63] is another interesting approach to obtaining an efficient BLAS library implementation. BTO is a BLAS-expression compiler that generates high performance implementations of basic linear algebra kernels. The user of the Build to Order BLAS compiler writes down a specification for a sequence of matrix and vector operations together with a description of the input and output parameters. The BTO-compiler then tries out many different choices of how to implement, optimize, and tune those operations for the available hardware. The result of this process is output as a C file containing a function that implements the specified composition of the operations described by the user. This approach is quite interesting as it allows to optimize, not only for particular hardware, but also for expressions.

1.3.2 Compiler Directives

Libraries augment the functionality of a language. They do not extend the language itself as they are not part of the grammar, but they do extend what can be done within the language such as providing control over processing threads and co-accelerators such as low-level APIs from the previous chapters. This attribute is both a force and a weakness. Another approach to extending a language is using compiler/interpreter directives. Directives are similar to a library in the sense that they are not part of the language grammar. Directives are hints, tags, decorators or pragmas inserted in source-code providing directives for the compiler to follow or ignore. The following descriptions cover some widespread and popular directive-based frameworks.

OpenMP[22] is one of the most widespread and well-supported compiler directives for C, C++, and Fortran compilers. OpenMP provides a convenient framework for orchestrating and synchronizing threads on multi-core processors. OpenMP supplies pragmas such as `#pragma parallel` which when in front of an anonymous code block will execute the code block in its own thread. Another construct `#pragma parallel for` provides a simple means for expressing data-parallelism. When `#pragma parallel for` is put in front of a for-loop the compiler will attempt to perform a fork-join parallelization over the loop-body. OpenMP is a standardized API which is integrated with C/C++/Fortran through directives. Support for coprocessors and accelerators have been added in OpenMP 4.0.

OpenACC[76] is a collection of directives for code-blocks and loop-bodies similar to OpenMP. With OpenACC these code-blocks and loop-bodies are targeted execution on coprocessors and accelerators such as GPUs. The benefit of using OpenACC directives in comparison to OpenCL/CUDA is that it encapsulates device initialization and data movement between host and accelerator. It simplifies many of the tasks required for offloading computations to an accelerator/coprocessor.

LEO[52] are Intel's Language Extensions for Offload, they provide offload capabilities for Intel Graphics and the Xeon Phi accelerator and coprocessor. Adding `#pragma offload target(mic|gfx)` will execute the codeblock following on an accelerator or Intel graphics. Adding `#pragma offload_transfer target(mic)` provide control for memory management on accelerators. LEO works with OpenMP to provide both task-based and loop-centric parallelization when targeting an accelerator. This simplifies porting already OpenMP parallelized code to run on the Xeon Phi. However, when used to target Intel graphics then only perfect loop-nest are offloadable and OpenMP is not supported.

OmpSs[27] abstracts parallelization to one level higher than that of OpenMP and OpenACC by providing task-based programming model. OmpSs targets the programming of heterogeneous and multi-core architectures and extends OpenMP 3.0 by offering asynchronous parallelism in the execution of the tasks. The main extension provided by OmpSs is the concept of data dependencies between tasks and not just the fork-join based parallelization of the `#pragma parallel for` or `#pragma acc loop`. This is done through directives such as `#pragma omp task in(...) out(...)`. This information is used during the execution by the underlying OmpSs runtime to control the synchronization of the different instances of tasks by creating a dependence graph that guarantees the proper order of execution. This mechanism provides a simple way to express the order in which tasks must be executed, without needing to add explicit synchronization.

Directive-based programming is a very promising technology for dealing with heterogeneous architectures.

1.3.3 Libraries

BLAS is an example of a successful high performance library to such an extent, that today it is not just a library but the defacto interface for linear-algebra. It conveniently hides the low-level details from the user by delegating the task of mapping BLAS-operations to the library provider. However, BLAS does have its short-comings when it comes to ease of use, but more importantly the BLAS interface is static. Optimizing a composition of BLAS-operations implies extending the interface with the composition such as the approach of the BTO-BLAS compiler.

The following listing describes current approaches to maintaining performance of libraries and to some extent increasing it by allowing composition of operations.

Blitz++[71] is a C++ template library for array manipulation. It exposes a single class called `blitz::Array<T_numtype,N_rank>` which provides a dynamically allocated N-dimensional array. The implementation is based on utilization of expression templates for performance. Expression templates facilitate lazy-evaluation at compile-time. Any expression performing operations on one or more `blitz::Array` generate a tree-like data structure of expressions. The expression structure is evaluated at compile-time and provides means for applying optimizations such as loop fusion, unrolling, tiling/blocking, and algorithm specialization. The high-level abstraction of arrays provides a convenient declarative notation effectively shielding the user from low-level optimizations. Blitz++ is being used in C++ projects but also as the backend for applications written in high-level languages such as R and Python.

Armadillo[60] is a C++ linear algebra library with a structure much similar to Blitz, but with an emphasis on ease of use by providing syntax similar to that of Matlab and Octave. It uses expression templates similarly to Blitz++ to obtain performance and also provide various matrix decompositions integration with LAPACK[3], or one its high performance drop-in replacements, such as MKL from Intel or ACML from AMD. Performance comparisons[60] suggest that the library is considerably faster than Matlab and Octave. Armadillo also provides integration with the programming language R. The integration was demonstrated at the R/Finance[2] conference in Chicago, IL, USA May 2013 showing a speedup of 66 on the same hardware for a Kalman Filter application written in R.

[2]R/Finance: Applied Finance with R, http://www.rinfinance.com/

Eigen[35] is yet another C++ library for linear algebra based on expression-templates with a focus on ease of use. The library provides very efficient utilization of a single CPU core. Current work on Eigen focuses on expanding support for multi-cores by delegating to BLAS libraries and further down the road orchestrate the parallelization within Eigen explicitly using OpenMP.

Thrust[8] is a C++ template library maintained by NVIDIA for parallel platforms based on the Standard Template Library (STL). Thrust allows the user to implement high performance parallel applications with minimal programming effort through a high-level interface that is fully interoperable with technologies such as C++, CUDA, OpenMP, and Thread Building Blocks (TBB). Thrust provides a collection of data parallel primitives such as *scan*, *sort*, and *reduce*, which can be composed to implement complex algorithms with concise, readable source code. The user effectively delegates the task of selecting the most efficient implementation to Thrust by describing computations in terms of high-level abstractions. Thrust specifically provide an STL-interface compatible with `std::vector`. The interface quite efficiently hides the low-level details of multi-core threading and GPU kernel-code generation. The STL-interface is provided through two `std::vector` compatible containers: `thrust::device_vector` and `thrust::↩ host_vector`. Thrust does not hide the essential challenge of deciding when to move a problem to the GPU. Thrust is thus specifically designed for a coprocessor architecture in which the host can be a general-purpose multiprocessor and the device a the coprocessor/accelerator. Users of Thrust, must decide when to use a device/coprocessor and when to use the host/multi-core for executing the composed STL-algorithm.

Bolt[59] is a C++ template library maintained[3] by AMD which provides an STL compatible library of high level constructs for creating accelerated data-parallel applications. Bolt can briefly be described as the AMD equivalent to Thrust – targeting the same architectures – but using OpenCL instead of CUDA. The high-level abstraction is very similar and, like Thrust, Bolt also exposes the locality of the arrays by distinguishing between a host and a device vector.

Cilk Plus[17, 57] is not only a library but also an extension to C and C++ that offers a quick and easy way to harness the power of both multi-core and vector processors. Cilk Plus provides parallel constructs in the form of three keywords `cilk_spawn`, `cilk_sync`, and `cilk_for`. The three keywords provide a simple model for explicit parallel programming while runtime and template libraries offer a well-tuned environment for building parallel applications. Cilk Plus consists of language extensions, a library and the runtime system to support it. Cilk requires compiler support in order to provide an array notation of unprecedented simplicity compared to purely library-based approaches. The Intel compiler supports Cilk and so does branches of gcc 4.8 and llvm.

These libraries have seen widespread use, and their popularity can be credited to their ease of use granted the declarative programming style based on operator overloads in the case of Blitz++, Armadillo, and Eigen and the STL-compatible interface of Thrust and Bolt. An observation regarding these libraries is that they fail to hide hardware specifics when it comes to utilizing separate memory spaces. Thrust and Bolt both expose this by requiring the user of the library to decide whether a given operation should operate on data on the device or on the host. Blitz++, Armadillo, and Eigen do not yet support execution on other targets than the CPU and main memory it is uncertain how these libraries will address this issue.

[3]Latest information available via https://github.com/HSA-Libraries/Bolt

1.3.4 Languages

The previous sections have mostly been concerned with C/C++/Fortran and the low-level APIs, libraries, and compiler directives available within them. The reason is that C is a fundamental building block. C is *the* language for systems programming. C provides the low-level control needed for constructing low-level APIs, libraries, and for controlling hardware. The C language provides structures to encapsulate related data, and MACROs for preprocessing and transforming source-code. The C language thus has very limited means for providing high-level abstractions for the user which makes it ill-suited for application-programming.

C++ with its integrated legacy support for C has been, and still is, a popular choice for encapsulating the low-level details of C by means of classes, operator overloads, and STL-container interfaces. C++ is even popular within Computational Financial as one of the leading educational institutions Carnegie Mellon teaches C++ as part of the curriculum in their master program in Computational Finance. Fortran has been and still remains the weapon of choice within the natural sciences for expressing computational experiments such as simulations and model-testing.

The following describes the approaches of more high-level languages concerning productivity and performance. Substantial research effort has been put into increasing programmer productivity for large-scale clusters and supercomputers. The challenge was to replace a programming model based on X+ MPI where X is one of C, C++ or Fortran with something that increased the productivity of the programmer. The earliest was High Performance Fortran (HPF), later ZPL[19]. The results of the efforts are the PGAS languages which provide convenient means for supporting the SPMD model in a distributed environment. PGAS is supported in a different form by Unified Parallel C (UPC[18]), CoArray Fortran (CAF[54]), IBM X10 (X10[61]), and Chapel[20, **?**].

HPF High Performance Fortran is a high-level data-parallel programmig system based on Fortran. It was one of the first attempts at creating a high-level language abstracting the details required by X + MPI approach. Although not successful HPF had a major impact on the design and implementation of data-parallel languages. A postmortem[37] of the language summarizes lessons learned from the rise and fall of HPF for future languages to consider in their design.

ZPL Z-level Programming Language is a parallel array programming language designed from first principles for fast execution on both sequential and parallel computers. ZPL can achieve efficiency comparable to hand-coded message passing by exploiting the latent parallelism of array operations. ZPL is not a PGAS language but the concepts introduced by ZPL such as regions live on in Chapel.

UPC Unified Parallel C is one of the earliest examples of a PGAS language, which combines the convenience of a global address space with the locality control and scalability of user-managed data partitioning. Any code which forms a valid C program is also valid UPC program. UPC is thus tightly coupled with the low-level nature of C. UPC provide high performance by hiding latency through single-sided communication but does little to advance the productivity of the programmer.

CAF CoArray Fortran is based on many iterations of work on Fortran. Fortran has long been a favorite among scientific programmers. For that reason, it has always been viewed as an important language to map to scalable parallel systems. CAF 2.0 is the current peak of the efforts based on the works of High Performance Fortran (HPF) and CoArray Fortran 1.0. CAF 2.0 is a partitioned global address space programming model based on one-sided communication, is a coherent synthesis of concepts from MPI, Unified Parallel C, and IBM's X10 programming language.

X10 is from IBM and named based on a mission statement when IBM set out develop a petaflop computer, which could be programmed ten times (hence X10) more productively than a computer of similar scale in 2002. The roadmap of the X10 team was to develop a programming model for large scale, concurrent systems that could be used to program a wide variety of computational problems, and could be accessible to a large class of professional programmers. The result is X10: a modern language in the strongly typed, object-oriented programming tradition whose design fundamentally focuses on concurrency and distribution, and which is capable of running with good performance at scale. UPC came from C, CAF from Fortran, and X10 came from Java and has a Java inheritance.

Chapel is an emerging parallel programming language that strives to improve the productivity of parallel programmers from desktops to supercomputers. It is the only one of the PGAS languages which provides its own grammar and do not come with a legacy such as C, Fortran or Java. It also strives to support more general styles of parallelism in software and hardware. Cray Inc. leads the design of Chapel in collaboration with members of academia, computing centers, and industry. It is developed in a portable, open-source manner under the BSD license. A key feature of this philosophy is that higher-level features are implemented within Chapel in terms of the lower-level concepts, ensuring that the various levels are compatible and at every level support composition of concepts. Chapel thus supports both low-level instrumentation of communication and locality while at the same time providing a convenient syntax and ease of use in terms high-level language constructs. It is as such the best of both worlds.

The above mentioned languages are based on PGAS and highly focused on large-scale computational clusters and supercomputers. They mainly target distributed memory systems. A wealth of other parallel programming languages exists which to a greater extent focus on obtaining high performance on a single node or workstation and utilizing parallelization at the node/workstation level. Such languages include but are not limited to Julia[10] and NESL[12].

NESL is a parallel language developed at Carnegie Mellon. The major focus of NESL is to provide a convenient and high-level data-oriented programming model. NESL advocates the expression of algorithms in a high-level form and hereby extract nested data-parallelism. NESL consists of a runtime system which executes virtual machine code (VCODE[13]). The NESL project itself seem to have stopped development around 1995-2000. However recent work has shown that NESLs model of nested data- parallelism is also highly applicable to efficient utilization of GPUs[9]. The group researching domain specific languages in the HIPERFIT context also investigate the use of NESL and nested-parallelism targeting GPUs.

MATLAB is a high-level language and interactive environment for technical computing. Matlab has a long history dating back to the 1970s where it was introduced in academia as alternative to Fortran and LINPACK[25]/EISPACK[31]. The name Matlab is an abbreviation of **Mat**rix **Lab**oratory which emphasizes the use of an array-oriented programming model where all variables are arrays/matrices. Matlab thus provided a convenient matrix-notation for performing linear algebra and mapped these to Fortran and calls to LINPACK/EISPACK. Today Matlab has widespread use in academia within domains of economics, science, as well as industry. It is today implemented in C and LINPACK/EISPACK has been replaced by LAPACK[3].

IDL IDL is an interactive data language that have seen widespread use in areas such as astronomy, atmospheric physics and medical imaging. It features a high-level declarative syntax and applies well to an interactive processing of large amounts of data. IDL is a commercial product though free implementations of the language exist most notably the GNU Data Language.

Julia is a high-level, high performance dynamic programming language for technical computing with syntax which is familiar to users of MATLAB, Python, and R. It provides a sophisticated llvm-based compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function library. The library, largely written in Julia itself, also integrates mature, best-of-breed C and Fortran libraries for linear algebra, random number generation, signal processing, and string processing. Julia provides parallel constructs such as `@spawn` and `@parallel` for local multi-core parallelization. The constructs `@distribute` and `@localize` for distribution on clusters. Julia mainly targets multi-core processors and grids or clusters of multi-cores. However as the CUDA compiler has now been contributed to the LLVM compiler project it could mean that Julia's compiler through the llvm-code code target CUDA based GPUs.

Another approach to uniting productivity and performance is to directly map low-level libraries into high-level languages. Below are two examples of this approach described.

R is a programming language and software environment for statistical computing and graphics. It is widely used among statisticians and data miners for developing statistical software and data analysis. It has also widespread adoption within mathematical/computational finance at the University of Copenhagen where it is used as an integrated teaching tool. The university of Washington similarly use R as part of the curriculum of their Master of Computational Finance program. R can be regarded as a domain-specific language for statistical computing, and it has very little focus for generic parallel constructs or high performance. However, there are multiple efforts of integrating R, via C++[28], with libraries such as Eigen[6] and Armadillo[29] as a means of increasing the computational capabilities of applications written in R.

Python is a high-level, high productivity dynamic interpreted programming language. Python has a philosophy of batteries-included which means that along with the distributions of the Python interpreter follows a rich standard library. The design of Python is thus to a have a somewhat small and simple language with a high focus of productivity and easily readable code. The language itself does not contain parallel constructs but relies on libraries for these purposes, libraries which are included with the distribution of the interpreter. Python has a suite of libraries consisting of Numerical Python (NumPy[69]), Scientific Python (SciPy[36]), Interactive Python (IPython[56]), and MatplotLib[34] which combined provides high-level interface similar to that of MATLAB with a base abstraction of a multi-dimensional array structure. Python has gained much popularity due to its simplicity and well-supported libraries especially within scientific computation communities. Python has also proven itself in finance communities Python has multiple interpreters with different performance characteristics, the most popular is the reference implementation CPython. CPython integrates well with C-libraries and the NumPy library also gains it performance boost by providing an efficient backend implementation of the array operations exposed in the library.

This ends the background describing the architectural diversity of current and near-future processing units and the multiple levels of abstraction the processing units themselves and the computing system they are part of can be programmed with.

1.4 Approach

As I started my Ph.D. studies the HIPERFIT initiative had just begun and the different research areas ran in parallel. As a consequence, there was no domain specific language for the backend to support. The backend instead became language agnostic by circumstance rather that design. However, what was given from the HIPERFIT context was the intent of creating a high-level domain specific language, most likely in the shape of a functional language.

With that in mind I looked at related work on programming systems for high performance and productivity as the previous two sections describe. The result was the following design criteria and goals.

- Support an array-oriented programming model
- Language integration via intermediate representation
- Target a performance that is comparable to straight forward hand-coded C/C++ for the same application

The motivation behind the focus on array-oriented programming is manifold. Considering productivity the widespread and actively used high-level languages such as Python/NumPy, Matlab, R, and Julia all use arrays and a convenient declarative notation for manipulating them. Without knowing exactly how the domain-specific language would look like it seemed safe to assume that it would provide abstractions similar to those just described. The existence of purely functional array-oriented languages such as Single-Assignment-C[62] also testifies that the model does not conflict with the intents of designing domain-specific languages using functional language design. From a performance perspective, history has shown that data-parallelism drives performance[19, 14]. In short the array-oriented model allows for convenient notation within the language and the backend can exploit the inherent data-parallelism of array operations for performance. Focusing on a programming model instead of a language was a way to bootstrap the studies.

What started out as a condition of circumstance became a novelty of the work. It allowed the exploration of using an intermediate representation for array operations as the bridge between language and backend and thereby building the bridge between performance and productivity. The three criteria formed the thesis queston: Is it possible to construct a language agnostic backend for high-level languages without sacrificing performance?

I have applied an experimental[23, 24] approach to explore this question. The contributions I have made in this regard are described in the following section. The experimental environment included the specification and construction of an eight-node beowulf-cluster. The specification was inspired by the current trends as described in section 1.2 and resulted in a node-configuration with two Opteron 6274 CPUs, 128GB of memory, three Radeon HD 7850 GPUs, and gigabit ethernet. A configuration that encapsulated several programming challenges for shared memory multi-core processors with NUMA architecture, distributed memory, GPUs, and the general trend of heterogenous architectures.

As previously mentioned then the design and implementation of the backend is a collaborative effort. The cluster encapsulates several performance related challenges where the focus of my studies is the efficient utilization of shared memory multi-core processors with NUMA architecture. That means exploring whether the declarative array-oriented programming provide sufficient information for a backend to efficiently manage non-uniform memory access without help from the programmer. Efficient utilization of other processing units as well as distributed memory are also of interest but out of scope for my work.

Focusing on multi-core processors also aid experimental testing and performance evaluation as it allows for comparative studies to existing languages, libraries and tools. The cluster served as a laboratory and to perform actual experiments in that laboratory I implemented a tool named Benchpress which is described in Chapter 4 section 4.5. Modern software engineering practices were applied to maintain good research conduct and laboratory practice. Including providing all source code as open source in publicly available repositories and using continous integration tools for correctness testing.

1.5 Contributions

The core of the thesis is based on the design, implementation and experimental evaluation of a backend for array-oriented programming which resulted in 12 papers that I have co-authored throughout my studies. 10 of which are published, one submitted but not yet published and one not yet submitted. My main contribution to this work is the C-targeting Array Processing Engine (CAPE) which I have described in the paper *Automatic mapping of array operations to specific architectures[48]* (see Part II section 6.1) submitted for publication in the Elsevier International Journal of Parallel Computing, ref: PARCO-D-15-00170.

Together, they describe the incremental steps in implementing a prototype, evaluating the performance, making observations, getting feedback, revising the design, revising the implementation and then re-iterating. That is, they describe performance related issues with different approaches culminating in the current state of the array-oriented backend Bohrium and the array processing engine CAPE.

The remainder of this section will go through the publications and describe my role in the work. The end of the section describes a divergence exploring interoperability with the parallel programming language Chapel.

The first experimental prototype was named Copenhagen Vector Bytecode (cphVB[42] see Part II section 6.2). The work on cphVB led to the publication and presentation[4] of a paper at the SciPy conference in Austin, Texas, June 2012. The contributions of cphVB are materializing the idea of using an intermediate representation (vector bytecode) as the mediator between a language and backend. As well as exploring the use of the virtual machine approach for processing array operations and observing performance related challenges for future work.

My contribution to cphVB was the implementation of the virtual machine using static dispatch for processing array operations on CPUs. The static dispatch processed a bytecode at a time, either performing a memory management operation or executing an array operation.

The first prototype matched and outperformed the performance of NumPy, however, the question was whether the backend would be able to match and outperform hand-coded C/C++ and the initial work revealed several challenges to meeting this goal.

When presenting and attending the conference I also became aware of a problem referred to as the Python *import problem*. The problem is an instance of the more general problem, that had been studied before[30, 26], of using dynamic loading when running on clusters with a shared filesystem. This problem was studied further and a novel approach to solving it was introduced and published with the paper *Bypassing the Conventional Software Stack Using Adaptable Runtime Systems*[46] (see Part II section 6.4) submitted to and presented at the Euro-Par 2014 parallel processing workshops.

```
print sum((rand(N)/100)+5)
```

Figure 1.4: Array-notation pseudo-code for computing the N random numbers, divide them with 100, add 5 and calculate the sum.

```
t1 = malloc(N);
for(i=0; i<N; N++)
  t1[i] = random_generator();

t2 = malloc(N);
for(i=0; i<N; N++)
  t2[i] = t1[i] / 100;
free(t1);

t3 = malloc(N);
for(i=0; i<N; N++)
  t3[i] = t2[i] + 5;
free(t2);

sum = 0;
for(i=0; i<N; N++)
  sum += t[3];
free(t3);
```

Figure 1.5: C-like pseudo-code processing array operations from figure 1.4.

```
...
output = malloc(N);
execute_array_operation(output, input);
free(input);
...
```

Figure 1.6: C-like pseudo-code illustrating an array processing pattern.

[4]The presentation was recorded and can be seen at https://youtu.be/HFxn3mSp9ww

A key observation was made during the exploration of approaches to processing array operations. Figure 1.5 illustrates the low-level operations performed when processing the array program in figure 1.4. In the case of Python/NumPy the loops in figure 1.5 are ufuncs[5] or calls to optimized libraries[1, 11, 2, 74, 43] when applicable and available. For every array operation the pattern consists of: `[allocate]; execute; [deallocate]`

For compound expressions arrays are allocated to store intermediate values between array operations as illustrated in figure 1.6. This pattern is common in languages that rely on library-delegation for efficient implementation of array operations.

There are multiple performance challenges with this approach which became the main focus of the studies to improve. The first step I took was experimenting with memory allocation, specifically a buffer allocation and reuse scheme dubbed the *software victim cache*. The work was done on a fork[6] of the NumPy project and the approach and findings described in the paper *Doubling the Performance of Python/NumPy with less than 100 SLOC*[47] (see Part II section 6.3) submitted to and presented at the PyHPC workshop in conjunction with SC13. The approach improved performance as side-effects of memory-allocation and first-touch page allocation were avoided which due to the `[allocate]; execute; [deallocate]` pattern on large arrays had a considerable improvement in terms of consumed wall-clock time. The work was a contribution to NumPy and generalizes to other languages to the extent that they apply the same array-processing pattern.

Python/NumPy served as a means to bootstrap the studies. However, the focus of the thesis is not on a high performance backend to Python/NumPy but rather a backend to array-oriented programming of which Python/NumPy is one example. Work was put into expanding language and hardware support resulting in the paper *Bohrium: a Virtual Machine Approach to Portable Parallelism*[41] (see Part II section 6.5) submitted to and presented at HIPS workshop in conjunction with IPDPS14. In addition to the name change the entire backend was a complete re-implementation with language support for Python, C, C++, CIL-based languages (C#, F#, etc.) and hardware support for CPUs, GPUs, and distributed memory. The step from cphVB to Bohrium was a large collaborative effort in which my contribution was the C++ language bridge, and array processing on CPU using the *victim cache* scheme.

On the topic of other languages then the C++ and NumCIL integrations were only demonstrated and their inner workings not described in the paper. Section 4.3 desribes the C++ integration in greater detail and lay out plans for future work. The paper *NumCIL and Bohrium: High productivity and high performance*[65] (see Part II section 6.8) submitted to and presented at the PPAM15 workshop on Language-Based Parallel Programming Models describes the CIL-based integration. I contributed to the Bohrium side of the language integration which involved modification of the C and C++ interface of Bohrium which the CIL-interface utilize.

```
t1 = malloc(N);
t2 = malloc(N);
t3 = malloc(N);
sum = 0;
for(i=0; i<N; N++) {
  t1[i] = random_generator();
  t2[i] = t1[i] / 100;
  t3[i] = t2[i] + 5;
  sum += t[3];
}
free(t1);
free(t2);
free(t2);
```

Figure 1.7: C-like pseudo-code processing array operations from figure 1.4 using composition.

```
sum = 0;
for(i=0; i<N; N++) {
  t1 = random_generator();
  t2 = t1 / 100;
  t3 = t2 + 5;
  sum += t3;
}
```

Figure 1.8: C-like pseudo-code processing array operationsarray operations from figure 1.4 using composition and contraction.

Returning to the topic of improving the array processing pattern as illustrated in figures 1.4, 1.5 and 1.6. Figure 1.7 illustrates the goal, which is to compose array operations, the motivation for array operation composition is the

[5]http://docs.scipy.org/doc/numpy/reference/ufuncs.html
[6]`victim_cache` branch of: https://github.com/cphhpc/numpy/

ability to apply array contraction as illustrated in figure 1.8. Achieving this goal significantly reduces the need for memory allocation and in some cases, such as the example in figure 1.4, allow streaming computations, reducing memory to a constant size. Thereby transforming othervise memory-bound array-expressions into compute-bound, removing pressure from the largest bottleneck in a compute system. Composition and contraction are thus essential for performance and required to reach an efficiency level matching that of hand-coded implementations in a low-level language.

However, performing composition and contraction are out of reach for the static dispatch approach. The static approach has to have an "answer" to every array operation in the form of a function. This is required since the backend must support high-level languages which includes interpreted languages inferring that the program is not known until runtime. This was implemented using C++ templates and instantiation of a massive library of about 900 functions. Having a function for every legal composition of array operations is infeasible.

The idea was to dynamically compile functions instead, the first iteration of work led to the paper *Just-In-Time Compilation of NumPy Vector Operations*[44], see Part II section 6.6 which was submitted to the GSTF Journal on Computing. The contribution of the paper was exploration of JIT-machinery and demonstrating feasibility of the approach by JIT-compiling a subset of bytecodes. My contribution to the work was working with Johannes Lund on the implementation. An essential observation was made namely that the internal program representation in Bohrium was insufficient for performing composition and contraction. This area became an essential area of research for every contributor to Bohrium. The description of work in this direction and challenges are described in greater detail in the not yet submitted paper *Fusion of Array Operations at Runtime*, see Part II section 6.12. My contribution to this paper primarily involve input to cost-functions and experimental validation of the theoretical models driving the construction of array operation composition and contraction.

The work on Bohrium's internal representation forked to explore different approaches, the first union of these efforts was realized with the paper *Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster*[40], see Part II section 6.7, which was submitted to and presented at PyHPC in conjunction with SC13. My contribution to the work was replacing the static dispatching for array processing on the CPU. Using lessons learned from previous work on JIT-compilation I reimplemented the CPU engine using the new program representation. However, numbers on parallelization were not provided in this paper and the implemented JIT-machinery worked in a single-instruction mode which generated, specialized and compiled code at runtime but did not apply composition and contraction.

As I continued work on these areas the first stable implementation and realization of array composition and contraction was contributed to and published with the paper *Prototyping for Exascale*[72], see Part II section 6.10, submitted to and presented at the Exascale Applications and Software Conference (EASC15). The paper describe the role of Bohrium as a prototyping tool for scientific computation, my contribution on JIT-compilation and code generation demonstrated scalable performance on multi-core architectures.

The latest advances in the JIT-machinary, codegeneration, and runtime instrumentation evolved into the C-targeting Array Processing Engine (CAPE) which was mentioned in the beginning of this section. The paper *Automatic mapping of array operations to specific architectures* (see Part II section 6.1), describes CAPE in greater detail, and provides a performance study showing the realization of the goal of bridging performance and productivity via the array-programming model.

My studies diverged as I got the opportunity of visiting and working with Bradford L. Chamberlain and the Chapel Team at Cray Inc. Chapel is an ideal candidate as a backend language as it provides the features sought after, that is, support for an array-oriented programming model and parallel execution. Since Chapel also provide a convenient array-notation it could potentially also be used in place of the domain specfic language sought after. However, from this perspective, then Chapel is lacking is an interactive environment such as for prototyping and visualization. The idea was therefore to explore the interoperability features of Chapel and experiment with using IPython/Python/NumPy providing the interactive environment and using Chapel under the hood for performance.

To facilitate this idea I contributed to the work described in the paper *Separating NumPy API from Implementation*[39] (see Part II section 6.9) which was submitted and presented at PyHPC in conjunction with SC14. The paper describes the Python module `npbackend` which factors out the work done in Bohrium of extracting the data-parallel operations of Python/NumPy into a self-contained component. The contribution of `npbackend` is that the backend implementing the NumPy array operations can be overloaded by invoking the Python program as `python -m npbackend program.py` the targeted backend can then be controlled via environment variable such as `NPBE_TARGET` with values such as `bohrium`, `pygpu`, or `numexpr`. That is, the NumPy backend implementation can be changed without modifying the program.

Work on interoperability between Python and Chapel materialized in the form of pyChapel[7][8], briefly described in the extended abstract titled *Scripting Language Performance Through Interoperability*[45], see Part II section 6.11. The extended abstract was submitted to, and presented[9] in more detail at HPSL in conjunction with PPoPP15. PyChapel consists of a foreign-function interface for calling Chapel procedures from Python and inlining Chapel code in Python. PyChapel also provide a Chapel module-compiler which compiles Chapel modules into Python modules. The work on PyChapel also encapsulate a minor contribution to the 1.10.0 release of the Chapel compiler which facilitated compilation of Chapel code into shared libraries. Additional work needs to be done to fully explore the potential of this Python and Chapel approach which is outlined in section 4.4.

[7]http://pychapel.readthedocs.org/
[8]https://github.com/chapel-lang/pychapel
[9]http://prezi.com/rzfzev1fzgul/

Chapter 2

Programming Model

In this chapter a description of an array-oriented programming model is introduced which encapsulate the high-level operations within linear algebra and generalizes them for use with tensors or specifically to multi-dimensional arrays. In the description of backends and libraries we saw the challenges of the low-level APIs and compiler-directives. We saw that the long-lived success of the BLAS-libraries as well as its shortcomings in terms of compositional optimization and how a BLAS-compiler and template-oriented libraries remedy these by providing even higher-level abstractions.

The problem with these approaches is that they break the strongest suit of BLAS, a stable interface. One cannot rely on a single interface for these operations but have to choose a specific binding such as Thrust to utilize multi-cores and NVIDIA GPUs or BOLT to utilize multi-cores and AMD GPUs. This breaks portability.

It is thus the purpose of the programming model to provide a stable high-level declarative interface which can be integrated into libraries and domain-specific languages as well as a single interface for a backend to implement. The goal is to hereby achieve productivity of the programmer, portability of the application and performance through the efficient implementation of the backend.

Section 2.1 describes what can be expressed within the programming model. The description provided here is not a formal language definition it is instead an abstract description of the model. Related work describing collection-oriented[64] languages provide examples and comparison of their support for collection-oriented programming. Languages compared include APL, SETL, CM-Lisp, Paralation Lisp, and Fortran 90.

A key difference between the collection-oriented model and the array-oriented model is concerned with data-representation. The collection-oriented model distinguishes between simple and nested collections. A simple collection is a list or vector of elements. A nested collection may contain collections as elements. These terms in the array-oriented model are one-dimensional and multi-dimensional arrays section 2.2 describes a key implementation concern for it.

2.1 Expressions

The following subsections describes the initialization of multi-dimensional arrays, the operations that can be performed on and with them.

2.1.1 Initialization

Initializers or generators define arrays. The generators themselves are not concerned with the dimensionality of the array they simply generate a flat array / vector of numbers with certain characteristics.

Zeros(n) generate n numbers with the value zero.

Ones(n) generate n numbers with the value one.

Value(n, k) generate n numbers with the value k where k is a scalar.

Linear(start, end, n) Generate n numbers with values increasing linearly between $start$ and end.

Uniform(n) generate n pseudo-random numbers with a uniform distribution.

Normal(n) generate a n pseudo-random numbers with a normal/Gaussian distribution.

Generate(n, func) generate n numbers using a stateful function.

Once generated the vector can be transformed into the shape needed.

2.1.2 Array Transformations

Reshaping is the basic transformations shaping data into a contextual useful form. Reshape is defined as $reshape(x, n_1, n_2, \ldots, n_m)$. Where x is an array, m is the rank or number of dimensions of the array, and $n_1, n_2, \ldots, n_m$ defines the length of the array in each dimension.

It is the intent with these primitives to not map directly to a domain such as linear algebra but instead provide building blocks on which a domain such as linear algebra can be built. An example would be how the instantiation of 3×3 matrix with uniformly distributed numbers could be built as:

```
randmat(m, n):
  x = uniform(m*n)
  return reshape(x, m, n)
```

In the example above m and n are integers denoting the $m \times n$ shape of the matrix x.

2.1.3 Operations and Operators

Element-wise, Scan, and Reduction are three fundamental operations which can be performed upon initialized arrays. A description of the operations are provided in the following subsections.

Element-wise

An element-wise operation applies an operator, such as those in figure 2.1, to the elements of one of more arrays. Operators are either unary or binary. The element-wise operations make up most of the common expression performed. An expression such as `sin((x * y)/2)`, where `x` and `y` are array variables, consists of three element-wise operations. Two using binary operators `*` and `/`. One using the unary operator `sin`.

Reductions

Reductions are operations such as computing the sum of all elements in an array A of length n: $\sum_{i=0}^{n} A_i$. However, reductions do apply to any binary associative operator such as those in table 2.1a. Reductions are often divided into *partial* and *complete/full* reductions. The summation as just described is a complete reduction. It takes a multi-dimensional array of any dimension and *reduces* it to a scalar by applying the binary operator to every element. A formal definition is provided below.

Definition. The reduction operation takes a binary operator $\bigoplus$ and an ordered set of n values $[a_1, a_2, \ldots, a_n]$ and returns the value $a_1 \bigoplus a_2 \bigoplus \cdots \bigoplus a_n$.

A *partial* reduction reduces the dimensionality of an array with n dimensions to an array with $n - k$ dimensions by applying the binary operator over an *axis* in the input array. Figure 2.1 illustrates three applications of a partial reduction with $k = 1$. A *complete* reduction is just a special-case of *partial* reduction where $k = n - 1$.

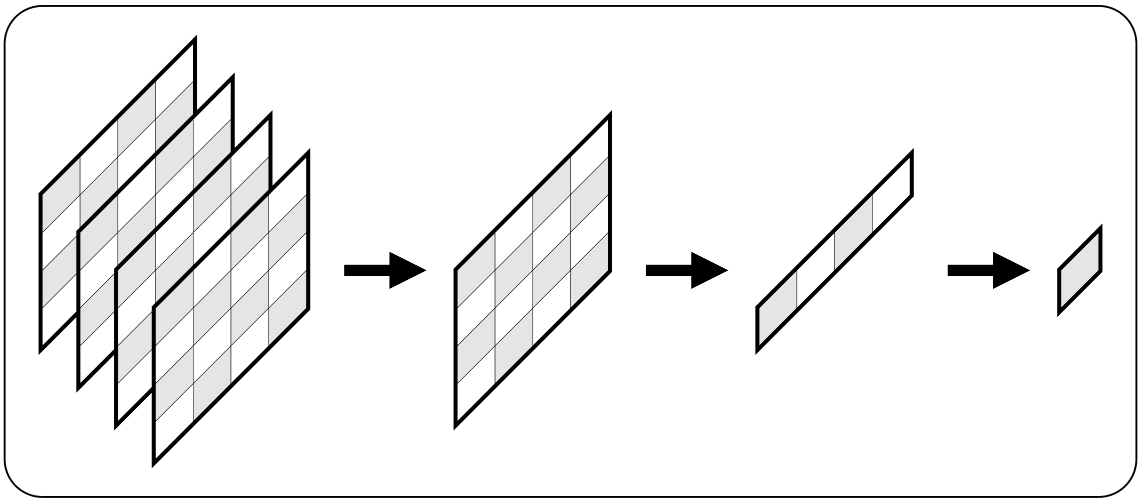


Figure 2.1: Three partial reductions with $k = 1$ on a $4 \times 4 \times 4$ array.

Reductions can in the programming model be described in this general form. However, the following operations should for convenience be available.

Numeric sum(x), product(x), mean(x), max(x), min(x).

Boolean all(x), any(x), count(x).

These should be available in both the complete forms and for partial applications. And provide a general construct for any binary associative operator or function: *reduce(A, k, axis, operator).*

Scan

The scan operation is similar to a reduction in the sense that it is stateful and accumulates a result. Unlike the reduction then the scan operation does not reduce the dimensionality. It stores the accumulations instead.

Definition. The **scan operation** takes a binary operator $\bigoplus$ and an ordered set of n values $[a_1, a_2, \ldots, a_n]$ and returns the ordered set of values $[a_1, a_1 \bigoplus a_2, \ldots, a_1 \bigoplus a_2 \bigoplus \cdots a_n]$.

Definition. The **exclusive scan operation** takes a binary operator $\bigoplus$ and an ordered set of n values $[a_1, a_2, \ldots, a_n]$ and returns the ordered set of values $[i, a_1, a_1 \bigoplus a_2, \ldots, a_1 \bigoplus a_2 \bigoplus \cdots a_{n-1}]$.

The scan operation is the general formulation, popular instances include prefix-sum or cumulative sum which is the scan operation applied with the addition operator. An application of prefix-sum to the input array $a = \{1, 2, 3, 4, 5, 6\}$ yields the result $\{1, 3, 6, 10, 15, 21\}$.

Gather/Scatter

The above-described operations apply operators in a pre-defined pattern on arrays. The gather and scatter operations, on the other hand, allow description of which array elements to read (gather) and which to write (scatter). The definition of which is controlled by an array of values. Also referred to as an index array.

Operators

Binary Associative $x \oplus y$	
Addition	$x + y$
Multiplication	$x \times y$
Minimum	$min(x, y)$
Maximum	$max(x, y)$
Logical And	$x \,\&\, y$
Logical Or	$x \mid y$
Bitwise And	$x \,\&\&\, y$
Bitwise Or	$x \mid\mid y$
Logical Exclusive Or	$x \wedge y$
Bitwise Exclusive Or	$xor(x, y)$

(a) Binary Associative.

Unary$\otimes x$	
Absolute value	$\lvert x \rvert$
Logical Not	$!x$
The inverse	x^{-1}
Exponential	$exp(x)$
Twos exponential	2^x
Base-2 Logarithm	$log_2(x)$
Base-10 Logarithm	$log_{10}(x)$
Square Root	$sqrt(x)$
Round up	$ceil(x)$
Remove decimal	$trunc(x)$
Round down	$floor(x)$

(b) Unary.

Binary $x \oplus y$	
Subtraction	$x - y$
Division	$\frac{x}{y}$
Power	x^y
Greater Than	$x > y$
Greater Than or Equal	$x \geq y$
Lesser Than	$x < y$
Lesser Than or Equal	$x \leq y$
Equal	$x = y$
Not Equal	$x \neq y$
Left Shift	$x \ll y$
Right Shift	$x \gg y$

(c) Binary Non-Associative.

Unary Geometry$\otimes x$	
Sinus	$sin(x)$
Cosinus	$cos(x)$
Tangens	$tan(x)$
Inverse Sinus	$asin(x)$
Inverse Cosinus	$acos(x)$
Inverse Tangens	$atan(x)$
Hyperbolic Sinus	$sinh(x)$
Hyperbolic Cosinus	$cosh(x)$
Hyperbolic Tangens	$tanh(x)$
Inverse Hyperbolic Sinus	$asinh(x)$
Inverse Hyperbolic Cosinus	$acosh(x)$
Inverse Hyperbolic Tangens	$atanh(x)$

(d) Unary Geometric Operations.

Table 2.1: Commonly supported binary and unary operators.

Array Broadcasting and Scalar Flooding

The binary element-wise operations above are only defined for arrays with an equal number of elements and dimensions. However, there are situations where arrays can be *broadcast* to allow the operation. The simplest instance is an operation with one array being a scalar. In this case, the scalar is replicated to fit the shape of the other array. This simple scheme generalizes as long as an array of lower dimension can be replicated to match the shape of an array of higher dimension. Such as an element-wise operation $X \otimes Y$ where X is a matrix of shape 3×3 and Y is a vector with 3 elements. In this case, the vector is replicated to create the matrix $Y^{'}$ of shape 3×3 in which each row of $Y^{'}$ is a replica of the vector Y. A similar operation would be possible for the Y^T in which case $Y^{'}$ would become a matrix of shape 3×3 in which each column is a replica of the vector Y^T.

Element-Access

The programming model focuses on extracting data-parallelism by providing a convenient high-level view of multi-dimensional arrays and performing operations upon them. The programming model thus highly **discourages** the use of imperative programming idioms such as:

```
for i in range(0, length(a)):
  a[i] = random()
```

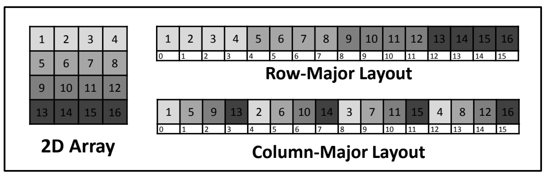
Such intent from the programmer should instead be expressed as:

```
a = random(number_of_elements)
```

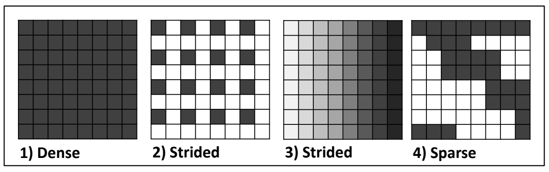
However, there are situtations in which the entire array is not of interest. Frequent patterns of this sort are the use of every nth element, every element but the first, every element but the last, elements from n to element m. The programming model supports such element access through the use of slicing. The slicing operation is defined as: $slice(i, j, s)$. Slicing selects a subset of the elements in a single dimension starting from element i, ending with element j and including every sth element. Slicing multiple dimensions can be used for blocking up matrices, and formulating stencil expressions.

2.2 Data Representation

A central implementation consideration for a backend supporting an array-oriented programming model as described in section 2 is the representation of multi-dimensional arrays. Array representation is concerned with describing the dimensionality, shape and how the array is laid out in memory. Figure 2.2b illustrates four cases which motivate for different layout schemes.



(a) Dense memory layouts of a two-dimensional array.



(b) multi-dimensional array patterns.

Dense representation is well-known as it is the default layout used by languages such as C and Fortran. As figure 2.2a illustrates then elements of a dense array is laid out as a continous string of bytes ordered by rows in C and by columns in Fortran. Dense layout is the simplest as the only meta-data needed is a flag defining element ordering. The main characteristic for a dense array in contrast to other layouts is that memory for every element is always allocated. The motivation for changing layout is concerned with lowering memory consumption often at the cost of complexity.

Strided representation is a generalization of the dense layout. It allows for skipping and re-using addresses in the memory space. This is done by associating a *stride* to every dimension which defines how to compute the address of the next element within that dimension. The row-major layout from 2.2a is represented by the strides 4×1 and the column-major layout by the stride 1×4.
Strided representation is useful for describing subsets of dense matrices such as the second matrix in figure 2.2a which is a subset of first the matrix. which uses the data from the first matrix using the stride 16×2. Other uses include compressed representation of matrixes which has a regular pattern in the coefficient values. Representing a 8×8 matrix where all elements has the value k can by achived with a stride of: 0×0. Using 0 as the stride for a dimension is the general technique for re-using memory. The third 8×8 matrix in figure 2.2b has identical values in all rows. This can be represented with a stride of 0×1.

Sparse representation allows for a compressed representation of a matrix. If the majority of the matrix elements is equal to the value k. Then a sparse representation will only allocate memory for the elements which was a value different from k. There exists a wealth of sparse layouts with different characteristics and space-time trade-offs, the simplest are compressed-sparse-row and compressed-sparse-column.

In the description of these different multi-dimensional array representations examples where only given for arrays with two dimensions. The representations do however generalize to arrays with any number of dimensions. A concern in this regard is that of practical importance since a three-dimensional matrix of shape $1024 \times 1024 \times 1024$ containing elements of single-precision point consume $4GB$ of memory. I have so far only worked with dense and strided array representations.

This ends the introduction of the array-oriented programming model, the following section describes how the backend implements support for handling it.

Chapter 3

Bohrium

Bohrium is a high performance backend for array-oriented programming. It provides the mechanics to couple a programming language or library with an architecture-specific implementation seamlessly. These mechanics cover two conceptually different areas: programming language *bridges* (subsection 3.3) and runtime *components* (subsection 3.2). Areas bound together by the program representation *vector bytecode* (subsection 3.1).

Described top-down, the language *bridge* maps array-operations to *vector bytecode*. Once mapped the Bohrium runtime *components* takes responsibility for transforming, scheduling and executing the intermediate representation. These responsibilities are handled by different classes of *components*:

Filters perform analysis and transformation of bytecode, such as organizing sequences of bytecode into blocks for the purpose of composition and contraction.

Vector Engine Managers (VEM) schedule the execution of vector bytecode on a vector engine or delegate scheduling to another manager.

Vector Engines (VE) translate vector bytecode to native code for a specific piece of hardware and execute it.

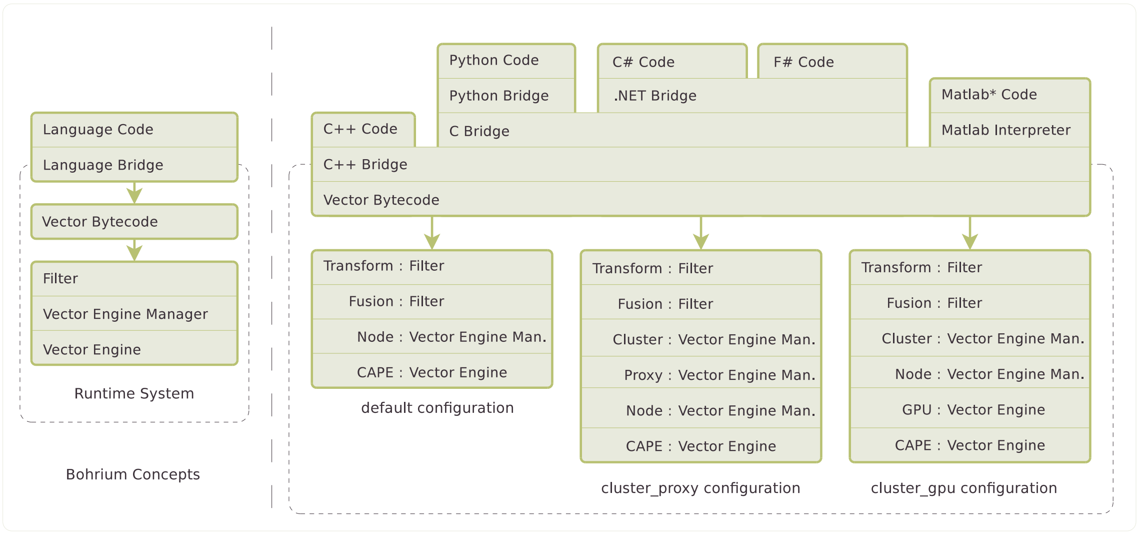


Figure 3.1: Overview of Bohrium concepts.

Figure 3.1 illustrates the Bohrium concepts and how components in the runtime can be combined to match a computing system. The core support library provides routines for manipulating bytecode, data-structures, and instrumenting the runtime components.

3.1 Bytecode

Bytecode forms the basis for the intermediate representation in Bohrium. It is inspired by assembly and represented as an ordered list of instructions. The ordering of instructions in the list guarantees that if an instruction uses an operand as input then a previous instruction has used the instruction as output. The intermediate representation in Bohrium thus simply consists of lists of bytecode instructions generated by the language bridge and sent to the runtime system for transformation, scheduling and execution. The ordering of instructions defines the ground for dependency analysis e.g. if two consecutive instructions write to two different operands then the two instructions can be scheduled for execution out of the order of the instruction list. They could also be scheduled for execution in parallel by a vector engine targeting a multi-core processor.

There is a substantial amount of transformation and scheduling decisions which are based on instruction dependencies. The primary motivations are *composition* and *contraction*. The vector engines in Bohrium have used the instruction list as the basis for analysis, and each vector engine has applied its own methods representing and analysis dependencies. The lessons learned, see Part 6.12, has merged into a unified representation and shared framework for analysis.

The bytecode is conceptually equivalent to a RISC-type instruction set with all instructions as three-address code (TAC) performing only memory, arithmetic, and logical operations. The vector bytecode has no conditional operators. This comparison higlights that the bytecode instruction-set is small and that instructions can have at most three operands, and notably that vector bytecode is not Turing complete. With this description in mind a vector bytecode can be textually represented as `ADD x,x,1`, where `ADD` is the Opcode and `x,x,1` are the operands. Sending this bytecode to Bohrium will eventually result in a vector engine executing it and computing: add 1 to x and store it in x. For this particular instruction, other ISAs has a specialized Opcode such as Increment. In Bohrium, it is up to the vector engine to do a value-specific optimization and determine if more efficient execution is possible.

At this point, the equivalence ends as instruction operands in bytecode are not registers or immediates but multi-dimensional arrays. Also, the instruction encodes the operands within the instruction. There are no operations for loading operands into memory which is often the case for RISC-type instruction sets. The principle of a RISC-instruction set is that there are few and simple instructions this is valid for Bohrium bytecode when viewed in the context of operations on multi-dimensional arrays. However, when translated to x86 native code the execution of a single bytecode instruction folds out into hundreds of thousands of instructions. From the perspective of native code, bytecode is extremely complex but from the point of view of array operations it is indeed compact and reduced.

Another trait of vector bytecode is that the type of operands is encoded within the operand and is not part of the Opcode. A humanly readable and trivially parsed JSON[21]-representation defines the bytecode and the allowed type and amount of operands. The bytecode encapsulate the following array operations:

Generate bytecodes are the source of structured data creation. These include random, range, and scalar flooding. The Random123[58] library is used to ensure consistent random number generation regardless of the parallel architecture in use.

Element-wise bytecodes apply a unary or binary operator to all array elements. Bohrium currently supports 53 element-wise operators, e.g. addition, multiplication, square root, logical and, bitwise and, equal and less than. For element-wise operations, Bohrium only allows data overlap between the input and the output arrays if the access pattern is the same. Combined with the fact that all operators are stateless, makes it straightforward to execute element-wise operations in parallel.

Reduction bytecodes reduce the dimensionality of an input array using a binary operator. Again, Bohrium does not allow data overlap between the input and the output arrays and the operator must be associative[1]. Bohrium currently supports 10 operators, e.g. addition, multiplication and minimum. Even though none of them are stateless, the reductions are all straightforward to execute in parallel because of the non-overlap and associative properties.

Scan bytecodes accumulate an input dimension using a binary operation. Again, Bohrium does not allow data overlap between the input and the output arrays and the operator must be associative. Bohrium currently supports 10 operators for scan bytecodes, e.g. addition, multiplication and minimum.

Gather bytecodes perform indexed reads. That is, given an input, output, and index array then the values of the index array is used to decide which elements of the input array is written to the output.

Scatter bytecodes perform indexed writes. That is, given an input, output, and index array then the values of the index array is used to decide which elements of the output array the input written is written to.

Data Management bytecodes determine the data ownership of arrays, and consists of three different bytecodes. The `SYNC` bytecode instructs a child component to place the array data in the address space of its parent component. The `FREE` bytecode instructs a child component to deallocate the data of a given array in the global address space. Finally, the `DISCARD` operator instructs a child component to deallocate any meta-data associated with a given array, and signals that any local copy of the data is now invalid. These three bytecodes enable lazy allocation where the actual array data allocation is delayed until it is used. Arrays are often created with a generator (e.g. random, scalar flooding) or with no data (e.g. intermediate), which may exist on the computing device exclusively. Thus, lazy allocation may save several memory allocations and copies.

Extension methods The bytecode classes mentioned above make up the bulk of a Bohrium execution. However not all algorithms may be efficiently implemented in this way. In order to handle operations that would otherwise be inefficient or even impossible, we introduce the fourth type of bytecode: `extension methods`. We impose no restrictions to this generic operation; the extension writer has total freedom. However, Bohrium does not guarantee that all components support the operation. Initially, the user registers the extension method with paths to all component-specific implementations of the operation. The user then receives a new handle for this `extension method` and may use it subsequently as a vector bytecode. Matrix multiplication and fast fourier transformation are examples of extension methods that are obviously needed. For matrix multiplication, a CPU specific implementation could simply call a native BLAS library and a Cluster specific implementation could call the ScaLAPACK[11] library.

This concludes the representation of array operations in Bohrium, the following section describe the runtime components for managing transformation, scheduling, and execution of the operations.

3.2 Runtime components

In the context of array operations, the bytecode is compact. A reduction Opcode provides bytecode for doing reductions as described in section 2.1.3, but only in its partial form. This means that

[1]

Mathematically associativity; we allow non-associativity because of floating point approximations in order for a language bridge to provide the functionality of a full reduction it has to send n reductions where n is the dimensionality of the array. It is one of the responsibilities of filters to analyse and transform the intermediate respresentation and annotate such sequences of bytecode into a form which can yield a better execution strategy for the vector engine. Filters can also do basic transformations such as `POW x,x,2` which on many hardware devices executes more efficiently as `MUL x,x,x`. More interesting transformations involve detecting grounds for doing loop-fusion or streaming.

3.2.1 Filters

The classification of components is meant to organize responsibilities but not nescesarily restrict itself. Filters in this sense serve as a wildcard for components which do not fit into the other classes. Bohrium currently consists of the following filters.

fuser encapsulates the dependency analysis for array operation composition and contraction. Multiple instances such as `topological`, `greedy`, `singleton`, and `optimal` of the fuser exists applying different algorithms for performing transformation of bytecode sequences. The work on fusers are described in greater detail in the paper *Fusion of Array Operations at runtime*, see Part II 6.12.

bccon contracts sequences of bytecode. The purpose of the component is to detect bytecode and sequences thereof for which more efficient but equivalent expressions exist. The simplest example would detect a bytecode such as `ADD x,x,0` and replace it with `NOP` or copy if the output is to a different operand than the input. The detection of sequences can for instance find the expression of a matrix multiplication and map it to an extension bytecode thereby delegating execution to a highly tuned library instead of the general operation composition.

bcexp expands a single bytecode to a sequence. Given the example above then it might seem detriment to performance. However, one example of its application is to detect one-dimensional reductions and transform them to two-dimensional reductions which execute more efficiently on the GPU.

pprint is a debug filter, it dumps an ASCII representation of the bytecode to the filesystem. It can conveniently be used in the runtime-stack before and after another to manually inspect transformation.

fuseprinter is a debug filter, it dumps a dot representation of the bytecode to the filesystem, visualizing the dependencies of the bytecode.

Another application of filters is the transformation of various corner cases, with regards to array-representation, into a general case thereby simplifying processing at a later stage such as code generation.

3.2.2 Vector Engine Managers

Vector engine managers are responsible for one memory address space in the hardware configuration. Bohrium currently have the following three vector engine managers:

node is very simple since the hardware already provides a shared memory address space; hence, the Node-VEM can simply forward all instruction from its parent to its child components.

cluster handles the global distributed address space of a compute cluster. It is quite complex and its implementation is based on previous work[38].

proxy is the component I implemented to support the exploration of *Bypassing the Conventional Software Stack Using Adaptable Runtime Systems*[46], see Part II section 6.4.

3.2.3 Vector Engines

The primary responsibility of a vector engine is to execute the instructions it receives in an order that comply with the dependencies between instructions. Furthermore, it has to ensure that its parent component has access to the results as governed by the data management bytecodes: *sync*, *free*, *discard*. Bohrium has these two vector engines:

gpu is an OpenCL based implementation focusing on array processing for GPUs. It is the based on the efforts of Troels Blum[15, 16].

CAPE is the C-targeting Array-Processing Engine and is main tool for the thesis, see Part II section 6.1. The primary focus of CAPE is shared memory multi-core architectures with non-uniform memory access. Additionally, ongoing work is showing promising results in encapsulating the concerns of accelerators which is described further in section 4.2.

3.2.4 Stack configuration

The runtime components can be combined to fit a given computing environment whether that is a laptop, a workstation, or a compute cluster. Configuration is done via a configuration file in an INI file format as shown in figure 3.2.

Stacks and components are identified by sections and the entry `type`, defines just that the type of the section. The type of a section can be any one of: `stack`, `filter`, `ve`, and `vem`. For stack-sections, the entries define the runtime configuration by chaining component identifiers. The stack_default in figure 3.2 is equivalent to the stack configuration visualized in figure 3.1 from the beginning of the chapter.

The component-sections define instanses of components. A component can have multiple sections with different options. E.g. the `bcexp` component has instances identified as `bcexp`, `bcexp_cpu`, and `bcexp_gpu`. These instances perform different transformations as some are useful for one architecture but not on another. As a user of Bohrum one can either choose to modify the configuration to match their system or switch runtime stack via the environment variable `BH_STACK`. Re-targeting a high-level implementation is thus as simple as switching an environment variable. The application itself does not need any modification.

```
[stack_default]
type = stack
stack_default = bcexp_cape
bcexp_cape = bccon
bccon = topological
topological = node
node = cape

...
[cape]
type = ve
impl = libbh_ve_cape.so
libs = libbh_visualizer.so,libbh_fftw.so
timing = false
bind = 1
vcache_size = 10
preload = true
jit_level = 3
jit_dumpsrc = true
jit_offload = 0
compiler_cmd="..."
compiler_inc="..."
compiler_ext="..."
object_path="..."
template_path="..."
kernel_path="..."
```

Figure 3.2: Default stack configuration for a desktop computing environment.

Components are also controllable via environment variables, this is done via a naming convention: `BH_SECTION_OPTION`. E.g. modifying the default thread-binding policy of the CAPE engine can be done via `BH_CAPE_BIND`, the size of the software victim cache via `BH_CAPE_VCACHE_-SIZE`, optimization-level of the JIT-compiler via `BH_CAPE_JIT_LEVEL` and controlling use of accelerator offloading via `BH_CAPE_OFFLOAD`.

3.3 Language Bridges

It is the task of the language bridge to map high-level abstractions within the language to bytecode. The technology used to do so is up to the implementer of the bridge. The bridge can be built at multiple levels, as a library, part of an interpreter or an extension of a compiler. Bohrium currently support several languages as visualized in figure 3.1 at the beginning of the chapter. A brief overview of the languages and their construction is provided below.

Python uses a library-based approach. The NumPy library for Python is a building block for representing and manipulating multi-dimensional arrays. SciPy and IPython are examples of such libraries. The combination of these libraries offers a rich environment for scientific computation. By providing a NumPy compatible library, a vast amount of existing applications can benefit from the Bohrium Runtime system without modifying the Python interpreter or changing a single line of Python code.

C++ implements a domain specific embedded language which is described in greater detail in Chapter 4.3. It is also a fundamental building block for language interoperability via the C bridge.

C implements an encapsulation of vector bytecode as functions. It provides a set of functions on the form: `BH_BYTECODE_STRUCTURE_TYPES`. Such as `bh_ewise_add_aaa_fff(...)` for creating a bytecode representing the element-wise addition of three arrays of floating points numbers. It is not intended for end-users but rather as an interoperability bridge between Bohrium and languages supporting C. The bridge is implemented as a thin wrapper on top of the C++ bridge. The C++ bridge performs the responsibilities of a language bridge for Bohrium, the C bridge then simply exposes a C-compatible interface for it.

CIL implements a bridge with the common-intermediate language CIL and via NumCIL library and facilitates use of Bohrium for languages including C# and F#.

MiniMatlab is a current work in progress which implements an interpreter for a subset of the Matlab language. The interpreter relies entirely on the Bohrium runtime to process array-operations. A project for future work will offer a language bridge for GDL[4], the OpenSource version of the defacto language used for data-analysis in Astronomy, IDL. Extending the GDL compiler is in this case a good choice for integrating Bohrium as arrays are first class citizens in the GDL language.

Choosing a strategy for integrating a language with the Bohrium runtime system is thus highly dependent on the language itself and its users. The following subsection describes the technical details on interfacing with the runtime system followed by a section describing the binary representation of vector bytecode along with the intermediate representation.

3.3.1 Responsibilties

Everything in the runtime system is a component which means the interface is the same regardless of whether the component transforms, schedules or executes bytecode. Table 3.1 shows the four functions that make up the interface. Components are initialized recursively, the bridge only has to initialize itself and its most immediate child in the stack. Listing 3.2 provides a minimal code example, without error-checking, of using the core library for initializing the runtime system.

The main task for a language bridge is to generate valid bytecode and send it to the runtime system via the C-interface. It is therefore the main concern for the language bridge to map language constructs to bytecode. The general idea is that if a certain task is beneficial in multiple languages then the runtime system should perform the task.

Listing 3.1: Component and Runtime Interface.

```
typedef bh_error (*bh_init)(const char *name);
typedef bh_error (*bh_shutdown)(void)
typedef bh_error (*bh_execute)(bh_ir* bhir);
typedef bh_error (*bh_extmethod)(const char *name,
                                 bh_opcode opcode);
```

Listing 3.2: Runtime Initialization and Use.

```
int64_t        component_count;
bh_component **components;
bh_component  *bridge,
              *runtime;

// Setup the bridge
bridge = bh_component_setup(NULL);
bh_component_children(bridge,
                      &component_count,
                      &components);

// Recursively initialize children
runtime = components[0];// Init runtime
runtime->init(runtime);

...
// Program is running
...                     // Create bytecode
runtime->execute(...);  // Send bytecode
...

// Program end
runtime->shutdown();    // Terminate
```

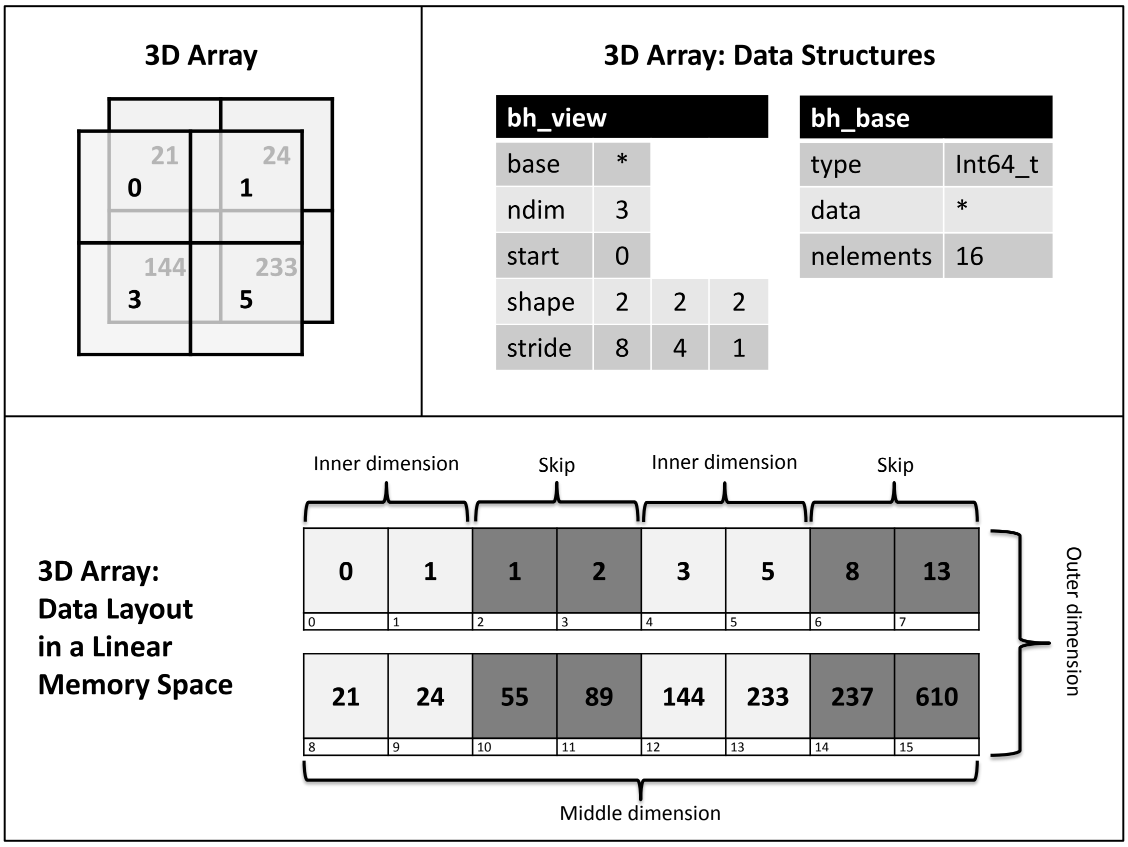
A simple example is the translation of an element-wise array-operation such as `x^2` into bytecode. In many languages translating this into `MUL t,x,x` instead of `POW t,x,2` is straightforward. However doing the transformation at the language-level requires that every language bridge must implement it. A simpler solution would be to implement the transformation within a filter-component and thus implement and maintain it in only one place. More importantly it facilitates the application of a transformation closer to the vector engine which can decide whether a given transformation is beneficial for execution on the hardware in question. That is, the stack configuration could include a transforming filter on top of one vector-engine component but not on top of another. Also, if such a transformation is unwanted by the user then it could simply be switched on/off in the runtime configuration.

This design choice in Bohrium provides for the implementation of simple language-bridges which can naïvely translate into bytecode. It is however not meant to discourage optimization at the language level but instead consider the type of optimization and transformations performed. For instance, utilizing the type checker of a compiled language as a means of removing validation from the runtime. However, it is more interesting and perhaps more fruitful to consider language based optimizations in the context of the high-level abstractions that the language offer its user before bytecode translation. The high-level abstractions provide information which is lost when translated into bytecode. An example would be a domain specific language for linear algebra, in this context utilizing identity, symmetry, and orthogonality provides for high-level transformation for evaluating whether a matrix is diagonalizable. Given the expression $AIIIIB$ where A and B are regular matrices and I is an elementary matrix, by using domain-specific knowledge within the language such an expression can be transformed to the single matrix product AB. Bohrium encourages such high-level transformation at the language-level and then sending the last expression to the runtime for execution.

When generating bytecode the language bridge must ensure the generation of valid bytecode. A valid instruction consists of an Opcode with the amount and type of operands as defined in the bytecode definition. Additionally, the shape of the operands must be the same or in the case of partial reductions have an output operand with one dimension less. The runtime system cannot execute a bytecode such as `ADD t,x,y` if t, x, and y does not have the same shape. It is the responsibility of the bridge to transform the shapes as a means of allowing the execution. Likewise transformation of operand meta-data is in general the responsibility of the language bridge.

The *bh_view* data-structure encapsulate the data-descriptor for arrays in Bohrium. Figure 3.3 illustrates how a three-dimensional operand shaped as $2 \times 2 \times 2$ is represented in Bohrium along with the layout of the operand in a linear memory space. The main purpose of the *bh_-view* is representing a multi-dimensional array using a strided indexing scheme. The attributes *bh_view.ndim* and *bh_view.shape* are self-explanatory. The *bh_view.base* attribute points to a *bh_base* data-structure which describes a physical memory allocation in a linear memory space.

The *bh_base.type* attribute dictates the primitive type the allocation should be used for, *bh_-base.data* points to the beginning of the allocation and *bh_base.nelements* states the amount of elements of the given primitive type there should be allocated memory for. The data-structure *bh_view* and *bh_base* are separate to distinguish between the memory allocation *bh_base* and what the allocation represents namely the *bh_view.* The allocation of the actual data belonging to an operand is often the responsibility of the vector engine as decisions such as to what, when and how to allocate memory on a processing unit are inherently tightly coupled with the hardware and should not be a concern for the language bridge.



bh_view			
base	*		
ndim	3		
start	0		
shape	2	2	2
stride	8	4	1

bh_base	
type	Int64_t
data	*
nelements	16

Figure 3.3: Representation and memory layout of a three-dimensional operand in Bohrium.

However, there are cases where the language bridge performs the allocation. This is the case when data originates from the language and not as output from the execution of bytecode. One example of this would be the reading an image-file from disk. In this case, the language takes care of loading the image into memory and assigning the data pointer to the *bh_base*.

The attributes *bh_view.start* and *bh_view.stride* make out the strided indexing scheme part of the multi-dimensional array representation. The listing below describes the general set of functionality that a language bridge must provide.

Aliases/Views to support these the language bridge must create a *bh_view* in which the *bh_view.base* points to the *bh_base* of another operand.

Slicing should be provided in collaboration with views. When the user wants to define a subset of values of an operand the bridge must provide a convenient syntax for doing so and then create a *bh_view* of the sliced operand and modify the shape and strides of the view to match the user-defined slice.

Broadcasting is another operation which is based on manipulation of views. The bridge must, if possible, broadcast the operand of an expression such as: $X + v$ where X is a matrix and v is a vector. This is done by instantiating a *bh_view* of v with strides replicated to match the shape of X.

Reshaping is done by the bridge by manipulating the *bh_view.ndim*, *bh_view.stride* and *bh_-view.shape* of an operand. This should be supported for arbitratry shapes also those changing the number of dimensions.

This ends the description of Bohrium, how it represents array-operations in the intermediate presentation, the data-descriptor for the arrays themselves, runtime components, and language bridges.

Chapter 4

Ongoing and Future Work

This chapter provides an overview of ongoing and future work. Section 4.1 describes the areas for further study of the array-oriented programming model and Bohrium's support for it. Section 4.2 details current efforts on support for coprocessors and accelerators in the C-targeting Array Processing Engine. Section 4.3 provides a lengthy description of the BXX DSEL, the C++ language bridge to Bohrium. Section 4.4 describes the potential areas of further exploration on interoperability between Python and Chapel. In the last section, section 4.5, is the Benchpress tool briefly introduced and future work outlined for facilitating comparative experiments.

4.1 Descriptive Power

Sections 2 and 3.1 describe the array-oriented programming model and Bohrium's representation of operations and support for the model. The model is applicable to a range of applications however, since it is not turing complete, it has limitations. Specifically, when the composition of generate, element-wise, reduction, scan, gather, and scatter operations are not sufficient to describe a given computation, what do you do?

Bohrium already has an answer to this question, which is, simply take your data out of Bohrium and implement the operation within the host-language or call a Bohrium extension method. However, there are operations which might be expected to be readily available and conveniently encapsulated, including:

Sort a fundamental operation used in a wealth of algorithms, it is quite essential to have sorting as a building block to obtain efficient performance.

Filter provides a way to *conditionally* select a subset of elements within an array and is another fundamental building block for the programming model.

Beyond expanding support for operations there is a need for expanding the operators. The most interesting challenge is to provide support for user-defined operators. That is, enable the user to define an operator/function for application with the generate, element-wise, reduction or other operations. An area deserving of future work is, therefore, to expand the expressive completeness or descriptive power of the model and Bohrium's capabilities for processing them.

4.2 Targeting accelerators with CAPE

The performance study showed the viability of the approach applied to the design and implementation of the C-targeting Array Processing engine CAPE for multi-core processors with NUMA-architecture. Ongoing work is exploring its applicability to accelerator architectures such as GPUs and Intel MIC. The two main areas to expand are: code generation and runtime instrumentation. The code generator must emit loop constructs with OpenACC directives for GPUs and parallel sections/loop constructs with LEO/OpenMP directives for MICs. Runtime instrumentation involves memory management on the accelerator device. Currently explored is a runtime extension using a simple `alloc`, `free`, `push`, and `pull` interface. The interface abstracts the details of LEO `#pragma offload_transfer` and OpenACC routines such as `acc_[create|delete|update_device|update_self]`. The extension manages relations between host pointers and device pointers, knows the state of buffers on host and device, and a scheme for data persistence is implemented.

4.3 BXX: Bohrium for C++

BXX, the Bohrium language bridge for C++, serves a dual purpose. It is the fundamental building block for other language bridges and a self-contained domain-specific embedded language (DSEL) for array programming in C++. As a building block, it serves a key role in Bohrium and as a DSEL it contributes to array programming in C++. BXX can be considered finished work although not yet published. Chapter 3 describes the role of BXX in Bohrium. This section briefly introduces the DSEL part of BXX and its contribution to array programming in C++ and what work needs to be done before it is ready for publication.

Work related to bridging the gaps between performance, productivity and portability in the context of C++ includes high-level abstractions such as vectors, matrices and tensors and a convenient array-notation for performing operations on and with them. Scalar-multiplication and matrix addition such as: $Z = 42 \cdot X + Y$ can for example be written as `Z = 42*X+Y;`. Libraries using this approach include Blitz++[71], Armadillo[60], and Eigen[35]. Users who come from the domain of mathematics find it familiar as it is identical to the mathematical notation except for the semi-colon terminating the expression. The array-notation leverages C++ language features for operator overload and generic programming. It is via these features possible to construct a domain-specific embedded language within C++. Expression templates (as defined by David Vandevoorde[70]) can at compile-time construct an expression tree of all operations performed on and with the library defined types. This DSEL/library design is quite clever in that the operator overload enables the convenient array-notation and the expression-tree provide a means to apply performance enhancing optimizations such as array-operation composition.

Expression templates do have some negative traits. Compiling an invalid C++ program with an error in the expression-template produces multiple pages consisting of the entire expression tree. Resulting in multiple screens of information that is hard to use to for debugging, even finding a syntax error is troublesome. Another issue is that expressions cannot trivially be passed as input to a function or returned from a function. The reason being that it would require specification of the expression-tree within the function signature which is extremely inconvenient. Eigen and Armadillo's solution is to provide a base class/type for all expressions. The most severe challenge for these libraries, even though they hide low-level details from the user, are that the implementations themselves are tightly bound and thereby not easily retargetable to different hardware architectures.

Purely library-based approaches based on expression templates have shortcomings in several areas. The Intel Cilk Plus provides C/C++ language extensions for array-notation that patches a great deal of these issues by providing a very convenient array notation, better error messages and exploits data-parallelism for performance. Language extensions such as Cilk Plus are advantageous. However, also quite invasive as it requires extending compiler support.

The BXX DSEL applies a similar approach to Blitz++, Armadillo, and Eigen for providing convenient array-notation, that is, operator overload and generic programming. However, instead of relying on expression templates for performance, BXX maps array operations to vector bytecode and thereby delegate scheduling and execution to the Bohrium runtime. The following subsection provides an overview of the look and feel of BXX.

Notation and Use

C++ is notorious for its support of multiple programming paradigms. It is thus possible for a DSEL in C++ to be object-oriented, function-oriented, or a mixture of both orientations. A functional approach integrates well with operator overloads as it very closely mimics the syntax of mathematical expressions such as $X = sin(Y) + Z$. An object-oriented style does also hold its merit for mathematical expressions *on* operands such as: X^t which can be represented as `X.transpose()`. I have opted for a functional paradigm as a means to keep the notation consistent and within a single paradigm. The following goes through the most common operations of the library and describes the notation by example.

```
// Declaration of variables
multi_array<float> x, y, z;
multi_array<int> q;
```

Declaring and defining array variables are separate operations. The declaration as shown above is only concerned with providing a name and type of an array. The initialization defines the shape along with the actual data. In the example below are two one-dimensional arrays (vectors) initialized with three ones and three pseudo-random numbers.

```
// Definition and initialization
x = ones<float>(3);    // y = [1.0, 1.0, 1.0]
y = random<float>(3);  // x = [0.225, 0.456, 0.965]
```

Variables can, once declared *and* defined, be used as *input* for operations such as element-wise addition or reduction.

Variables can, once declared but *not* defined, be used to *store* the result of an operation. They will then inherit the shape based on the result of the operation when assigned.

```
// Element-wise operations
z = x + y;  // z = [1.225, 1.456, 1.965]
// Reduction
z = sum(z); // z = [4.645]
```

Array variables reference either another array variable or the anonymoys result of an operation. Directly assigning a variable to another will create a view/alias of the other variable. Given two variables `x` and `y`, where `y` is an alias of `x`, any operation on `y` will also affect `x` and vice versa as illustrated in the example below.

```
// Aliasing
y = x;              // y is an alias of x
y += 1;
cout << x << endl;  // [1.225, 1.456, 1.965]
```

In case an actual copy of a variable is needed the user has to explicitly request a copy. Copies also occur implicitly when variables are type-cast. Both of these situations are illustrated below.

```
// Explicit copy elements of arrays
z = copy(x);       // z = [1.225, 1.456, 1.965]
// Typecasting, copies implicitly
q = as<int>(x);    // q = [1, 1, 2]
```

The definition/initialization assigns the shape of a variable. It can be changed at a later point in time as long as the number of elements remain the same. The code below provide a couple of shape transformation examples.

```
multi_array<float> u;
u = random<float>(9);
...
u = reshape(u, 3, 3); // Turn vector into a 3x3 matrix
u = transpose(u);     // Transpose the matrix
...
u = reshape(u, 9);    // Turn 3x3 matrix into a vector
```

We have so far covered how to describe alias and explicit copies. This leaves the notation for updating an array. The code below show how to update either a part of or the entire array.

```
y(x);                 // Update every element y with the values from x
y[_(0,-1, 2)] = 42; // Update every second element
```

The update of the every second element in the example above introduces the slicing notation. This notation is the most brittle from a productivity perspective compared to the notations provided by languages such as Matlab, R, Python and Cilk Plus. However, it is as good as it gets when using a library-based approach. A sliced array variable can occur anywhere a regular array variable can occur and there are a couple of shorthands that condenses the slicing notation.

```
multi_array<float> grid;
grid = random<float>(9,9);

// Sliced aliases
center = grid[_INNER][_INNER];

// Slicing shorthand
y[_ALL]    // All elements
y[_ABF]    // All but first
y[_ABL]    // All but last
y[_INNER]  // All but first and last
```

Further examples of the notation, as well as examples of applications such as Black-Scholes, Heat-Equation, Rosenbrock, Leibnitz PI, Monte Carlo Pi, and Shallow Water can be inspected on the benchpress website: benchpress.rtfd.org.

The DSEL supports basic functionality for legacy support with C++ in the form of the iterator-interface for element-wise traversal. Overload of the shift-operator provides a convenient means of outputting the contents of the array.

```
for(multi_array<float>::iterator it=y.begin(); it != y.end(); ++it) {
  printf("%d", *it);
}
...
cout << y << endl;
```

The use of the iterator is highly discouraged as it forces synchronization of memory with the C++ memory space. Each element needs to be exposed and printed to screen in the above example. The iterator forces memory, which could be distributed out on GPU device memory or distributed in a cluster, to be copied back into main-memory for the thread requesting the iterator.

The iterator should for this reason only be used at the end of an application when results from computations need to be reported back to the user of the application. The BXX library also provide `export/import` methods for retrieving and providing a pointer to the underlying storage used by the arrays. The `export/import` are intended for interoperability with visualization libraries and to integrate with legacy code.

Future Work

In the paper introducing CAPE ("Automatic mapping of array operations to specific architectures", see the Publications chapter section 6.1), a performance study using BXX is provided comparing speedup. Speedup of hand-coded implementations of a set of benchmarks in C++ using OpenMP. Results showed that BXX/CAPE can match and outperform these implementations. The extraction of the data-parallel array-operations via the array-notation provided by BXX and the delegation of optimization, scheduling and execution to Bohrium/CAPE is the driver of performance. The main contribution of BXX is the exploration of a novel approach to high performance DSEL implementation in C++. BXX provides convenient array-notation without the quirks of expression templates, without requiring compiler-support and performance matching hand-coded implementations in C++ with OpenMP.

Future work will study the performance further, figure 4.1 show a first step in this direction. The figure shows a graph of elapsed wall-clock as a function of threads for the Black Scholes benchmark implemented with Armadillo, Blitz++, OpenMP (C++/P), and BXX with (BXX+0) and without (BXX-O) optimization. As the figure shows, the expression-template approach of Armadillo and Blitz++ is able to perform array-contraction and thereby decent single-threaded performance. However, Blitz++ does not support parallelization and Armadillo relies on BLAS libraries for parallelization. Parallelization is thus only successful when operations are directly mappable to BLAS which fails for this benchmark. The focus for a future performance study is therefore Intel Cilk Plus which should be able to utilize Intel MKL in a similar manner but also perform parallelization and optimization of general array operations.

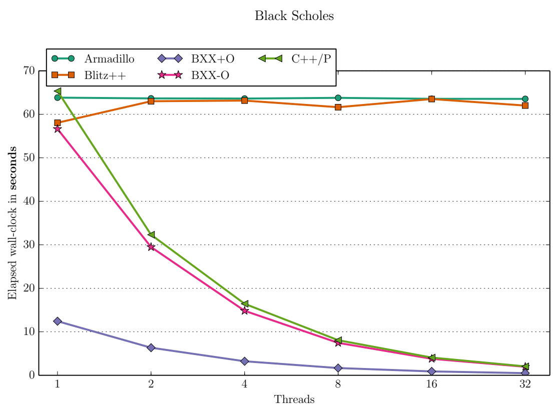


Figure 4.1: Results from the Black Scholes benchmark reported as elapsed wall-clock in seconds. Four different C++ implementations using Armadillo, Blitz++, Hand-coded parallelization with OpenMP, and BXX with and without optimization. Executed on machine with two Opteron 6274 processors and a total of 32 cores.

In addition to a performance study, usability features will be added such as interface-compatibility with existing libraries. Thereby enabling interoperability with Blitz++, Armadillo, and Eigen or porting implementations using these libraries. Also for usability, the set of functions for linear algebra will be expanded, and other convenient functions added.

4.4 PyChapel

The pyChapel module introduces interoperability between Python and Chapel. However, it can with little effort be expanded to support any language capable of interoperating with C. Including but not limited to Fortran and Haskell and thereby providing a generic and simplified approach to foreign function interfaces in Python.

The ideas I had for Python and Chapel interoperability were cut short as I prioritized exploration of the Bohrium/CAPE approach. However, interoperability between Python and Chapel is a powerful combination that allow a study of language symbiosis. From Pythons perspective, the interoperability allow access to a low-level language that supports the same abstraction level as NumPy. One project to pursue is finishing the work on pyChapel and the `npbackend` module. Thereby transparently mapping Python/NumPy array operations to Chapel. Effectively increasing the performance of the Python/NumPy program without changing a single line of code. When those abstractions fail, then Chapel is available, under the hood, for low-level instrumentation as a significantly more convenient alternative to parallel programming than C, C++, and Fortran.

I also spent time creating a *Chapel for Python Programmers*[1] guide, describing common Python idioms in Chapel terms. This work could be expanded to encapsulate further aspects and also to create a `pythonic` Chapel module. That is a module with functions, iterators, and constructs familiar to the Python programmer.

From the Chapel perspective, an interesting development in interactive environments has happened which would be interesting to explore. The interactive environment provided by IPython/IPython Notebook has evolved into a self-contained system named Jypyter. Jypyter supports 40 different kernels to drive computation. The potential here is that Chapel could leverage the features of the interactive environment by implementing a PyChapel kernel for Jypyter.

In summary, my work exploring Python and Chapel interoperability stopped because of time constrains not for lack of potential or interesting areas to explore.

4.5 Benchpress

Benchpress[2] is a collection of tools and benchmark implementations I wrote for experimental validation and performance testing. It provides the following functionality/commands.

bp-run is the main driver; it polls the system for hardware and software information, but most importantly it executes suites of benchmark implementations and stores results in a JSON file. It facilitates interaction with SLURM and various parameters including the amount of trials to run for each benchmark. It collects data via other tools such as perf, time and in addition to the benchmark emitted elapsed wall-clock time. Hooks, post/pre-execution commands, can be applied for cleaning up data and side-effects before and after a benchmark run.

bp-info provides information about the location of benchmarks, hooks, commands, and the module itself.

bp-times renders a `bp-run` result file in plain text formats such as ASCII and CSV.

bp-grapher renders a `bp-run` result file in different formats such as PNG, HTML, and pdf displaying benchmark results using either a generic renderer or one specialized to an experiment.

bp-compile as the suggests compiles benchmarks that require compilation. It is a minimal compilation framework ensuring that miscellaneous flags are equivalent for C and C++ implementations.

The tool itself is finished work and has aided experimental testing throughout the studies. What is left to be done is expanding the set of Benchmarks. Currently, a rich collection of Python and a good selection of C/C++ implementations exists. However, for future work these must be expanded to include hand-coded implementations to serve for comparative studies of CAPE and accelerators.

[1]http://chapel-for-python-programmers.readthedocs.org/
[2]http://benchpress.readthedocs.org/

Chapter 5

Conclusion

I set out to explore the thesis: It is possible to construct a language agnostic backend for high-level declarative languages without sacrificing performance. The approach for this thesis involved investigation of related work and the construction of a backend prototype. Related work on parallel programming languages for large-scale distributed memory architectures revealed data-parallelism as the driver for performance.

Parallel languages such as HPF, ZPL, and Chapel inspired the idea of exploring the construction of a backend that could exploit the inherent data-parallelism of array operations for performance. A backend that would aid productivity by supporting high-level declarative array-notation in use by languages such as Python, Matlab, and R. The backend would have the additional benefit of facilitating language integration via, vector bytecode, an intermediate representation encapsulating array operations and data management. The exploration thus turned into the construction and experimental evaluation of a backend that would:

- Support an array-oriented programming model
- Enable language integration via an intermediate representation
- Target a performance that is comparable to straight forward hand-coded C/C++ for the same application

The backend materialized through collaborative efforts in the form of Bohrium, a high performance backend for array-oriented programming. Bohrium features integration with Python, C, C++, C#, and F#. Integration with these languages are facilitated via NumPy for Python, via the BXX DSEL for C++, and via the NumCIL library for the CIL-based languages C# and F#. The C language integration serves as a language interoperability interface encapsulating the primitives for the vector bytecode intermediate representation.

The parallel languages inspiring this work focused on delivering performance for large-scale distributed memory. In contrast to those efforts, the performance context of the thesis are the challenges presented by the current and next-generation processing units consisting of heterogeneous architectures with complex memory hierarchies at the node-level. With a focus on managing non-uniform memory access on multi-core shared memory processors.

Bohrium serves as the framework for language integration and program representation. Managing the concern of efficient hardware utilization at the node-level is materialized in the construction of the C-targeting Array Processing Engine (CAPE). CAPE envelopes machinery instrumenting parallel runtimes, buffer management, and JIT-compilation with a C-targeting code generator emitting C99 code with OpenMP, Intel LEO, and OpenACC directives. Considerations for non-uniform memory access involves explicit work distribution in the emitted code and managing thread locality in concert by the CAPE runtime.

Benchmarks were implemented using CAPE via Bohrium in Python with NumPy, in C++ with BXX, and without CAPE in C++ with OpenMP. These benchmarks were the basis for a performance evaluation of high-level declarative implementations based on the array notation to explicit low-level parallel code. They form an experimental evaluation of the thesis. Results showed that the implementations using CAPE via Python/NumPy outperformed the low-level code with 20.1%, and the C++/BXX implementation outperformed the low-level code with 69.5%. These results included benchmarks favoring optimizations performed by CAPE. Results on the same benchmarks disabling those optimizations showed that the implementations using CAPE via Python/NumPy performed 12.0% worse than the low-level implementation whereas the C++/BXX implementation performed 1.5% better than the low-level implementation.

What these results describe is that the CAPE JIT-compiler can exploit high-level information to safely apply optimizations that a low-level compiler cannot. A Python implementation can thereby outperform a low-level implementation in C++ when high-level optimizations are applicable. When they are not, then the convenience of Python introduces an overhead of 12.0% in performance.

Consider the result of the BXX DSEL without the optimizations. In this case, the performance matches the low-level implementation. The convenience provided by BXX does not incur a tradeoff in performance. The information conveyed by this result is two-fold. First, this reveals a difference in the two language bridges Python/NumPy and BXX DSEL. This is due to a vulnerability of the algorithm for contraction and composition. Solutions to this problem are provided by ongoing work on fusion at runtime.

Second and most importantly, the CAPE JIT-compiler and runtime can manage the low-level concerns of the architecture required to obtain efficient utilization of a multi-core processor with non-uniform memory access. This result supports my thesis and show the potential of Bohrium, CAPE, and the array-oriented programming model.

Bibliography

[1] Intel Math Kernel Library. http://software.intel.com/en-us/articles/intel-mkl/.

[2] An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[4] Sylwester Arabas, Marc Schellens, Alain Coulais, Joel Gales, and Peter Messmer. Gnu data language (gdl)-a free and open-source implementation of idl. In *EGU General Assembly Conference Abstracts*, volume 12, page 924, 2010.

[5] Sergio Barrachina, Maribel Castillo, Francisco D Igual, Rafael Mayo, and Enrique S Quintana-Orti. Evaluation and tuning of the level 3 cublas for graphics processors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[6] Douglas Bates and Dirk Eddelbuettel. Fast and elegant numerical linear algebra using the rcppeigen package. *Journal of Statistical Software*, 52(5):1–24, 2 2013.

[7] Amr Bayoumi, Michael Chu, Yasser Hanafy, Patricia Harrell, and Gamal Refai-Ahmed. Scientific and engineering computing using ati stream technology. *Computing in science & engineering*, 11(6):92–97, 2009.

[8] Nathan Bell and Jared Hoberock. Thrust: A parallel template library. *GPU Computing Gems Jade Edition*, page 359, 2011.

[9] Lars Bergstrom and John Reppy. Nested data-parallelism on the gpu. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 247–258. ACM, 2012.

[10] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. September 2012.

[11] Laura Susan Blackford. ScaLAPACK. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing 96 Supercomputing 96*, page 5, 1996.

[12] Guy E Blelloch. Nesl: A nested data-parallel language.(version 3.1). Technical report, DTIC Document, 1995.

[13] Guy E Blelloch and Siddhartha Chatterjee. Vcode: A data-parallel intermediate language. In *Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the*, pages 471–480. IEEE, 1990.

[14] Guy E Blelloch and Gary W Sabot. Compiling collection-oriented languages onto massively parallel computers. In *Frontiers of Massively Parallel Computation, 1988. Proceedings., 2nd Symposium on the Frontiers of*, pages 575–585. IEEE, 1988.

[15] Troels Blum, Mads R. B. Kristensen, and Brian Vinter. Transparent GPU Execution of NumPy Applications. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2014 IEEE 28th International*. IEEE, 2014.

[16] Troels Blum and Brian Vinter. Code Specialization of Auto Generated GPU Kernels. In *Communicating Process Architectures 2015*. IOS Press, 2015.

[17] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[18] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

[19] Bradford L Chamberlain. *The design and implementation of a region-based parallel programming language*. PhD thesis, University of Washington, 2001.

[20] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[21] Douglas Crockford. The application/json media type for javascript object notation (json). 2006.

[22] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, 1998.

[23] Peter J Denning. Acm president's letter: What is experimental computer science? *Communications of the ACM*, 23(10):543–544, 1980.

[24] Peter J Denning. Is computer science science? *Communications of the ACM*, 48(4):27–31, 2005.

[25] Jack J Dongarra, James R Bunch, Cleve B Moler, and Gilbert W Stewart. *LINPACK users' guide*, volume 8. Siam, 1979.

[26] Matthew GF Dosanjh, Patrick G Bridges, Suzanne M Kelly, and James H Laros. A peer-to-peer architecture for supporting dynamic shared libraries in large-scale systems. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 55–61. IEEE, 2012.

[27] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[28] Dirk Eddelbuettel. Rcppeigen. In *Seamless R and C++ Integration with Rcpp*, volume 64 of *Use R!*, pages 177–192. Springer New York, 2013.

[29] Dirk Eddelbuettel and Conrad Sanderson. Rcpparmadillo: Accelerating r with high-performance c++ linear algebra. *Computational Statistics & Data Analysis*, 71:1054–1063, 2014.

[30] Wolfgang Frings, Dong H Ahn, Matthew LeGendre, Todd Gamblin, Bronis R de Supinski, and Felix Wolf. Massively parallel loading. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 389–398. ACM, 2013.

[31] Burton S Garbow, James M. Boyle, Cleve B Moler, and Jack J Dongarra. Matrix eigensystem routines eispack guide extension. 1977.

[32] Kazushige Goto and Robert A Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12, 2008.

[33] Linley Gwennap. Adapteva: More flops, less watts. *Microprocessor Report*, 6(13):11–02, 2011.

[34] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, pages 90–95, 2007.

[35] B Jacob and G Guennebaud. Eigen is a c++ template library for linear algebra: Matrices, vectors, numerical solvers, and related algorithms, 2012.

[36] Eric Jones, Travis Oliphant, and Pearu Peterson. Scipy: Open source scientific tools for python. *http://www.scipy.org/*, 2001.

[37] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 7–1. ACM, 2007.

[38] Mads R. B. Kristensen. *Towards Parallel Execution of Sequential Scientific Applications*. PhD thesis, University of Copenhagen, Niels Bohr Institute, Denmark, Blegdamsvej 17, 2100 Copenhagen, Denmark, 9 2012.

[39] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, and Kenneth Skovhede. Separating NumPy API from Implementation. In *5th Workshop on Python for High Performance and Scientific Computing (PyHPC'14)*, 2014.

[40] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster. In *4th Workshop on Python for High Performance and Scientific Computing (PyHPC'13)*, 2013.

[41] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: a Virtual Machine Approach to Portable Parallelism. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 312–321. IEEE, 2014.

[42] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, and Brian Vinter. cphvb: A system for automated runtime optimization and parallelization of vectorized applications. In *Proceedings of The 11th Python In Science Conference (SciPy'12). Austin, Texas, USA.*, 2012.

[43] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[44] Johannes Lund, Mads R. B. Kristensen, Simon A. F. Lund, and Brian Vinter. Just-in-time compilation of numpy vector operations. *Journal on Computing (JoC)*, 3(3), 2014.

[45] Simon A. F. Lund, Bradford L. Chamberlain, and Brian Vinter. Scripting Language Performance Through Interoperability. `http://polaris.cs.uiuc.edu/hpsl/abstracts/a6-lund.pdf`, 2015. [Online; accessed September 2015].

[46] Simon A. F. Lund, Mads R. B. Kristensen, Brian Vinter, and Dimitrios Katsaros. Bypassing the conventional software stack using adaptable runtime systems. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science*, pages 302–313. Springer International Publishing, 2014.

[47] Simon A. F. Lund, Kenneth Skovhede, Mads R. B. Kristensen, and Brian Vinter. Doubling the Performance of Python/NumPy with less than 100 SLOC. In *4th Workshop on Python for High Performance and Scientific Computing (PyHPC'13)*, 2013.

[48] Simon A. F. Lund and Brian Vinter. Automatic mapping of array operations to specific architectures. *submitted to the International Journal on Parallel Computing*, m(n):x–y, 2015.

[49] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[50] Gordon E Moore et al. Progress in digital integrated electronics. *SPIE MILESTONE SERIES MS*, 178:179–181, 2004.

[51] Maxim Naumov. Incomplete-lu and cholesky preconditioned iterative methods using cusparse and cublas. *Nvidia white paper*, 2011.

[52] C.J. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, R. Murty, F. Chinchilla, R. Deodhar, and R. McGuire. Offload compiler runtime for the intel xeon phi coprocessor. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1213–1225, May 2013.

[53] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, Inc., 1996.

[54] Robert W Numrich and John Reid. Co-arrays in the next fortran standard. In *ACM SIGPLAN Fortran Forum*, volume 24, pages 4–17. ACM, 2005.

[55] NVIDIA. Compute unified device architecture. `http://docs.nvidia.com/cuda/`. [Online; accessed June 2013].

[56] Fernando Perez and Brian E Granger. Ipython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29, 2007.

[57] Arch D Robison. Composable parallel patterns with intel cilk plus. *Computing in Science and Engineering*, 15(2):66–71, 2013.

[58] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: As easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 16:1–16:12, New York, NY, USA, 2011. ACM.

[59] Ben Sander. Bolt: A c++ template library for heterogeneous computing. *AMD Fusion Developer Summit*, 12, 2012.

[60] Conrad Sanderson et al. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, Technical report, NICTA, 2010.

[61] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The asynchronous partitioned global address space model. In *Proceedings of The First Workshop on Advances in Message Passing*, 2010.

[62] Sven-Bodo Scholz. Single assignment c: Efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, November 2003.

[63] Jeremy G Siek, Ian Karlin, and Elizabeth R Jessup. Build to order linear algebra kernels. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[64] Jay M Sipelstein and Guy E Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, 1991.

[65] Kenneth Skovhede and Simon A. F. Lund. Numcil and bohrium: High productivity and high performance. In *Proceedings of the 11th International Conference on Parallel Processing and Applied Mathematics (PPAM)*, LNCS. Springer, 2015.

[66] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference (Vol. 1): Volume 1-The MPI Core*, volume 1. MIT press, 1998.

[67] J.E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[68] Alan M Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 42(2):230–265, 1936.

[69] S. Van Der Walt, S.C. Colbert, and G. Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[70] David Vandevoorde and Nicolai M Josuttis. *C++ templates: the Complete Guide*. Addison-Wesley Professional, 2002.

[71] ToddL. Veldhuizen. Arrays in Blitz++. In Denis Caromel, RodneyR. Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer Berlin Heidelberg, 1998.

[72] Brian Vinter, Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, and Kenneth Skovhede. Prototyping for exascale. In *Proceedings of the 3rd International Conference on Exascale Applications and Software*, EASC '15, pages 77–81, Edinburgh, Scotland, UK, 2015. University of Edinburgh.

[73] John Von Neumann. First draft of a report on the edvac. *Annals of the History of Computing, IEEE*, 15(4):27–75, 1993.

[74] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998. CD-ROM Proceedings. **Winner, best paper in the systems category.** URL: `http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps`.

[75] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. Qthreads: An api for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[76] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc - first experiences with real-world applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012.

[77] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 blas performance optimization on loongson 3a processor. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 684–691. IEEE, 2012.

Part II

Publications

6.1 Automatic mapping of array operations to specific architectures

Elsevier Editorial System(tm) for Parallel Computing

Manuscript Draft

Manuscript Number:

Title: Automatic mapping of array operations to specific architectures

Article Type: Research Paper

Keywords: High-level languages; Array programming; Just-in-time compilation; Code generation; Parallelization; non-uniform memory access

Corresponding Author: Mr. Simon Andreas Frimann Lund, M.Sc.

Corresponding Author's Institution: University of Copenhagen

First Author: Simon Andreas Frimann Lund, M.Sc.

Order of Authors: Simon Andreas Frimann Lund, M.Sc.

Abstract: Array-oriented programming has been around for about thirty years and provides a fundamental abstraction for scientific computing. However, a wealth of popular programming languages in existence fail to provide convenient high-level abstractions and exploit parallelism. One reason being that hardware is an ever-moving target.

For this purpose, we introduce CAPE, a C-targeting Array Processing Engine, which manages the concerns of optimizing and parallelizing the execution of array operations. It is intended as a backend for new and existing languages and provides a portable runtime with a C-interface.

The performance of the implementation is studied in relation to high-level implementations of a set of applications, kernels and synthetic benchmarks in Python/NumPy as well as low-level implementations in C/C++. We show the performance improvement over the high-productivity environment and how close the implementation is to handcrafted C/C++ code.

Cover letter

Please consider this manuscript for publication in Journal of Parallel Computing.

Title: Automatic mapping of array operations to specific architectures

Affiliations:
Simon Andreas Frimann Lund
Niels Bohr Insistute, University of Copenhagen, Denmark

Brian Vinter
Niels Bohr Insistute, University of Copenhagen, Denmark

Problem: Mapping high-level language constructs and operations upon them to specific hardware architectures. Specifically representations of multi-dimensional arrays and operations with and upon them to heterogeneous hardware configuration with a focus on non-uniform memory access.

Approach: Integrate with existing languages via vector bytecode, an intermediate representation for array operations. Focus here on the transparent support of operations via array-notation, that is, entirely declarative expressions without any explicit terms or hints from the application programmer. The focus is on presenting and experimentally evaluate the performance of an array processing engine, which manage memory, specialize, JIT-compile, and execute array-operations.

Contributions:

- Code generator for array operations with parallelization and composition of multiple array operations
- Caching JIT-Compiler and object storage for array operation kernels
- Runtime instrumenting compilation, buffer management, array operation scheduling and execution

Overlap: Brian Vinter, Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, and Kenneth Skovhede. 2015. Prototyping for Exascale. In *Proceedings of the 3rd International Conference on Exascale Applications and Software* (EASC '15), A. Gray, L. Smith, and M. Weiland (Eds.). University of Edinburgh, Edinburgh, Scotland, UK, 77-81.
In the *Prototyping for Exascale* paper an early iteration of the implementation was used in a performance evaluation, however, the paper does not discuss the design and implementation of the contributions presented above. Nor does it discuss the results in relation to the goals of the manuscript submitted for consideration of publication.

Closest prior:
Newburn, Chris J., et al. "Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language." Code generation and optimization (CGO), 2011 9th annual IEEE/ACM international symposium on. IEEE, 2011.
The paper stated above is the closest related prior which explore the use of retargetable and dynamic compilation via an intermediate representation.
The work presented in the manuscript submitted for consideration of publication improves upon prior work in several areas. Support for multiple languages is presented as diverse as Python and C++. Support and consideration non-uniform memory access and accelerators is presented. NUMA is evaluated and compared to hand-coded implementations. A different design to the dynamic compilation approach is presented.

***Highlights (for review)**

* Code generator for array operations with parallelization and composition of multiple array operations
* Caching JIT-Compiler and object storage for array operation kernels
* Optimized memory allocation scheme for array buffers
* Runtime instrumenting compilation, buffer management, array operation scheduling and execution
* Benchmark applications comparing Python/NumPy applications with equivalent serial and parallel C/C++ implementations

Automatic mapping of array operations to specific architectures

Simon A. F. Lund, Brian Vinter

Niels Bohr Institute, University of Copenhagen, Copenhagen, Denmark

Abstract

Array-oriented programming has been around for about thirty years and provides a fundamental abstraction for scientific computing. However, a wealth of popular programming languages in existence fail to provide convenient high-level abstractions and exploit parallelism. One reason being that hardware is an ever-moving target.

For this purpose, we introduce CAPE, a C-targeting Array Processing Engine, which manages the concerns of optimizing and parallelizing the execution of array operations. It is intended as a backend for new and existing languages and provides a portable runtime with a C-interface.

The performance of the implementation is studied in relation to high-level implementations of a set of applications, kernels and synthetic benchmarks in Python/NumPy as well as low-level implementations in C/C++. We show the performance improvement over the high-productivity environment and how close the implementation is to handcrafted C/C++ code.

Keywords: high-level languages, array programming, just-in-time compilation, code generation, parallelization, non-uniform memory access

1. Introduction

Scientific computing demands high performance to enable processing of real-world dataset and increasing simulation accuracy.

Traditional languages used in High-Performance Computing (HPC) such as C, C++, and Fortran provide low-level control over the hardware. Control that enables the programmer to utilize the available memory hierarchy and processing units efficiently with the goal of maximizing program throughput.

A key concern for programs written in low-level languages is that performance is inherently non-portable due to the tight coupling between application logic and hardware instrumentation. Thus re-targeting a program written in a low-level language requires re-implementation.

Different layers of abstraction can aid performance portability, such as using optimizing compilers to abstract differences in instruction set architecture. Directive based programming as facilitated by OpenMP[1], Intel LEO[2], and

OpenACC[3] encapsulate parallelization on multi-core and many-core architectures. Other approaches include OpenCL[4], which provides a common framework for expressing parallel programs on both multi-core and many-core architectures. Despite these advances in abstraction and encapsulation for traditional HPC languages, expressing on-node parallelism remains tightly coupled as OpenMP is preferable for targeting multi-core CPUs, LEO for Intel accelerators and graphics, and OpenACC for NVIDIA graphics.

Applying low-level languages to scientific computing thus seem like a necessary evil to meet performance requirements. However, high performance is not the only demand; exploratory scientific computing requires rich interactive environments for data exploration facilitated by visualization, experimentation and prototyping. High-level languages such as Python/NumPy[5], Matlab[6], R[7], Octave[8], and IDL[9] caters to these demands. The open source packages Python/IPython[10]/NumPy[5]/SciPy[11] is an example of an increasingly popular software stack for scientific computing. A stack based on an array-oriented programming model facilitated by NumPy.

The declarative expression form of array-oriented programming allows for convenient abstractions from low-level concerns, albeit at the cost of trading in the performance obtainable from explicitly instrumenting hardware. Python/NumPy has despite the performance trade-off made its way into large compute-sites due to projects such as PyOpenCL[12], PyCUDA[12], and PyMIC[13]. These libraries offer interoperability with low-level languages and APIs. Python/NumPy in conjunction with such libraries provides the rich environment required for exploratory scientific computing and a more convenient approach to low-level programming.

Low-level programming in a high-level language thus seems to be a favorable approach for scientific computing when compared to a purely low-level approach. However, it has its shortcomings, since it is as vulnerable to the concerns of performance portability as a purely low-level implementation, and the high-level abstraction and encapsulation is lost.

The quest for performance is thus a resource demanding task that does not align well with the needs of exploratory scientific computing. Subsection 1.2 exemplifies some of the performance challenges that high-level declarative approaches face. Subsection 1.1 exemplifies the low-level pitfalls a programmer might get stuck in when faced with low-level concerns. Subsection 1.3 provide an overview of the contributions of the work in this paper and how it provides a means to overcome the high-level performance challenges and low-level pitfalls.

1.1. Low-Level pitfalls on NUMA architectures

Parallel programming has well-known pitfalls such as race conditions and deadlocks. Pitfalls that do not reveal them themselves until a program enters a certain state at runtime and are thus hard to reproduce and debug. Multi-core architectures with non-uniform memory access (NUMA) introduce performance pitfalls in addition to the challenge of implementing a correct parallel program.

The severity of the performance degradation is illustrated best with an example. Figure 1 provides pseudo-code for a synthetic benchmark simulating a

```
# SS: Serial initialization and access
memset(grid, 0, sizeof(*grid)*nelements);
for(size_t eidx=0; eidx<nelements; ++eidx) {
    grid[eidx] = (grid[eidx]+1)*0.2;
}
```

```
# SP: Serial initialization, parallel access
memset(grid, 0, sizeof(*grid)*nelements);
#pragma omp parallel for
for(size_t eidx=0; eidx<nelements; ++eidx) {
    grid[eidx] = (grid[eidx]+1)*0.2;
}
```

```
# PP: Parallel initialization and access
#pragma omp parallel for
for(size_t eidx=0; eidx<nelements; ++eidx) {
    grid[eidx] = 0.0;
}
#pragma omp parallel for
for(size_t eidx=0; eidx<nelements; ++eidx) {
    grid[eidx] = (grid[eidx]+1)*0.2;
}
```

Figure 1: C-like pseudocode for synthetic scientific workload, grid is a data-pointer, and nelements quantify the number of elements that the grid points to, full source-code available at `http://benchpress.readthedocs.org/benchmarks/synth_init.html`

common workload. That is, at first a grid is initialized and secondly accessed to update it.

The benchmark demonstrates the scalability challenges due to remote access on NUMA architectures, figure 2 provide a speedup graph using the serial implementation *SS* as a baseline.

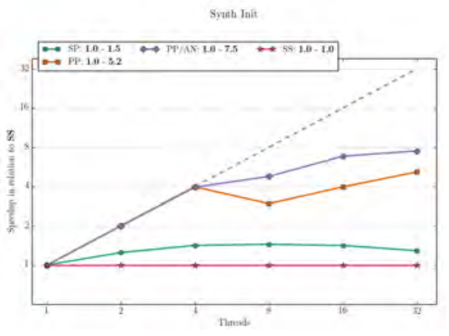


Figure 2: Ten timesteps updating array of 100000000 doubles. Effects of ccNUMA on scalability

The configuration used to run the benchmark had a total of 32 cores and 128GB of shared memory. The configuration partitions these resources across two sockets each with an AMD Opteron 6272 processor; each processor has two NUMA-nodes. Each NUMA-node consists of eight cores and an on-chip memory controller with two channels to 32GB of memory.

First, comparing *SS* to *SP* on figure 2. The best speedup of $\times 1.5$ is achieved with eight threads, at 32 threads performance degrades to $\times 1.3$. Obtaining a mere $\times 1.3$ speedup on a system with 32 cores illustrate how devastating NUMA-effects can be to performance. The reason is that the serial initialization will commit pages on the NUMA-node on which the single thread is running. Forcing the majority of threads in the parallel loop to access remote memory.

Secondly, we compare *SS* to *PP* on figure 2. The synthethic *PP* implementation is intended to illustrate that the programmer has parallelized initilization. With 32 threads the best speedup of $\times 5.2$ is obtained.

What is noticeable here is that linear speedup is obtained up to four threads. Up to four threads the operating system kernel is able to dynamically schedule threads to avoid remote access. Exceeding four threads, the dynamic scheduling pays a performance hit as threads either has to access remote memory or migrate to a different core on a NUMA-node to which the data is local.

Third and last, we compare *SS* to *PP/AN*, in this case, the best speedup of $\times 7.5$ is obtained with 32 threads. The point worth noting for this result is that the implementation of *PP/AN* is identical to *PP*. The difference is that the runtime parameter `GOMP_CPU_AFFINITY` is set to control thread-affinity. Binding threads is advantageous when it matches parallel loop distribution. The static distribution of the parallel loop divide iterations into chunks, for the results provided in figure 2, then 100000000 iterations are divided with a chunk size of $\frac{loop-count}{thread-count} = 3125000$, when executing with 32 threads. Thread0 is thus responsible for executing loop-iteration $1-3125000$ and thereby implicitly array elements with index $0 - 3124999$. Binding thread0 to core0 that is on NUMA node0, ensures that thread0 will be assigned the same sequence of iterations across executions of parallel loop constructs. Thereby avoiding remote access since the array elements are available on the NUMA-node on which the thread is executing.

Obtaining a best-case speedup of $\times 7.5$ on a 32 core system might be dissappointing. However, not in this case, since the non-tuned synthetic benchmark is inherently memory-bound as it barely performs any compute work for each update of the array. The number of compute cores in the machine is not the bottleneck, the main-memory bandwidth is. The execution of *PP/AN* with 32 threads achieves a throughput of 30.04GB/s. For comparison the tuned STREAM[14] (Triad) benchmark was executed and obtained a throughput of 43.1GB/s on the same machine.

The serial initialization vs parallel access is labeled as a performance pitfall as profiling the code will show the data-access loop as the one consuming the most time. A programmer might therefore not consider how data is initialized and less synthethic applications might use a random number generator for initialization or loading data from file.

Another point worth mentioning which the speedup graph and discussion of results above do not reveal is that the parallel execution without an affinity policy has a high deviation from the mean in terms of elapsed wall-clock time. The deviation is due to the dynamic scheduling of threads performed by the operating system.

Thread scheduling and data locality is thus essential concerns for the programmer to achieve reasonable throughput and predictable performance.

1.2. Performance challenges for High-Level Approaches

Listing 1 provides an example of a Python/NumPy implementation of a heat equation solver. The code exemplifies the declarative expression-form that Python/NumPy and other array-oriented language provide. In addition to the syntax and expression-form the code illustrates some performance enhancing features. Lines 3-7 construct array views which are references to subsets of the grid elements equivalent to those illustrated in figure 3. The use of views facilitate a significant reduction in required memory in contrast to copying them. Another feature of NumPy is the use of operator overload, as a consequence the operations: `*`, `+`, `-`, `abs`, and `sum` are delegated to routines implemented in C. The data-descriptor for NumPy arrays is also an efficient representation for multi-dimensional arrays when compared to using built-in Python data structures such as lists.

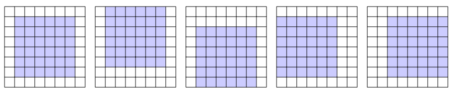


Figure 3: Array views for five-point stencil.

```
1  import numpy as np
2  solve(grid, epsilon):
3    center = grid[ 1:-1, 1:-1]
4    north  = grid[-2:  , 1:-1]
5    south  = grid[ 2:  , 1:-1]
6    east   = grid[ 1:-1,  :2 ]
7    west   = grid[ 1:-1, 2:  ]
8    delta  = epsilon+1
9    while delta > epsilon:
10     tmp = 0.2 * (
11       north + south +
12       east  + west  +
13       center
14     )
15     delta = np.sum(
16       np.abs(tmp-center)
17     )
18     center[:] = tmp
```

Listing 1: Python/NumPy implementation of the heat equation solver. The `grid` is a two-dimensional NumPy array and the `epsilon` is a Python scalar.

However, porting the example from figure 1 to a language such as C can still yield a performance improvement of an order of magnitude. That is even without considering parallelization. The main obstable lies with the interpreted nature of Python, since when each array operation in the loop body is executed it materializes an array to store the result. A consequence is that Python/NumPy become inherently memory-bound.

Considering parallelization would introduce an even larger performance gap. A low-level implementation will store the results of the heat-equation in registers and thus latency-hide the load/store of grid values. Even if the NumPy array operations would be implemented as parallel routines in C, they would remain memory bound due to the temp array materialization.

1.3. Our contributions

We introduce CAPE a C-targeting Array Processing Engine. The purpose of which is to obtain the best of both worlds, that is, the performance from explicitly instrumenting array operations in a low-level language with the productivity of the high-level abstractions. To this end, CAPE supports high-level declarations of array operations, dynamically generates and compiles efficient code for them, manages memory by allocating supporting buffers backing the arrays and executes the operations.

- Code generator for array operations with parallelization and composition of multiple array operations (Section 3.2.1);
- Caching JIT-Compiler and object storage for array operation kernels (Section 3.2)
- Optimized memory allocation scheme for array buffers (Section 3.3).
- Runtime instrumenting compilation, buffer management, array operation scheduling and execution (Section 3).
- Benchmark applications for comparing Python/NumPy applications with equivalent serial and parallel C/C++ implementations (Section 4).

This paper integrates previous work on Bohrium[15] (Section 2), a framework and runtime for extracting array-operations from languages. Bohrium allows for experimentation with subprogram optimization, specifically the data-parallel array operations.

This paper focuses on introducing the design of CAPE and restricts the scope of the implementation and performance study to multi-core CPUs with ccNUMA architecture. Although supporting other compute-oriented architectures such as Intel MIC, GPUs and APUs, are an integral part of the design of CAPE, it is out of scope for this paper to describe the implementation. Section 5, on ongoing and future work, provide an overview of how CAPE addresses these architectures. Throughout the description of the CAPE implementation, attention will be given on these architectures, yet a comparative performance study is out of scope for this paper.

2. Bohrium

Bohrium is a framework which significantly reduces the costs associated with high-performance program development. Bohrium provides the mechanics to couple a programming language or library with an architecture-specific implementation seamlessly. These mechanics cover two conceptually different areas: programming language bridges (subsection 2.2) and runtime components (subsection 2.3). These two areas are bound together by the program representation vector bytecode (subsection 2.1).

2.1. Vector Bytecode

A vital part of Bohrium is the *Vector Bytecode* that constitutes the link between the high-level user language and the low-level execution engine. The bytecode is designed with the declarative vector programming model in mind where the bytecode instructions operate on input and output vectors. To avoid excessive memory copying, the vectors can also be shaped into multi-dimensional vectors. These reshaped vector views are then not necessarily comprised of elements that are contiguous in memory. Each dimension comprises a stride and size, such that any regularly shaped subset of the underlying data can be

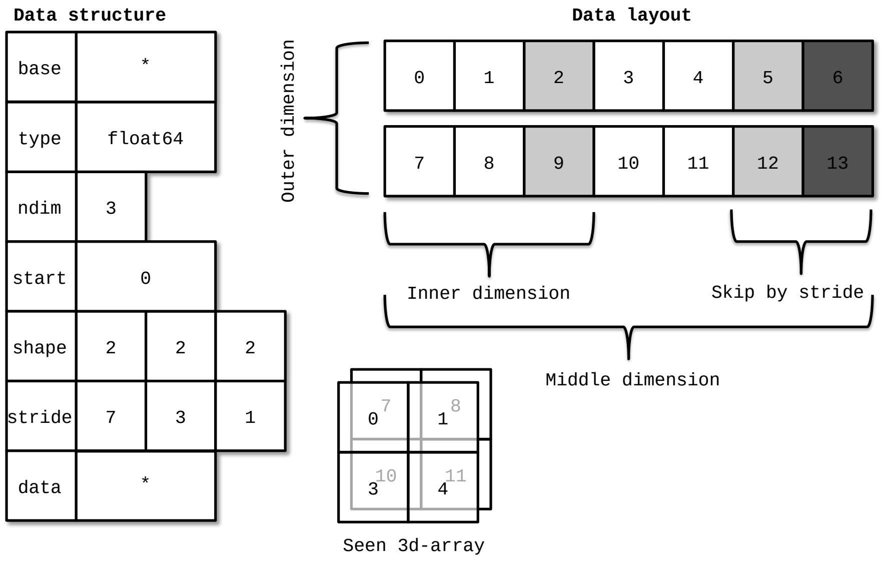


Figure 4: Descriptor for n-dimensional vector and corresponding interpretation

accessed. We have chosen to focus on a simple, yet flexible, data structure that allows us to express any regularly distributed vectors. Figure 4 shows how the shape is implemented and how the data is projected.

The aim is to have vector bytecode that support data parallelism implicitly and thus makes it easy for the bridge to translate the user language into the bytecode efficiently. Additionally, the design enables the VE to exploit data parallelism. In the following we will go through the six types of vector bytecodes in Bohrium.

Element-wise bytecodes performs a unary or binary operation on all vector elements. Bohrium currently supports 53 element-wise operations, e.g. addition, multiplication, square root, logical and, bitwise and, equal and less than. For element-wise operations, we only allow data overlap between the input and the output vectors if the access pattern is the same, which, combined with the fact that they are all stateless, makes it straightforward to execute them in parallel.

Reduction bytecodes reduce an input dimension using a binary operator. Again, we do not allow data overlap between the input and the output vectors and the operator must be associative[1]. Bohrium currently supports 10 reductions, e.g. addition, multiplication and minimum. Even though none of them are stateless, the reductions are all straightforward to execute in parallel because of the non-overlap and associative properties.

Scan bytecodes accumulate an input dimension using a binary operation. Again, we do not allow data overlap between the input and the output vectors

[1]Mathematically associativity; we allow non-associativity because of floating point approximations

and the operator must be associative[2]. Bohrium currently supports 10 scans, e.g. addition, multiplication and minimum.

Gather/Scatter bytecodes perform indexed reads and writes.

Data Management bytecodes determine the data ownership of vectors, and consists of three different bytecodes. The *SYNC* bytecode instructs a child component to place the vector data in the address space of its parent component. The *FREE* bytecode instructs a child component to deallocate the data of a given vector in the global address space. Finally, the *DISCARD* operator instructs a child component to deallocate any meta-data associated with a given vector, and signals that any local copy of the data is now invalid. These three bytecodes enable lazy allocation where the actual vector data allocation is delayed until it is used. Often vectors are created with a generator (e.g. random, constants) or with no data (e.g. temporary), which may exist on the computing device exclusively. Thus, lazy allocation may save several memory allocations and copies.

Extension methods The above three types of bytecode make up the bulk of a Bohrium execution. However not all algorithms may be efficiently implemented in this way. In order to handle operations that would otherwise be inefficient or even impossible, we introduce the fourth type of bytecode: *extension methods.* We impose no restrictions to this generic operation; the extension writer has total freedom. However, Bohrium does not guarantee that all components support the operation. Initially, the user registers the extension method with paths to all component-specific implementations of the operation. The user then receives a new handle for this *extension method* and may use it subsequently as a vector bytecode. Matrix multiplication and FFT are examples of extension methods that are obviously needed. For matrix multiplication, a CPU specific implementation could simply call a native BLAS library and a Cluster specific implementation could call the ScaLAPACK library[16].

2.2. Language Bridges

Language bridges are responsible for providing convenient high-level abstractions within a given language and mapping these abstractions to *Vector Bytecode* for processing by the Bohrium runtime.

Currently Bohrium has bridges for Python via npbackend[17], which provides a transparent mapping of Python/NumPy array operations. The Microsoft CIL-based languages are supported via the NumCIL[18][19] library. A C-interface exists but is not intended for the application programmer, it is instead provided as a building block for implementing language bridges. C++ is supported via the BXX library which provides a multidimensional array that can be utilized conveniently via operator overloads and a suite of math and linear algebra functions.

[2]Mathematically associativity; we allow non-associativity because of floating point approximations

```
 1 ADD t1, center, north
 2 ADD t2, t1, south
 3 FREE t1
 4 DISCARD t1
 5 ADD t3, t2, east
 6 FREE t2
 7 DISCARD t2
 8 ADD t4, t3, west
 9 FREE t3
10 DISCARD t3
11 ...
```

```
12 ...
13 MUL tmp, t4, 0.2
14 FREE t4
15 DISCARD t4
16 MINUS t5, tmp, center
17 ABS t6, t5
18 FREE t5
19 DISCARD t5
20 ADD_REDUCE t7, t6
21 FREE t6
22 ...
```

```
23 ...
24 DISCARD t6
25 ADD_REDUCE delta, t7
26 FREE t7
27 DISCARD t7
28 COPY center, tmp
29 FREE tmp
30 DISCARD tmp
31 SYNC delta
```

Figure 5: Bytecode generated in each iteration of the Python/NumPy implementation of the heat equation solver (Fig. 1). Note that the `SYNC` instruction at line `31` transfers the scalar `delta` from the Bohrium address space to the NumPy address space in order for the Python interpreter to evaluate the `while` condition (Fig. 1, line `9`).

Figure 5 illustrates the list of vector byte code that the NumPy Bridge will generate when executing one of the iterations in the Python/NumPy implementation of the heat equation solver (Fig. 1). The example demonstrates the nearly one-to-one mapping from the NumPy vector operations to the Bohrium vector byte code. The code generates seven temporary arrays (`t1`,...,`t7`) that are not specified in the code explicitly but are a result of how Python interprets the code. In a regular NumPy execution, the seven temporary arrays translate into seven memory allocations and de-allocations thus imposing an extra overhead.

2.3. Runtime Components

Regardless of the language bridge used, then the Bohrium runtime processes array operation based on its stack configuration. The stack configuration consists of an ordering of Bohrium runtime components. A language bridge sends *Vector Bytecode* to the topmost component and it trickles down the stack through the different components. Components are labeled by their role, that is by the task they are intended to perform. The most essential component types are:

Filter components are used for transforming *Vector Bytecode*. A simple runtime-value optimizing filter could for example map *pow* to *mul* and *sqrt* for suitable values of the exponent.

Vector Engine Manager (VEM) components are responsible for one memory address space in the hardware configuration. The current version of Bohrium implements two VEMs: the Node-VEM that handles the local address space of a single machine and the Cluster-VEM that handles the global distributed address space of a compute cluster.

The Node-VEM is very simple since the hardware already provides a shared memory address space; hence, the Node-VEM can simply forward all instruction from its parent to its child components. The Cluster-VEM, on the other hand, has to distribute all vectors between Node-VEMs in the cluster.

Vector Engine (VE) components has to execute instructions it receives in an order that comply with the dependencies between instructions. Further-

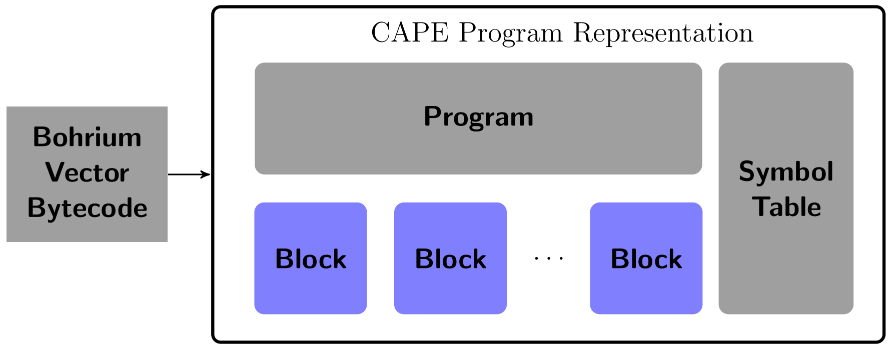


Figure 6: Mapping Bohrium Vector Bytecode to CAPE TAC program representation.

more, it has to ensure that its parent component has access to the results as governed by the Data Management bytecodes. Vector engine components are thus responsible for implementing the low-level details for a given hardware target.

Bohrium consists of a number of components that communicate by exchanging Vector Bytecode. Because they all use the same communication protocol they can be combined in various ways. The idea is to make it possible to combine components in a setup that match a specific execution environment. Re-targeting an application using Bohrium therefore does not require changing application code, the runtime stack-configuration is simply changed to match the execution environment. Changing stack configuration can be done by modifying the default configuration via the Bohrium configuration configuration file. Re-targeting can alternatively be done by selecting a predifined configuration via the environment variable `BH_STACK`.

3. CAPE

CAPE is in Bohrium terms a vector engine. That is, it is the component in the Bohrium stack which is responsible for managing the low-level concerns of mapping array operations, represented as vector bytecode, to a specific architecture.

3.1. Program Representation

The first task for the CAPE runtime, when executing a vector bytecode program, is to run two passes over the bytecode instructions to construct auxiliary information about instructions and operands, and organize instructions into executable blocks.

The first pass constructs an array of instructions represented as CAPE three-address-code (`tac`/`kp_tac`) and an array of CAPE operands (`kp_operand`). We refer to the former as the `program` and the latter as the `symbol table`. A `kp_tac` is equivalent to a Bohrium bytecode except it uses symbolic names for operands, specifically, numerical values that can be used to look up the operand array

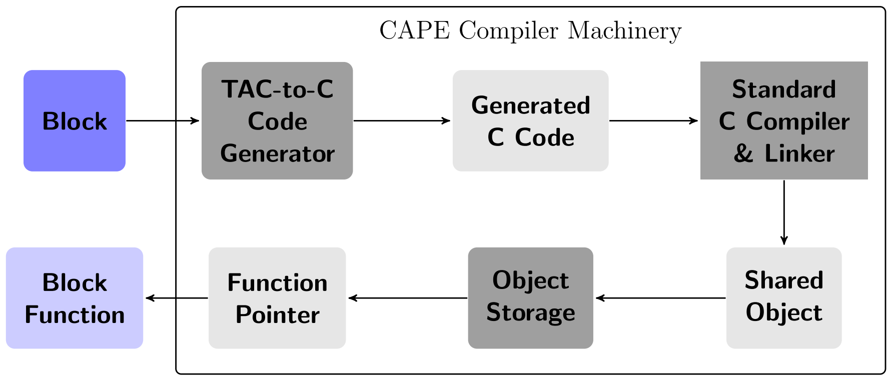


Figure 7: JIT-compiler machinery

descriptor in the `symbol table`. The array-descriptor, `kp_operand`, is similar to the data-structure described in 4 with an additional attribute named `layout`. A `kp_operand` describes how to access an underlying buffer (`kp_buffer`).

The second pass constructs CAPE `blocks` (`kp_block`). A `block` consists of a subset of `tacs` in the `program`, an iteration space (`kp_iterspace`), a unique identifier and a hashed representation of the identifier. A `block` additionally maintains a local scope, that is, the `symbol table` is regarded as globally scoped operands, and the operands used by instructions in the block has an additional symbolic value which is "local" to the block.

Blocks form the basic unit for scheduling and executing array operations. How the JIT-compiler and execution model use this program representation is described in the following sections.

3.2. JIT-Compiler

The JIT-compiler handles the construction of *executable* code for a given block. The machinery to facilitate this purpose is illustrated in figure 7. Given a block, it returns a block function. To do so it generates C source code using a Tac-to-C code generator (Section 3.2.1). The generated C source code is compiled using a standard C compiler & linker to produce a shared object containing the block function. Object storage (Section 3.2.2) dynamically loads the object and returns a pointer to the block function.

3.2.1. Code Generator

CAPE is C-targeting, the meaning of this phrasing is that it generates C source code, specifically C99. Also, it is meant to convey the information that CAPE could instrument any hardware target that is addressable with C99 and a standard compiler.

Other choices for code-targets include CUDA C and translation with the `nvcc` CUDA compiler driver, thereby also limiting hardware targets to CUDA architectures. Alternatively, by targeting OpenCL C and letting the OpenCL compiler driver translate generated code, one could provide support for a wider

```
void KP_IDENTHASH(kp_buffer** buffers,
                  kp_operand** operands,
                  kp_iterspace* iterspace)
{
// Buffers   (data pointers)
double* buf0_data = buffers[0]->data;
...

// Operands  (strides and buffer offset)
const int64_t opd0_start = operands[0]->start;

// Iterspace (shape, layout, #elements)
const int64_t iterspace_nelem = iterspace->nelem;
...

//    Parallel section   - prequel

{ //  Parallel section   - entry

  //  - operand to buffer mapping
  double* restrict opd0 = buf0_data + opd0_start;
  double opd1;
  //  - work distribution
  //  - initialize accumulator variables

  //  Parallel section   - loop constructs
  {
      ...array operations...
  }
  //  Parallel section   - exit
}

//    Parallel section   - sequel
}
```

Figure 8: C-like pseudocode skeleton for block functions.

range of hardware targets using the same programming model for generated code.

Using OpenCL is indeed enticing, and initial prototyping experimented with OpenCL but was discarded primarily for matters of practical concern altough also for performance considerations. Matters such as a series of oddities with different OpenCL implementations that led to significant compilation overhead and brittle drivers causing unstable and curious behavior. A performance consideration was the lack of control over thread-locality, there was no means to perform thread binding which as illustrated earlier is essentiel to performance on NUMA architechtures. This was prior to the introduction of Device Fission in OpenCL 1.2. The oddities of OpenCL implementations outweighed the convenience of a unified programming model which led to the choice of generating C99 and using OpenMP, Intel LEO, and OpenACC directives for parallelization and HWLOC to control locality.

Figure 8 illustrate the general structure and signature of a block function. A block function only performs compute-oriented array operations and is in this sense somewhat similar to a CUDA or OpenCL kernel. The CAPE runtime handles data management.

The first part of a block function "unpacks" arguments, "unpacking" in the sense that the struct attributes of buffers, operands and the iteration space are declared as "flat" local variables. Examples of some of the unpacking is provided in figure 8 line 6, 10, and 13. Exactly which attributes are unpacked depends on the requirements of the code in the parallel section. The motivation for this unpacking is to reduce the amount of data transferred into the parallel section. Line 21 in figure 8 illustrates similar a concern. Here the mapping of an operand `opd0` to its supporting buffer `buf0` with offset `start` is declared and annotated with `restrict` when applicable. Such that the C compiler can enforce strict aliasing rules and reduce alias checking at runtime.

The actual computations on array elements are performed in the loop constructs of the parallel section. The multi-dimensional array descriptor used by operands and described in figure 4 drives these loop constructs. The descriptor uses the following coordinate based formula to calculate the address of an element.

$$\texttt{addr} = \texttt{base\_addr} + \sum_{\texttt{r}=0}^{\texttt{rank-1}} \texttt{stride[r]} \times \texttt{coord[r]}$$

A loop construct implementing this formula is quite inefficient as it must use memory for maintaining the coordinate and additionally perform an excessive amount of calculations per element.

To circumvent this issue, the code generator emits specialized loop constructs. Loop constructs are specialized using the `KP_LAYOUT` values for operands, iteration space, as well as the dimensionality/rank. The most significant values are: `KP_SCALAR`, `KP_CONTRACTABLE`, `KP_CONTIGUOUS`, `KP_CONSECUTIVE`, and `KP_STRIDED`.

The `KP_SCALAR` layout hints that the operand is either an actual scalar or a broadcasted/flooded. Broadcasted scalars occur in an array expression such as: `a + 42.0` where `a` is an array. In which case the scalar `42.0` would be elevated to an array described with the same shape as `a` and stride = 0 in each dimension. By using the layout information the general formula can be entirely discarded.

The `KP_CONTRACTABLE` is one of the most significant drivers for emitting efficient code. It signifies that the operand is only used within the block, that is, to store intermediate values. The code generator only need to emit a placeholder variable such as the one on line 22 in figure 8.

The `KP_CONTIGUOUS` and `KP_CONSECUTIVE` are also essential means of avoiding to implement the general formula. Both signify that the following linear formula can be used instead.

$\texttt{addr} = \texttt{base\_addr} + \texttt{k} * \texttt{element\_number}$

Where `k` is a constant, `k=1` in the case of `KP_CONTIGUOUS` and `k>1` for `KP_CONSECUTIVE`. This information allow the code generater to translate into code that increment operand data pointers using either `opd0++` or `opd0 += constant_stride`.

There are, however, a caveat for exploiting the information in this way as it forces a specific ordering on the traversal of elements. If the operations performed on the arrays are element-wise or the array are one-dimensional then it is safe. Conversely if the arrays have rank larger than one and a partial reduction or a scan is performed as one of the operations in the block and one of these operations are not performed in the same order as the constant increment imposes, then the optimization must be discarded.

The last layout value, `KP_STRIDED`, is the least attractive as it signifies that the stride information from the array descriptor must be used. However, a last effort is attempted to avoid the general formular. That is, a loop nest is added for each dimension, and a step-value for each dimension is precalculated. The step-value is equal to the stride in the given dimension minus the step-values of the loop nest inside it. Using the step-value, a step in the current loop nest can increment the operand with code such as `opd0 += step_value`.

The specialization of the pointer arithmetic and loop constructs serve the purpose of avoiding the computationally and memory-expensive general formular. Even more importantly they are generated to serve specialized work distribution in the entry of the parallel section. The specialized work distribution targets even distribution of work between NUMA nodes, to optimize for memory throghput. The flattening of loop constructs and division of the strided loop constructs minimize remote access.

The current implementation is thus primarily focused on emitting efficient code for multi-core NUMA architectures with regards to work distribution and loop constructs.

However, the careful unpacking of meta-data and restricted transfer of meta-data to parallel sections are also performed with thought of accelerators, as parallel sections on these architectures means transferring not only the buffer data as handled by the runtime but also the meta-data to device. The area left for future work is thus the specialization of loop constructs.

The result of code generator is a string in memory, this string is either piped to the C compiler or stored in the filesystem using the naming convention `KP_IDENTHASH.c` where `IDENTHASH` is an ASCII representation of the block identifier.

3.2.2. Object Storage

Generating source code, invoking the C compiler, and loading the generated code consumes about 70-80 miliseconds of wall-clock time for each block function. Executing the block function can take anywhere from a few microseconds to several seconds of wall-clock time. These numbers vary based on a long list of parameters including C compiler flags, the specific compiler used, complexity of the kernel function, I/O throughput of the system, and the number of elements in the arrays when executing the block-function.

A worst-case scenario for a JIT-compilation approach is that time spent on JIT-compilation and execution exceeds what would othervise be spent on executing statically compiled code. In literature and production systems different policies of 'if' and 'when' to JIT-compile are used. CAPE applies an aggressive policy. To minimize compilation overhead a caching object storage is used instead. The cache persists to the filesystem and is reused and shared with other processes via the filesystem. The CAPE policy is always compile, unless the block function exists in cache.

```
process(bytecode) {

  program, symbol_table, blocks[] = map(
    bytecode
  );

  thread_manager.bind();

  for block in blocks {
    block_function = jit_compiler(block);
    memory_manager(block);  // Allocation
    block_function(block);  // Execution
    memory_manager(block);  // De-allocation
  }
}
```

Figure 9: Pseudo-code illustration of the bytecode processing performed by the CAPE runtime.

The design of the CAPE compiler allows for compiling block-functions ahead of time in a addition to just in time. Thereby allowing execution to overlap the compilation phase. However, in the current implementation the runtime does not exploit this, but run in a simpler lookup/compile/execute mode.

Objects are stored in the filesystem using two naming conventions, the first names objects as `KP_IDENTHASH.ext`, where `IDENTHASH` is an ASCII representation of the block identifier and `.ext` is the extension name used for shared libraries on the given platform. This naming convention is used when objects are created at runtime by the JIT-compiler. The second convention names objects as `KP_COLLECTION.ext` which is accompanied by another file named `KP_COLLECTION.index`. This naming convention is used for pre-compilation and the `COLLECTION` object therefore contains multiple block functions, the identifiers of which are stored in the `.index` file.

Since object storage is shared via the filesystem, care is taken to protect against race condition that can occur when multiple processes are running or when using a shared filesystem. Additionally, care is taken to avoid sharing objects compiled for different hardware targets.

3.3. Execution Model

As Section 3.2 describe, then block functions execute the compute-oriented array operations. While data management operations, integration with other runtimes, libraries, and the JIT-compilation machinery is the responsibility of the CAPE runtime. Figure 9 demonstrates the general execution model used by CAPE.

Thread Manager

Figure 2 demonstrated the need for managing thread affinity to avoid remote access on NUMA architectures. In the example from the figure, thread affinity is controlled by instrumenting the GNU OpenMP runtime via the environment variable `GOMP_CPU_AFFINITY`. Such an approach is not desirable for the CAPE runtime for several reasons.

First, the concerns of the hardware architecture should be hidden from end-user. Be it a programmer/scientist doing exploratory scientific computing or a user running the program written by the scientist. The JIT-compiler machinery can use any standard C compiler and as a consequence thereof, different OpenMP runtimes. Different OpenMP runtimes have different environment variables for managing thread affinity, such as `KMP_AFFINITY` for Intel compiler, and `MP_BIND/MP_BLIST` for the Portland Group compiler. Delegating control over thread affinity to the end-user would thus expose, not only the hardware architecture but also the CAPE compiler machinery.

Secondly, the applied thread binding has performance consequences for the loop constructs in block functions. It is thus essential for performance that CAPE runtime and the code generator complement each other in the efforts of load distribution of the parallel sections and binding of threads.

The CAPE runtime, therefore, implements thread binding. The HWLOC[20] library is used to traverse the topology of processing units on the system and binding threads. The thread manager is invoked prior to processing blocks as figure 9 illustrate.

Threads are bound using a single affinity policy. The code generator is aware of the thread policy and generates code optimizing for the policy. An area left for future work is the exploration of dynamically specializing the affinity policy and re-binding threads for a given block.

Memory Manager

Allocation of buffers is performed by a memory manager in the CAPE runtime. The memory manager integrates an approach from previous work[21]. Previous work investigated an expansion of the memory-allocation routines of the NumPy library. The general idea is to apply a buffer-reuse scheme for large memory allocations. The scheme named victim-caching was shown to provide a speedup of up to 2.29, on average 1.32 across a benchmark suite of 15 applications. Most importantly, at no time did the victim cache scheme perform worse than unmodified NumPy. The results of previous work motivated the implementation of an equivalent victim-cache scheme for allocation of buffers in CAPE.

In addition to the victim-cache scheme the memory manager allocates page-aligned memory via `mmap` on the host. The memory manager is responsible for all aspects of allocating buffers for array-operation operands. This also includes the allocation of buffers on accelerator devices, it is left for future work do describe this part of the memory manager.

The memory manager is invoked before and after the execution of a block function, as figure 9 illustrate.

4. Performance Study

This section describes various performance aspects of CAPE. Section 4.1 provide numbers of a set of Python/NumPy benchmarks. These numbers show relative speedup, using the regular NumPy implementation as a baseline, to the

Machine:	
Processor:	AMD Opteron 6272
Clock:	2.1 GHz
L3 Cache:	16MB
Memory:	128GB DDR3
Network:	Gigabit Ethernet
Compiler:	GCC 4.8.4
Software:	Ubuntu 14.04, Linux 3.13, Python 2.7.6, NumPy 1.8.2

Table 1: Machine Specifications

same applications but where the NumPy array operations are extracted using npbackend/Bohrium and CAPE is used for processing them. The section will thus show the speedup obtainable by the Bohrium/CAPE approach. An additional set of numbers is provided in the section, namely using Bohrium/CAPE approach but disabling optimizations for array-operation compositioning. Those numbers will show what is gained by the Bohrium/CAPE approach in contrast to simply implementing a parallel version of the regular NumPy library.

Section 4.2 provide relative speedup of a subset of the benchmarks in 4.1. The baseline used for these numbers is a serial implementation in C. For comparison parallel implementations in C/C++ are also provided. These numbers thus provide data for comparative study of how far, or how close the Bohrium/CAPE approach is to handwritten low-level implementations.

Environment

The benchmark implementations are part of an OpenSource benchmark tool and suite named Benchpress. The source code for the implementations and the Benchpress tool is available online at `http://benchpress.readthedocs.org/`. For reproducibility, the exact version used can be obtained from the source code repository at `https://github.com/bh107/benchpress.git` revision `0aa2942`.

The implementation of CAPE, BXX, Bohrium, and npbackend is similarly available `https://github.com/bh107/bohrium.git`, the numbers produced used revision `b4d3586`.

Source code including the raw numbers from the performance study is also available at the University of Copenhagen Electronic Research Data Archive as a public archive with id `YXJjaGl2ZS0xSWhQSmU=`, `http://www.erda.dk/public/archives/YXJjaGl2ZS0xSWhQSmU=/published-archive.html`.

Each benchmark was executed five times and the elapsed wall-clock time was recorded. The best and the worst result, that is, the one consuming the least and the most time were discarded, and an average of the remaining three samples used for computing relative numbers.

Table 1 provide information about the machine on which the benchmarks were executed.

4.1. Python/NumPy

Table 2 provide numbers of the performance gained from the Bohrium/-CAPE approach. The numbers show speedup using the regular NumPy library as a baseline, the raw samples of elapsed wall-clock time can be inspected in

	CAPE-AC		CAPE	
Threads	1	32	1	32
Synthetic Inplace Update	1.4	8.6	14.1	236.9
Synthetic Stream Ones	1.5	9.7	10.0	137.3
Synthetic Stream Range	0.9	8.0	1.8	45.5
Synthetic Stream Random	1.0	18.0	1.0	29.8
1D Stencil	0.7	8.5	1.1	21.8
2D Stencil	0.6	8.8	3.6	53.9
3D Stencil	0.8	11.0	1.8	48.6
27 Point Stencil	1.0	5.5	1.2	13.6
Jacobi	1.4	10.3	3.9	76.2
Gauss Elimination	0.7	1.6	2.1	3.8
LU Factorization	0.7	1.6	1.9	3.4
Leibnitz PI	1.0	6.1	2.2	48.3
Monte Carlo PI	1.0	17.7	1.0	30.1
Matrix Multiplication	0.9	10.1	2.1	47.3
Rosenbrock	1.5	9.9	12.5	169.4
Black Scholes	3.4	27.4	6.0	84.9
Game of Life v1	1.2	8.6	1.5	32.2
Game of Life v2	1.2	8.0	2.2	39.6
Heat Equation	1.3	9.1	4.4	41.0
Lattice Boltzmann 3D	0.8	4.0	0.8	5.3
NBody	4.7	17.2	5.1	23.6
NBody Nice	1.2	7.3	0.4	8.1
SOR	1.4	8.3	1.9	20.1
Shallow Water	1.5	9.2	6.9	62.2
Water-Ice Simulation	0.8	5.7	1.6	10.9
kNN Naive 1	0.7	10.2	1.0	26.1

Table 2: Benchmarks results, regular NumPy library used as baseline.

the file `numpy.txt` in the frozen data archive. Two different configurations of CAPE are used.

The label CAPE denotes the default configuration. In the default configuration all safe optimizations are applied. This includes the victim-cache buffer allocation scheme, bytecode-transformations, and most importantly the array-operation composition and array-contraction optimization.

The label CAPE-AC denotes a modification of the default configuration, specifically the array-operation composition and array-contraction optimizations are disabled.

The table additionally shows numbers for these two configurations executing using a single thread and 32 threads.

Discussion

On the majority of benchmarks, CAPE can manage load distribution for parallelization, avoid materialization of intermediate results, and apply runtime-value optimizations such as transforming operations into equivalent, but more efficient to compute, expressions. CAPE is thereby able to deliver considerable performance improvements over the regular NumPy library.

A key contributor to the obtained performance improvement is the ability of CAPE to avoid materialization of arrays for intermediate results. One can compare which is most efficient: array-contraction without parallelization (CAPE and a single threads) or without array-contraction but with parallelization (CAPE-AC and 32 threads). If only a single optimization is applied then parallelization is on the most part favorable when considing elapsed wall-clock time. However, for a few benchmarks a single thread is able to outperform 32 threads thanks to array-contraction.

Only considering parallelization, without array-contraction, severily limits scalability as the materialization of intermediate results makes any benchmark inherently memory bound. The symbiosis between parallization and array-contraction is what enables efficient utilization of the hardware in these benchmarks. As array-contraction reduce pressure on memory-bandwidth, scalability improves.

Array-operation composition and array-contraction are key drivers for exploiting locality, reducing memory-bandwidth pressure and they consistently yield improvements on throughput. The last statement is, however, contradicted by the Nbody Nice benchmark. The exact cause of the performance penalty is left for future work, current exploration has revealed that using certain commercial compilers does not exhibit the same behavior. Indicators seem to point toward pointer runtime alias checks but experiments are currently inconclusive.

4.2. C/C++

Table 3 provide numbers of the performance gained from the Bohrium/-CAPE approach in relation to handwritten implementations in C/C++. The numbers show speedup using a serial implementation of each benchmark in C as a baseline. The raw samples of elapsed wall-clock time can be inspected in the file `c_cpp.txt` in the frozen data archive. Numbers are provided for two

	C++ / OpenMP		NumPy / CAPE		C++ / CAPE	
Threads	1	32	1	32	1	32
Black Scholes +O	0.9	29.1	4.8	67.3	4.9	118.9
Black Scholes –O	0.9	29.1	1.1	26.1	1.1	31.7
Heat Equation	0.6	7.1	0.7	7.0	0.8	7.6
Leibnitz PI	1.0	22.6	0.6	14.6	0.6	15.2
Monte Carlo PI	1.0	29.8	1.0	27.8	0.9	28.2
Mxmul	1.0	9.5	1.0	14.9	1.1	15.1
Rosenbrock	1.0	21.0	1.2	15.8	1.2	21.4
Shallow Water	0.5	9.1	0.7	6.6	0.7	10.9

Table 3: Benchmarks results, serial C implementation used as baseline.

different Bohrium language bridges, Python/NumPy and C++/BXX, using the default configuration of CAPE.

The C baseline, the parallel C++ implementations, and the CAPE JIT-compiler uses the same version of the GNU compiler collection as well as the same optimization flags, specifically: `-O3 -march=native`. The serial C implemention and the CAPE JIT-compiler additionally used `-std=c99` and the C++ implementations used `-std=c++11`. The CAPE JIT-compiler and the C++ implementations additonally used `-fopenmp`.

The label C++/OpenMP denotes the handwritten implementation of the benchmark in C++ using OpenMP for parallelization, thread binding was managed via `GOMP_CPU_AFFINITY=0-31`.

The label NumPy/CAPE denotes that the benchmark was implemented in Python/NumPy and array processing delegated to CAPE via the npbackend language bridge.

The label C++/CAPE denotes that the benchmark was implemented in C++ and array processing delegated to CAPE via the BXX language bridge.

The table additionally shows numbers for these two configurations executing using a single thread and 32 threads.

Discussion

The C implementation of the Black Scholes benchmark is compute-bound as the C++/OpenMP implementation show by achieving a near perfect linear speedup using 32 threads. The numbers reported by the Python/NumPy and C++ implementations using CAPE obtain super-linear speedup of $\times 67.3$ and $\times 118.9$ using 32 threads and a speedup of about $\times 4.8$ using a single thread/core.

How the C++/CAPE and NumPy/CAPE is able to achieve such results requires a good explanation. In the case of the Black Scholes benchmark, the reason is quite simple. CAPE performs a runtime-value optimization of the math-function `pow`. The function is the largest hot-spot in the benchmark and CAPE transforms it to sequences of multiplications which are vastly faster to compute. For comparison results are also provided where the `pow` optimization is switched off.

Comparing the results from the two language bridges NumPy and BXX with 32 threads it is clear that the BXX language bridge consistently provide better results than the NumPy bridge. Once the language bridge has mapped operations, within the host language, to Bohrium bytecode then the knowledge of where the bytecode originated from is lost. There is no special treatment of bytecode from different language bridges. However, there is variation in how bytecode is created in the host language.

The measurable difference is related to how CAPE constructs blocks and does array-contraction/array-operation composition. The algorithm used to construct blocks and annotate `KP_CONTRACTABLE` operands is sensitive to the order of instructions in the bytecode program. The BXX language bridge emits data management bytecodes immediatly after the last array-operation bytecode, that uses a given array, is emitted. This is advantagous for the block-construction algorithm. The Python/NumPy bridge emits data management bytecodes when variables exit scope or are for other reasons garbage collected. Data management bytecodes might not be immediately emitted after the last use. This is disadvantagous for the block-construction algorithm. As a consequence, the NumPy language bridge loses some opportunities for adding instructions to blocks and to annotating operands for contraction (`KP_CONTRACTABLE`). Which leads to worse scalability due to increased memory bandwidth pressure. Future work will explore means of mitigating such issues.

5. Future Work

An area of continued investigation is the exploitation of runtime values for optimization. In this context, the information used for constructing Blocks is investigated. The current implementation does a single pass over the program and adds tacs into a block as long as adding the tac does not break a set of rules. If the rules are broken, the block is ended, and a new block created. This naive approach is cheap to compute at runtime and fairly good at producing blocks that allow for composing array operations. However, it is vulnerable to the order of instructions. An approach is being investigated which builds a graph of data dependencies between instructions. The data dependencies can then be used to drive the construction of blocks, instead of the instruction order.

For CAPE to fully utilize accelerator architectures such as GPUs and Intel MIC two main areas needs expansion: the code generator and the runtime. The code generator must emit loop constructs with OpenACC directives for GPUs and parallel sections/loop constructs with LEO/OpenMP directives for MICs. Additionally the CAPE runtime must manage memory on the accelerator device. Currently explored is a runtime extension using a simple `alloc`, `free`, `push`, and `pull` interface. The interface abstracts the details of LEO `#pragma offload_transfer` and OpenACC routines such as `acc_[create|delete|update_device|update_self]`. The extension manages relations between host pointers and device pointers, knows the state of buffers on host and device, and focuses on data persistence on the device to minimize data transfers between host and device.

The performance study revealed two benchmarks for which CAPE did not manage load distribution well, namely Gauss Elimination and LU Factorization. Those benchmarks execute well-known operations from linear algebra for which highly tuned libraries exist. Future work will explore the integration of such libraries without losing opportunities for array operation composition. Bohrium and CAPE already support third party library integration, although, the approach is in an explicit form. That is, the language bridge must send a bytecode extension opcode signaling a mapping between the extension opcode and library function. Future work in this area will explore transparent detection of bytecode sequences that are equivalent to well-known highly specialized functions and perform the mapping automatically. This will let CAPE remain responsible for the processing of general array operations and detecting specific functions while delegating the responsibility of processing specific functions to a highly tuned library.

6. Related Work

Intel Array Building Blocks[22], a retargetable dynamic compilation framework is closely related in its goal of closing the high-productivity and high-performance gap. Its design is also similar consisting of a dynamic compilation of high-level array operations to low-level code. However, ArrBB has been discontinued and replaced with the slightly more explicit and C/C++ only approach of Intel Cilk Plus[23] and Intel Threading Building Blocks.

Other approaches providing a C-based interface include StarPU, which in design also seek to address the challenge of performance portability, that is removing the details of the executing hardware from the programmer. StarPU provide extensions to gcc and focus on supporting c-language family and traditional HPC languages.

The work in this paper focuses on a transparent approach where the programmer is only provided with the array operations and sequential semantics. Bohrium/CAPE is as such targeted towards applications written in non-traditional HPC languages, Python/NumPy, CIL, and C++.

The use of StarPU tasks is somewhat equivalent to a CAPE Block. A key difference is that CAPE Blocks are generated by the runtime based on a sequence of array operations, using different codegenerators based on the required target. Instead the StarPU tasks are user-defined, that is, the user writes the different implementations. A wealth of language extensions for expressing parallelism in C exists, including OpenMP, OpenACC, Intel LEO, StarPU, StarSS, to name a few.

Bohrium and CAPE address a slightly different audience and intent which is to provide the machinery to support high-level array abstractions with serial semantics which execute efficiently on target hardware. The importance of the CAPE c-interface is for interoperability and ABI-compability with other low-level runtimes as well as the integration with existing languages.

Very closely related to this work the work on Transparent GPU Execution of NumPy Application[24]. The main difference being the code-generator target and runtime. The work in[24] uses an OpenCL based codegenerator and runtime for targeting GPUs. The main difference being the code-generator target. Where the work on CAPE comes from previous work on Bohrium targeting multi-core CPUs, emitting C99 code using parallel directives and runtimes instead of OpenCL.

7. Conclusion

This paper introduced CAPE, a C-targeting Array Processing Engine, which manages the concerns of optimizing and parallelizing the execution of array operations. CAPE enables automatic mapping of array operations to specific architectures through the use of the language bridges/integrations provided by Bohrium.

A performance study showed super-linear speedup on a set of Python/NumPy benchmarks, executed on a 32 core processor with NUMA architecture. The main contributing factors to the obtained speedup are the composition of array-oriented expressions that enable array-contraction, that is, avoid the materialization of arrays that store intermediate results. The second performance enhancing feature is the NUMA-aware code generator and runtime that enables efficient parallelization of array expressions on this architecture.

The performance study also compared the performance of CAPE to seven benchmarks handwritten in C, and C++ using OpenMP. Results showed that CAPE used via the Bohrium C++ library BXX, was able to match the performance of the handwritten parallel C++ implementation on five benchmarks and outperform the hand-written code on two benchmarks.

Additionally, results showed that CAPE used via the Python language bridge was able to match the performance of the hand-written parallel C++ implementation on three out of seven benchmarks. It underperformed 30-50 percent on two benchmarks and outperformed the hand-written code on two benchmarks.

References

[1] L. Dagum, R. Menon, Openmp: an industry standard api for shared-memory programming, Computational Science & Engineering, IEEE 5 (1) (1998) 46–55.

[2] C. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, R. Murty, F. Chinchilla, R. Deodhar, R. McGuire, Offload compiler runtime for the intel xeon phi coprocessor, in: Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International, 2013, pp. 1213–1225. doi:10.1109/IPDPSW.2013.251.

[3] C. Enterprise, C. Inc., NVIDIA, the Portland Group, The OpenACC Application Programming Interface (v2.0), OpenACC-Standard.org (June 2013).

[4] the Khronos group, Sycl: C++ single-source heterogeneous programming for opencl, `https://www.khronos.org/sycl` (2014).

[5] S. Van Der Walt, S. Colbert, G. Varoquaux, The numpy array: a structure for efficient numerical computation, Computing in Science & Engineering 13 (2) (2011) 22–30.

[6] Matlab Programming Environment, `http://www.mathworks.com/products/matlab/`, [Online; accessed March 13th 2015].

[7] R. Ihaka, R. Gentleman, R: a language for data analysis and graphics, Journal of computational and graphical statistics 5 (3) (1996) 299–314.

[8] J. W. Eaton, D. Bateman, S. Hauberg, Gnu octave, Network thoery, 1997.

[9] K. P. Bowman, An Introduction to Programming with IDL: Interactive data language, Academic Press, 2006.

[10] F. Perez, B. E. Granger, Ipython: a system for interactive scientific computing, Computing in Science & Engineering 9 (3) (2007) 21–29.

[11] E. Jones, T. Oliphant, P. Peterson, Scipy: Open source scientific tools for python, http://www. scipy. org/.

[12] A. Klckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation, Parallel Computing 38 (3) (2012) 157 – 174. `doi:10.1016/j.parco.2011.09.001`.

[13] M. Klemm, J. Enkovaara, pymic: A python offload module for the intel r xeon phi tm coprocessor.

[14] J. D. McCalpin, Stream: Sustainable memory bandwidth in high performance computers, Tech. rep., University of Virginia, Charlottesville, Virginia, a continually updated technical report. http://www.cs.virginia.edu/stream/ (1991-2007).
URL `http://www.cs.virginia.edu/stream/`

[15] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, B. Vinter, Bohrium: a Virtual Machine Approach to Portable Parallelism, in: Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International, IEEE, 2014, pp. 312–321.

[16] L. S. Blackford, ScaLAPACK, in: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing 96 Supercomputing 96, 1996, p. 5. `doi:{10.1145/369028.369038}`.

[17] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, Separating NumPy API from Implementation, in: 5th Workshop on Python for High Performance and Scientific Computing (PyHPC'14), 2014.

[18] K. Skovhede, B. Vinter, NumCIL: Numeric operations in the Common Intermediate Language, Journal of Next Generation Information Technology 4 (1).

[19] K. Skovhede, S. A. F. Lund, Numcil and bohrium: High productivity and high performance, in: Proceedings of the 11th International Conference on Parallel Processing and Applied Mathematics (PPAM), LNCS, Springer, 2015.

[20] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications, in: IEEE (Ed.), PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, Pisa, Italy, 2010. `doi:10.1109/PDP.2010.67`. URL `https://hal.inria.fr/inria-00429889`

[21] S. A. F. Lund, K. Skovhede, M. R. B. Kristensen, B. Vinter, Doubling the Performance of Python/NumPy with less than 100 SLOC, in: 4th Workshop on Python for High Performance and Scientific Computing (PyHPC'13), 2013.

[22] Intel array building blocks virtual machine, `http://software.intel.com/sites/default/files/m/b/6/a/arbb_vm.pdf`, [Online; accessed 12 March 2013].

[23] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. Zhou, Cilk: An efficient multithreaded runtime system, Vol. 30, ACM, 1995.

[24] T. Blum, M. R. B. Kristensen, B. Vinter, Transparent GPU Execution of NumPy Applications, in: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2014 IEEE 28th International, IEEE, 2014.

6.2 cphVB: A System for Automated Runtime Optimization and Parallelization of Vectorized Applications

cphVB: A System for Automated Runtime Optimization and Parallelization of Vectorized Applications

Mads Ruben Burgdorff Kristensen[∗†], Simon Andreas Frimann Lund[†], Troels Blum[†], Brian Vinter[†]

Abstract—Modern processor architectures, in addition to having still more cores, also require still more consideration to memory-layout in order to run at full capacity. The usefulness of most languages is deprecating as their abstractions, structures or objects are hard to map onto modern processor architectures efficiently.

The work in this paper introduces a new abstract machine framework, cphVB, that enables vector oriented high-level programming languages to map onto a broad range of architectures efficiently. The idea is to close the gap between high-level languages and hardware optimized low-level implementations. By translating high-level vector operations into an intermediate vector bytecode, cphVB enables specialized vector engines to efficiently execute the vector operations.

The primary success parameters are to maintain a complete abstraction from low-level details and to provide efficient code execution across different, modern, processors. We evaluate the presented design through a setup that targets multi-core CPU architectures. We evaluate the performance of the implementation using Python implementations of well-known algorithms: a jacobi solver, a kNN search, a shallow water simulation and a synthetic stencil simulation. All demonstrate good performance.

Index Terms—runtime optimization, high-performance, high-productivity

Introduction

Obtaining high performance from today's computing environments requires both a deep and broad working knowledge on computer architecture, communication paradigms and programming interfaces. Today's computing environments are highly heterogeneous consisting of a mixture of CPUs, GPUs, FPGAs and DSPs orchestrated in a wealth of architectures and lastly connected in numerous ways.

Utilizing this broad range of architectures manually requires programming specialists and is a very time-consuming task – time and specialization a scientific researcher typically does not have. A high-productivity language that allows rapid prototyping and still enables efficient utilization of a broad range of architectures is clearly preferable. There exist high-productivity language and libraries that automatically utilize parallel architectures [Kri10], [Dav04], [New11]. They are however still few in numbers and have one problem in common. They are closely coupled to both the front-end, i.e. programming language and IDE, and the back-end, i.e. computing device, which makes them interesting only to the few using the exact combination of front and back-end.

A tight coupling between front-end technology and back-end presents another problem; the usefulness of the developed program expires as soon as the back-end does. With the rapid development of hardware architectures the time spend on implementing optimized programs for specific hardware, is lost as soon as the hardware product expires.

In this paper, we present a novel approach to the problem of closing the gap between high-productivity languages and parallel architectures, which allows a high degree of modularity and reusability. The approach involves creating a framework, cphVB[*] (Copenhagen Vector Bytecode). cphVB defines a clear and easy to understand intermediate bytecode language and provides a runtime environment for executing the bytecode. cphVB also contains a protocol to govern the safe, and efficient exchange, creation, and destruction of model data.

cphVB provides a retargetable framework in which the user can write programs utilizing whichever cphVB supported programming interface they prefer and run the program on their own workstation while doing prototyping, such as testing correctness and functionality of their programs. Users can then deploy exactly the same program in a more powerful execution environment without changing a single line of code and thus effectively solve greater problem sets.

The rest of the paper is organized as follows. In Section *Programming Model.* we describe the programming model supported by cphVB. The section following gives a brief description of *Numerical Python*, which is the first programming interface that fully supports cphVB. Sections *Design* and *Implementation* cover the overall cphVB design and an implementation of it. In Section *Performance Study*, we conduct an evaluation of the implementation. Finally, in Section *Future Work* and *Conclusion* we discuss future work and conclude.

∗ *Corresponding author: madsbk@nbi.dk*
† *University of Copenhagen*

*. Open Source Project - Website: http://cphvb.bitbucket.org.

Related Work

The key motivation for cphVB is to provide a framework for the utilization of heterogeneous computing systems with the goal of obtaining high-performance, high-productivity and high-portability (HP^3). Systems such as pyOpenCL/pyCUDA [Klo09] provides a direct mapping from front-end language to the optimization target. In this case, providing the user with direct access to the low-level systems OpenCL [Khr10] and CUDA [Nvi10] from the high-level language Python [Ros10]. The work in [Klo09] enables the user to write a low-level implementation in a high-productivity language. The goal is similar to cphVB – the approach however is entirely different. cphVB provides a means to hide low-level target specific code behind a programming model and providing a framework and runtime environment to support it.

Intel Math Kernel Library [Int08] is in this regard more comparable to cphVB. Intel MKL is a programming library providing utilization of multiple targets ranging from a single-core CPU to a multi-core shared memory CPU and even to a cluster of computers all through the same programming API. However, cphVB is not only a programming library it is a runtime system providing support for a vector oriented programming model. The programming model is well-known from high-productivity languages such as MATLAB [Mat10], [Rrr11], [Idl00], GNU Octave [Oct97] and Numerical Python (NumPy) [Oli07] to name a few.

cphVB is more closely related to the work described in [Gar10], here a compilation framework is provided for execution in a hybrid environment consisting of both CPUs and GPUs. Their framework uses a Python/NumPy based front-end that uses Python decorators as hints to do selective optimizations. cphVB similarly provides a NumPy based front-end and equivalently does selective optimizations. However, cphVB uses a slightly less obtrusive approach; program selection hints are sent from the front-end via the NumPy-bridge. This approach provides the advantage that any existing NumPy program can run unaltered and take advantage of cphVB without changing a single line of code. Whereas unPython requires the user to manually modify the source code by applying hints in a manner similar to that of OpenMP [Pas05]. This non-obtrusive design at the source level is to the author's knowledge novel.

Microsoft Accelerator [Dav04] introduces ParallelArray, which is similar to the utilization of the NumPy arrays in cphVB but there are strict limitations to the utilization of ParallelArrays. ParallelArrays does not allow the use of direct indexing, which means that the user must copy a ParallelArray into a conventional array before indexing. cphVB instead allows indexed operations and additionally supports **array-views**, which are array-aliases that provide multiple ways to access the same chunk of allocated memory. Thus, the data structure in cphVB is highly flexible and provides elegant programming solutions for a broad range of numerical algorithms. Intel provides a similar approach called Intel Array Building Blocks (ArBB) [New11] that provides retargetability and dynamic compilation. It is thereby possible to utilize heterogeneous architectures from within standard C++. The retargetability aspect of Intel ArBB is represented in cphVB as a plain and simple configuration file that define the cphVB runtime environment. Intel ArBB provides a high performance library that utilizes a heterogeneous environment and hides the low-level details behind a vector oriented programming model similar to cphVB. However, ArBB only provides access to the programming model via C++ whereas cphVB is not biased towards any one specific front-end language.

On multiple points cphVB is closely related in functionality and goals to the SEJITS [Cat09] project. SEJITS takes a different approach towards the front-end and programming model. SEJITS provides a rich set of computational kernels in a high-productivity language such as Python or Ruby. These kernels are then specialized towards an optimality criteria. This approach has shown to provide performance that at times out-performs even hand-written specialized code towards a given architecture. Being able to construct computational kernels is a core issue in data-parallel programming.

The programming model in cphVB does not provide this kernel methodology. cphVB has a strong NumPy heritage which also shows in the programming model. The advantage is easy adaptability of the cphVB programming model for users of NumPy, Matlab, Octave and R. The cphVB programming model is not a stranger to computational kernels – cphVB deduce computational kernels at runtime by inspecting the vector bytecode generated by the Bridge.

cphVB provides in this sense a virtual machine optimized for execution of vector operations, previous work [And08] was based on a complete virtual machine for generic execution whereas cphVB provides an optimized subset.

Numerical Python

Before describing the design of cphVB, we will briefly go through Numerical Python (NumPy) [Oli07]. Numerical Python heavily influenced many design decisions in cphVB – it also uses a vector oriented programming model as cphVB.

NumPy is a library for numerical operations in Python, which is implemented in the C programming language. NumPy provides the programmer with a multidimensional array object and a whole range of supported array operations. By using the array operations, NumPy takes advantage of efficient C-implementations while retaining the high abstraction level of Python.

NumPy uses an array syntax that is based on the Python list syntax. The arrays are indexed positionally, 0 through length – 1, where negative indexes is used for indexing in the reversed order. Like the list syntax in Python, it is possible to index multiple elements. All indexing that represents more than one element returns a view of the elements rather than a new copy of the elements. It is this view semantic that makes it possible to implement a stencil operation as illustrated in Figure 1 and demonstrated in the code example below. In order to force a real array copy rather than a new array reference NumPy provides the "copy" method.

In the rest of this paper, we define the **array-base** as the originally allocated array that lies contiguously in memory. In addition, we will define the **array-view** as a view of the

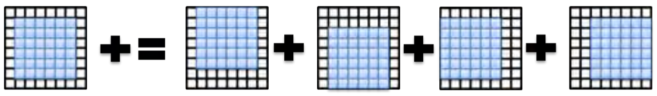


Fig. 1: *Matrix expression of a simple 5-point stencil computation example. See line eight in the code example, for the Python expression.*

elements in an **array-base**. An **array-view** is usually a subset of the elements in the **array-base** or a re-ordering such as the reverse order of the elements or a combination.

```
center = full[1:-1, 1:-1]
up     = full[0:-2, 1:-1]
down   = full[2:  , 1:-1]
left   = full[1:-1, 0:-2]
right  = full[1:-1, 2:  ]
while epsilon < delta:
    work[:] = center
    work += 0.2 * (up+down+left+right)
    center[:] = work
```

Target Programming Model

To hide the complexities of obtaining high-performance from a heterogeneous environment any given system must provide a meaningful high-level abstraction. This can be realized in the form of domain specific languages, embedded languages, language extensions, libraries, APIs etc. Such an abstraction serves two purposes: 1) It must provide meaning for the end-user such that the goal of high-productivity can be met with satisfaction. 2) It must provide an abstraction that consists of a sufficient amount of information for the system to optimize its utilization.

cphVB is not biased towards any specific choice of abstraction or front-end technology as long as it is compatible with a vector oriented programming model. This provides means to use cphVB in functional programming languages, provide a front-end with a strict mathematic notation such as APL [Apl00] or a more relaxed syntax such as MATLAB.

The vector oriented programming model encourages expressing programs in the form of high-level array operations, e.g. by expressing the addition of two arrays using one high-level function instead of computing each element individually. The NumPy application in the code example above figure 1 is a good example of using the vector oriented programming model.

Design of cphVB

The key contribution in this paper is a framework, cphVB, that support a vector oriented programming model. The idea of cphVB is to provide the mechanics to seamlessly couple a programming language or library with an architecture-specific implementation of vectorized operations.

cphVB consists of a number of components that communicate using a simple protocol. Components are allowed to be architecture-specific but they are all interchangeable since all uses the same communication protocol. The idea is to make it possible to combine components in a setup that perfectly match a specific execution environment. cphVB consist of the following components:

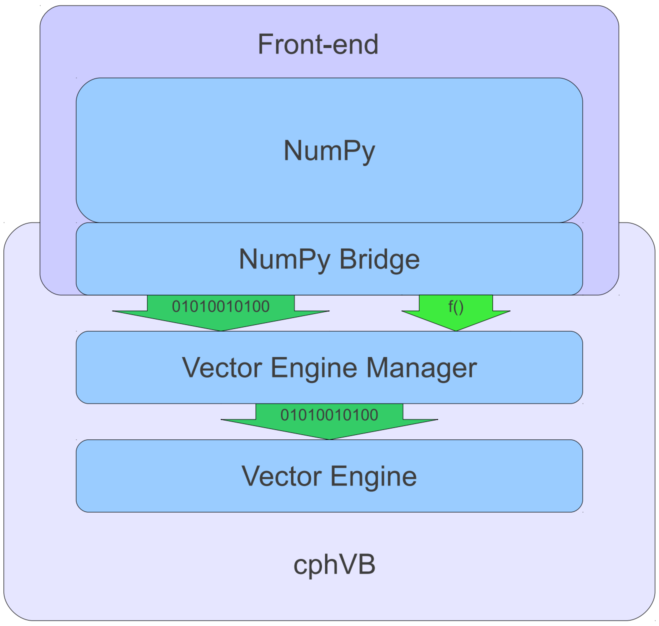


Fig. 2: *cphVB design idea.*

Programming Interface
: The programming language or library exposed to the user. cphVB was initially meant as a computational back-end for the Python library NumPy, but we have generalized cphVB to potential support all kinds of languages and libraries. Still, cphVB has design decisions that are influenced by NumPy and its representation of vectors/matrices.

Bridge
: The role of the Bridge is to integrate cphVB into existing languages and libraries. The Bridge generates the cphVB bytecode that corresponds to the user-code.

Vector Engine
: The Vector Engine is the architecture-specific implementation that executes cphVB bytecode.

Vector Engine Manager
: The Vector Engine Manager manages data location and ownership of vectors. It also manages the distribution of computing jobs between potentially several Vector Engines, hence the name.

An overview of the design can be seen in Figure 2.

Configuration

To make cphVB as flexible a framework as possible, we manage the setup of all the components at runtime through a configuration file. The idea is that the user can change the setup of components simply by editing the configuration file before executing the user application. Additionally, the user only has to change the configuration file in order to run the application on different systems with different computational resources. The configuration file uses the ini syntax, an example is provided below.

```
# Root of the setup
[setup]
```

```
bridge = numpy
debug = true

# Bridge for NumPy
[numpy]
type = bridge
children = node

# Vector Engine Manager for a single machine
[node]
type = vem
impl = libcphvb_vem_node.so
children = mcore

# Vector Engine using TLP on shared memory
[mcore]
type = ve
impl = libcphvb_ve_mcore.so
```

This example configuration provides a setup for utilizing a shared memory machine with thread-level-parallelism (TLP) on one machine by instructing the vector engine manager to use a single multi-core TLP engine.

Bytecode

The central part of the communication between all the components in cphVB is vector bytecode. The goal with the bytecode language is to be able to express operations on multidimensional vectors. Taking inspiration from single instruction, multiple data (SIMD) instructions but adding structure to the data. This, of course, fits very well with the array operations in NumPy but is not bound nor limited to these.

We would like the bytecode to be a concept that is easy to explain and understand. It should have a simple design that is easy to implement. It should be easy and inexpensive to generate and decode. To fulfill these goals we chose a design that conceptually is an assembly language where the operands are multidimensional vectors. Furthermore, to simplify the design the assembly language should have a one-to-one mapping between instruction mnemonics and opcodes.

In the basic form, the bytecode instructions are primitive operations on data, e.g. addition, subtraction, multiplication, division, square root etc. As an example, let us look at addition. Conceptually it has the form:

```
add $d, $a, $b
```

Where `add` is the opcode for addition. After execution `$d` will contain the sum of `$a` and `$b`.

The requirement is straightforward: we need an opcode. The opcode will explicitly identify the operation to perform. Additionally the opcode will implicitly define the number of operands. Finally, we need some sort of symbolic identifiers for the operands. Keep in mind that the operands will be multidimensional arrays.

Interface

The Vector Engine and the Vector Engine Manager exposes simple API that consists of the following functions: initialization, finalization, registration of a user-defined operation and execution of a list of bytecodes. Furthermore, the Vector Engine Manager exposes a function to define new arrays.

Bridge

The Bridge is the **bridge** between the programming interface, e.g. Python/NumPy, and the Vector Engine Manager. The Bridge is the only component that is specifically implemented for the programming interface. In order to add cphVB support to a new language or library, one only has to implement the bridge component. It generates bytecode based on programming interface and sends them to the Vector Engine Manager.

Vector Engine Manager

Instead of allowing the front-end to communicate directly with the Vector Engine, we introduce a Vector Engine Manager (VEM) into the design. It is the responsibility of the VEM to manage data ownership and distribute bytecode instructions to several Vector Engines. It is also the ideal place to implement code optimization, which will benefit all Vector Engines.

To facilitate late allocation, and early release of resources, the VEM handles instantiation and destruction of arrays. At array creation only the meta data is actually created. Often arrays are created with structured data (e.g. random, constants), with no data at all (e.g. empty), or as a result of calculation. In any case it saves, potentially several, memory copies to delay the actual memory allocation. Typically, array data will exist on the computing device exclusively.

In order to minimize data copying we introduce a data ownership scheme. It keeps track of which components in cphVB that needs to access a given array. The goal is to allow the system to have several copies of the same data while ensuring that they are in synchronization. We base the data ownership scheme on two instructions, **sync** and **discard**:

Sync
: is issued by the bridge to request read access to a data object. This means that when acknowledging a **sync** request, the copy existing in shared memory needs to be the most resent copy.

Discard
: is used to signal that the copy in shared memory has been updated and all other copies are now invalid. Normally used by the bridge to upgrading a read access to a write access.

The cphVB components follow the following four rules when implementing the data ownership scheme:

1. The Bridge will always ask the Vector Engine Manager for access to a given data object. It will send a **sync** request for read access, followed by a **release** request for write access. The Bridge will not keep track of ownership itself.
2. A Vector Engine can assume that it has write access to all of the output parameters that are referenced in the instructions it receives. Likewise, it can assume read access on all input parameters.
3. A Vector Engine is free to manage its own copies of arrays and implement its own scheme to minimize data copying. It just needs to copy modified data back to share memory when receiving a **sync** instruction and delete all local copies when receiving a **discard** instruction.

4. The Vector Engine Manager keeps track of array ownership for all its children. The owner of an array has full (i.e. write) access. When the parent component of the Vector Engine Manager, normally the Bridge, request access to an array, the Vector Engine Manager will forward the request to the relevant child component. The Vector Engine Manager never accesses the array itself.

Additionally, the Vector Engine Manager needs the capability to handle multiple children components. In order to maximize parallelism the Vector Engine Manager can distribute workload and array data between its children components.

Vector Engine

Though the Vector Engine is the most complex component of cphVB, it has a very simple and a clearly defined role. It has to execute all instructions it receives in a manner that obey the serialization dependencies between instructions. Finally, it has to ensure that the rest of the system has access to the results as governed by the rules of the **sync**, **release**, and **discard** instructions.

Implementation of cphVB

In order to demonstrate our cphVB design we have implemented a basic cphVB setup. This concretization of cphVB is by no means exhaustive. The setup is targeting the NumPy library executing on a single machine with multiple CPU-cores. In this section, we will describe the implementation of each component in the cphVB setup – the Bridge, the Vector Engine Manager, and the Vector Engine. The cphVB design rules (Sec. Design) govern the interplay between the components.

Bridge

The role of the Bridge is to introduce cphVB into an already existing project. In this specific case NumPy, but could just as well be `R` or any other language/tool that works primarily on vectorizable operations on large data objects.

It is the responsibility of the Bridge to generate cphVB instructions on basis of the Python program that is being run. The NumPy Bridge is an extension of NumPy version 1.6. It uses hooks to divert function call where the program access cphVB enabled NumPy arrays. The hooks will translate a given function into its corresponding cphVB bytecode when possible. When it is not possible, the hooks will feed the function call back into NumPy and thereby forcing NumPy to handle the function call itself.

The Bridge operates with two address spaces for arrays: the cphVB space and the NumPy space. All arrays starts in the NumPy space as a default. The original NumPy implementation handles these arrays and all operations using them. It is possible to assign an array to the cphVB space explicitly by using an optional cphVB parameter in array creation functions such as `empty` and `random`. The cphVB bridge implementation handles these arrays and all operations using them.

In two circumstances, it is possible for an array to transfer from one address space to the other implicitly at runtime.

1. When an operation accesses an array in the cphVB address space but it is not possible for the bridge to translate the operation into cphVB code. In this case, the bridge will synchronize and move the data to the NumPy address space. For efficiency no data is actually copied instead the bridge uses the `mremap`[†] function to re-map the relevant memory pages.
2. When an operations access arrays in different address spaces the Bridge will transfer the arrays in the NumPy space to the cphVB space. Afterwards, the bridge will translate the operation into bytecode that cphVB can execute.

In order to detect direct access to arrays in the cphVB address space by the user, the original NumPy implementation, a Python library or any other external source, the bridge protects the memory of arrays that are in the cphVB address space using `mprotect`[‡]. Because of this memory protection, subsequently accesses to the memory will trigger a segmentation fault. The Bridge can then handle this kernel signal by transferring the array to the NumPy address space and cancel the segmentation fault. This technique makes it possible for the Bridge to support all valid Python/NumPy application since it can always fallback to the original NumPy implementation.

In order to gather greatest possible information at runtime, the Bridge will collect a batch of instructions rather than executing one instruction at a time. The Bridge will keep recording instruction until either the application reaches the end of the program or untranslatable NumPy operations forces the Bridge to move an array to the NumPy address space. When this happens, the Bridge will call the Vector Engine Manager to execute all instructions recorded in the batch.

Vector Engine Manager

The Vector Engine Manager (VEM) in our setup is very simple because it only has to handle one Vector Engine thus all operations go to the same Vector Engine. Still, the VEM creates and deletes arrays based on specification from the Bridge and handles all meta-data associated with arrays.

Vector Engine

In order to maximize the CPU cache utilization and enables parallel execution the first stage in the VE is to form a set of instructions that enables data blocking. That is, a set of instructions where all instructions can be applied on one data block completely at a time without violating data dependencies. This set of instructions will be referred to as a kernel.

The VE will form the kernel based on the batch of instructions it receives from the VEM. The VE examines each instruction sequentially and keep adding instruction to the kernel until it reaches an instruction that is not **blockable** with the rest of the kernel. In order to be blockable with the rest

Processor	Intel Core i5-2510M
Clock	2.3 GHz
Private L1 Data Cache	128 KB
Private L2 Data Cache	512 KB
Shared L3 Cache	3072 KB
Memory Bandwidth	21.3 GB/s
Memory	4GB DDR3-1333
Compiler	GCC 4.6.3

TABLE 1: *ASUS P31SD.*

of the kernel an instruction must satisfy the following two properties where A is all instructions in the kernel and N is the new instruction.

1. The input arrays of N and the output array of A do not share any data or represents precisely the same data.
2. The output array of N and the input and output arrays of A do not share any data or represents precisely the same data.

When the VE has formed a kernel, it is ready for execution. Since all instruction in a kernel supports data blocking the VE can simply assign one block of data to each CPU-core in the system and thus utilizing multiple CPU-cores. In order to maximize the CPU cache utilization the VE may divide the instructions into even more data blocks. The idea is to access data in chunks that fits in the CPU cache. The user, through an environment variable, manually configures the number of data blocks the VE will use.

Performance Study

In order to demonstrate the performance of our initial cphVB implementation and thereby the potential of the cphVB design, we will conduct some performance benchmarks using NumPy[§]. We execute the benchmark applications on ASUS P31SD with an Intel Core i5-2410M processor (Table 1).

The experiments used the three vector engines: *simple*, *score* and *mcore* and for each execution we calculate the relative speedup of cphVB compared to NumPy. We perform strong scaling experiments, in which the problem size is constant though all the executions. For each experiment, we find the block size that results in best performance and we calculate the result of each experiment using the average of three executions.

The benchmark consists of the following Python/NumPy applications. All are pure Python applications that make use of NumPy and none uses any external libraries.

- **Jacobi Solver** An implementation of an iterative jacobi solver with fixed iterations instead of numerical convergence. (Fig. 3).
- **kNN** A naive implementation of a k Nearest Neighbor search (Fig. 4).

†. The function mremap() in GNU C library 2.4 and greater.

‡. The function mprotect() in the POSIX.1-2001 standard.

- **Shallow Water** A simulation that simulates a system governed by the shallow water equations. It is a translation of a MATLAB application by Burkardt [Bur10] (Fig. 5).
- **Synthetic Stencil** A synthetic stencil simulation the code relies heavily on the slicing operations of NumPy. (Fig. 6).

Discussion

The jacobi solver shows an efficient utilization of data-blocking to an extent competing with using multiple processors. The *score* engine achieves a 1.42x speedup in comparison to NumPy (3.98*sec* to 2.8*sec*).

On the other hand, our naive implementation of the k Nearest Neighbor search is not an embarrassingly parallel problem. However, it has a time complexity of $O(n^2)$ when the number of elements and the size of the query set is n, thus the problem should be scalable. The result of our experiment is also promising – with a performance speedup of of 3.57x (5.40*sec* to 1.51*sec*) even with the two single-core engines and a speed-up of nearly 6.8x (5.40*sec* to 0.79) with the multi-core engine.

The Shallow Water simulation only has a time complexity of $O(n)$ thus it is the most memory intensive application in our benchmark. Still, cphVB manages to achieve a performance speedup of 1.52x (7.86*sec* to 5.17*sec*) due to memory-allocation optimization and 2.98x (7.86*sec* to 2.63*sec*) using the multi-core engine.

Finally, the synthetic stencil has an almost identical performance pattern as the shallow water benchmark the *score* engine does however give slightly better results than the *simple* engine. Score achieves a speedup of 1.6x (6.60*sec* to 4.09*sec*) and the *mcore* engine achieves a speedup of 3.04x (6.60*sec* to 2.17*sec*).

It is promising to observe that even most basic vector engine (*simple*) shows a speedup and in none of our benchmarks a slowdown. This leads to the promising conclusion that the memory optimizations implemented outweigh the cost of using cphVB. Adding the potential of speedup due to data-blocking motivates studying further optimizations in addition to thread-level-parallelization. The *mcore* engine does provide speedups, the speedup does however not scale with the number of cores. This result is however expected as the benchmarks are memory-intensive and the memory subsystem is therefore the bottleneck and not the number of computational cores available.

Future Work

The future goals of cphVB involves improvement in two major areas; expanding support and improving performance. Work has started on a CIL-bridge which will leverage the use of cphVB to every CIL based programming language which among others include: C#, F#, Visual C++ and VB.NET. Another project in current progress within the area of support is a C++ bridge providing a library-like interface to cphVB

§. NumPy version 1.6.1.

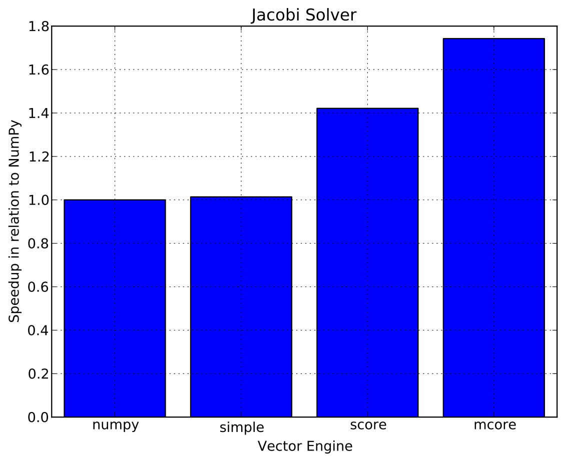


Fig. 3: *Relative speedup of the Jacobi Method. The job consists of a vector with* 7168*x*7168 *elements using four iterations.*

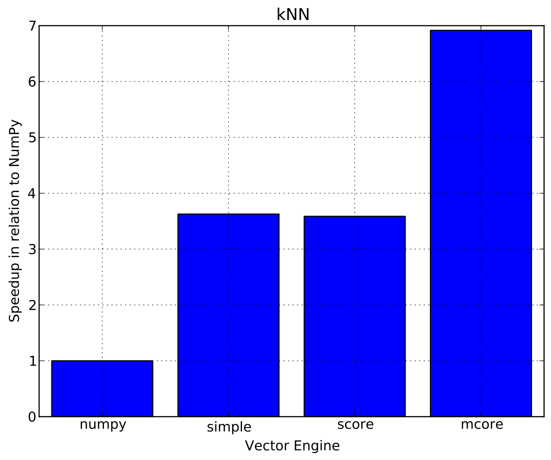


Fig. 4: *Relative speedup of the k Nearest Neighbor search. The job consists of 10.000 elements and the query set also consists of 1K elements.*

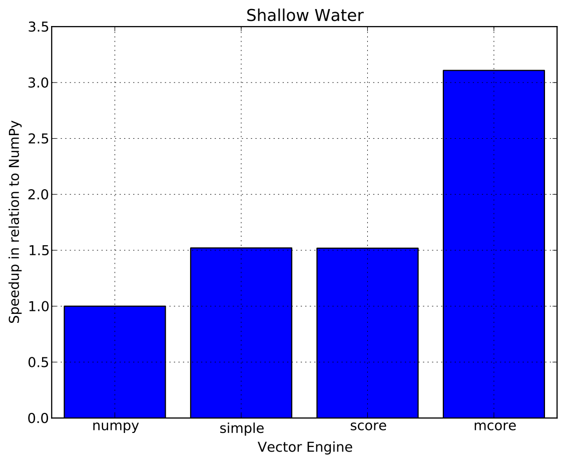


Fig. 5: *Relative speedup of the Shallow Water Equation. The job consists of 10.000 grid points that simulate 120 time steps.*

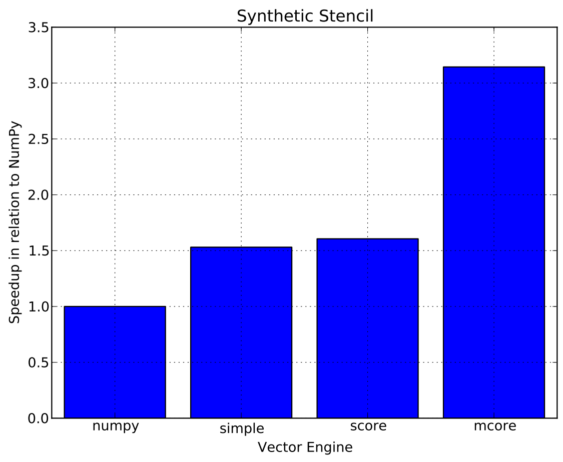


Fig. 6: *Relative speedup of the synthetic stencil code. The job consists of vector with* 10240*x*1024 *elements that simulate 10 time steps.*

using operator overloading and templates to provide a high-level interface in C++.

To improve both support and performance, work is in progress on a vector engine targeting OpenCL compatible hardware, mainly focusing on using GPU-resources to improve performance. Additionally the support for program execution using distributed memory is on progress. This functionality will be added to cphVB in the form a vector engine manager.

In terms of pure performance enhancement, cphVB will introduce JIT compilation in order to improve memory intensive applications. The current vector engine for multi-cores CPUs uses data blocking to improve cache utilization but as our experiments show then the memory intensive applications still suffer from the von Neumann bottleneck [Bac78]. By JIT compile the instruction kernels, it is possible to improve cache utilization drastically.

Conclusion

The vector oriented programming model used in cphVB provides a framework for high-performance and high-productivity. It enables the end-user to execute vectorized applications on a broad range of hardware architectures efficiently without any hardware specific knowledge. Furthermore, the cphVB design supports scalable architectures such as clusters and supercomputers. It is even possible to combine architectures in order to exploit hybrid programming where multiple levels of parallelism exist. The authors in [Kri11] demonstrate that combining shared memory and distributed memory parallelism through hybrid programming is essential in order to utilize the Blue Gene/P architecture fully.

In a case study, we demonstrate the design of cphVB by implementing a front-end for Python/NumPy that targets multi-core CPUs in a shared memory environment. The implementation executes vectorized applications in parallel without any user intervention. Thus showing that it is possible to retain the high abstraction level of Python/NumPy while fully utilizing the underlying hardware. Furthermore, the implementation demonstrates scalable performance – a k-nearest neighbor search purely written in Python/NumPy obtains a speedup of more than five compared to a native execution.

Future work will further test the cphVB design model as new front-end technologies and heterogeneous architectures are supported.

References

[Kri10] M. R. B. Kristensen and B. Vinter, *Numerical Python for Scalable Architectures*, in Fourth Conference on Partitioned Global Address Space Programming Model, PGAS{'}10. ACM, 2010. [Online]. Available: http://distnumpy.googlecode.com/files/kristensen10.pdf

[Dav04] T. David, P. Sidd, and O. Jose, *Accelerator : Using Data Parallelism to Program GPUs for General-Purpose Uses*, October. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=70250

[New11] C. J. Newburn, B. So, Z. Liu, M. Mccool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, *Intels Array Building Blocks : A Retargetable , Dynamic Compiler and Embedded Language*, Symposium A Quarterly Journal In Modern Foreign Literatures, pp. 1–12, 2011. [Online]. Available: http://software.intel.com/en-us/blogs/wordpress/wp-content/uploads/2011/03/ArBB-CGO2011-distr.pdf

[Klo09] A. Kloeckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, o and A. Fasih, *PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation*, Brain, vol. 911, no. 4, pp. 1–24, 2009. [Online]. Available: http://arxiv.org/abs/0911.3456

[Khr10] K. Opencl, W. Group, and A. Munshi, *OpenCL Specification*, ReVision, pp. 1–377, 2010. [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:OpenCL+Specification#2

[Nvi10] N. Nvidia, *NVIDIA CUDA Programming Guide 2.0*, pp. 1–111, 2010. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/32prod/toolkit/docs/CUDAProgrammingGuide.pdf

[Ros10] G. V. Rossum and F. L. Drake, *Python Tutorial*, History, vol. 42, no. 4, pp. 1–122, 2010. [Online]. Available: http://docs.python.org/tutorial/

[Int08] Intel, *Intel Math Kernel Library (MKL)*, pp. 2–4, 2008. [Online]. Available: http://software.intel.com/en-us/articles/intel-mkl/

[Mat10] MATLAB, version 7.10.0 (R2010a). Natick, Massachusetts: The MathWorks Inc., 2010.

[Rrr11] R Development Core Team, *R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing*, Vienna, Austria, 2011. [Online]. Available: http://www.r-project.org

[Idl00] B. A. Stern, *Interactive Data Language*, ASCE, 2000.

[Oct97] J. W. Eaton, *GNU Octave*, History, vol. 103, no. February, pp. 1–356, 1997. [Online]. Available: http://www.octave.org

[Oli07] T. E. Oliphant, *Python for Scientific Computing*, Computing in Science Engineering, vol. 9, no. 3, pp. 10–20, 2007. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4160250

[Gar10] R. Garg and J. N. Amaral, *Compiling Python to a hybrid execution environment*, Computing, pp. 19–30, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1735688.1735695

[Pas05] R. V. D. Pas, *An Introduction Into OpenMP*, ACM SIGARCH Computer Architecture News, vol. 34, no. 5, pp. 1–82, 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1168898

[Cat09] B. Catanzaro, S. Kamil, Y. Lee, K. Asanov'i, J. Demmel, c K. Keutzer, J. Shalf, K. Yelick, and O. Fox, *SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization*, in Proc of 1st Workshop Programmable Models for Emerging Architecture PMEA, no. UCB/EECS-2010-23, EECS Department, University of California, Berkeley. Citeseer, 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-23.html

[And08] R. Andersen and B. Vinter, *The Scientific Byte Code Virtual Machine*, in Proceedings of the 2008 International Conference on Grid Computing & Applications, GCA2008 : Las Vegas, Nevada, USA, July 14-17, 2008. CSREA Press., 2008, pp. 175–181. [Online]. Available: http://dk.migrid.org/public/doc/published_papers/sbc.pdf

[Apl00] "why apl?" [Online]. Available: http://www.sigapl.org/whyapl.htm

[Sci02] R. Pozo and B. Miller, *SciMark 2.0*, 2002. [Online]. Available: http://math.nist.gov/scimark2/

[Bur10] J. Burkardt, *Shallow Water Equations*, 2010. [Online]. Available: http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_2d/

[Bac78] J. Backus, *Can Programming be Liberated from the von Neumann Style?: A Functional Style and its Algebra of Programs*, Communications of the ACM, vol. 16, no. 8, pp. 613–641, 1978.

[Kri11] M. Kristensen, H. Happe, and B. Vinter, *Hybrid Parallel Programming for Blue Gene/P*, Scalable Computing: Practice and Experience, vol. 12, no. 2, pp. 265–274, 2011.

6.3 Doubling the Performance of Python/NumPy with less than 100 SLOC

Doubling the Performance of Python/NumPy with less than 100 SLOC

Simon A. F. Lund, Kenneth Skovhede, Mads R. B. Kristensen, and Brian Vinter
Niels Bohr Institute, University of Copenhagen, Denmark
{safl/skovhede/madsbk/vinter}@nbi.dk

Abstract—**A very simple, and outside NumPy, commonly used trick of buffer-reuse is introduced to the NumPy library to speed up the performance of scientific applications in Python/NumPy. The implementation, which we name software victim-caching, is very simple. The code itself consists of less than 100 lines of code, and took less than one day to add to NumPy, though it should be noted that the programmer was very familiar with the inner workings of NumPy. The result is an improvement of as much as 2.29 times speedup, on average 1.32 times speedup across a benchmark suite of 15 applications, and at no time did the modification perform worse than unmodified NumPy.**

I. Introduction

Python/NumPy is gaining momentum in high performance computing, often as a glue language between high performance libraries, but increasingly also with all or parts of the functionality written directly in Python/NumPy. Python/NumPy represents an easy transition from Matlab prototypes, to the extent where we observe scientists working directly in Python/NumPy since their productivity is as high as in Matlab. While Python/NumPy is still not as efficient as C++ or Fortran, which are the more common HPC languages, the productivity of the higher-level language often becomes the choice of the programmer. As a rule of thumb, we expect Python/NumPy to be approximately four to five times slower than C, and the balance in choosing a programming language is thus often a balance between faster programming or faster execution and is stands to reason that, as Python/NumPy solutions close the performance gap to compiled languages, the higher productivity language will gain further traction. In our work to improve the performance of NumPy[1] we came across a behavior which we initially attributed to our work on cache optimizations, turned out to be the effects of a far simpler scheme where by temporary array allocations in NumPy are more efficiently reused.

The amount of memory that is reserved for buffer-space is naturally defined by the user through a standard environment variable. In this work, we experiment with three fixed buffer-sizes 100, 512 and 1024 mega bytes. Programmers can experiment with different buffer-sizes, however, very large buffers rarely make an impact.

The resulting changes to NumPy, less than 100 lines in total, counted using SLOCCount[2], provides advantages over conventional NumPy from none, but never worse, to 2.29 times speedup. Our suite of 15 benchmarks has an average speedup of 1.32, and thus, with no requirement to the application programmer closes the gap to compiled languages a little further.

The rest of this paper is comprised as follows; related work since this is not a new idea outside Python, a section on the implementation details, then the benchmarks are introduced and results are presented.

II. Related Work

In computer architecture, a victim-cache is a small fully-associative cache where any evicted cache-line is stored and thus granted an extra chance for remaining in the cache, before being finally evicted[3]. At the CPU level victim-caching is particularly efficient at masking cache-line tag conflicts. Since NumPy does not have any cache, the victim cache may appear unrelated, but the idea of a fully associative cache that holds buffers a little while until they are fully evicted, is very similar.

In functional languages a similar buffer reuse scheme, copy collections, is found efficient in [4]. In this work, the buffer is very large, and numerous techniques for buffer location and replacement are considered; most of this is similar to page replacement algorithms at the operating system level.

Keeping control of buffers in relational databases is fairly closely related to maintain NumPy buffers, since relational databases also have a high locality of similar sized buffers[5] but dissimilar to NumPy, the space available for buffers is very high, and a more advanced replacement algorithm is needed since databases are multiuser systems, and the buffer patterns is thus less simple than what we can observe in NumPy.

Even though the victim cache technique itself is not related to garbage collection, the idea of memory reuse is very similar. Within a runtime with managed garbage collection, memory allocations are pooled to avoid repeated requests to the operating system[6]. While this is useful for repeated small allocations, most implementations assume that large allocations will stay in memory.

III. Software Victim-Caching

We have dubbed the adopted technique software victim-caching, since the basic functionality is very similar to victim-caching as it is known in computer architecture. The idea is very simple; when NumPy releases an array we do not release the memory immediately, but keep the buffer in a victim-cache, when NumPy issues a new array allocation we first do a lookup in the victim-cache, and if a matching array is found, it is returned rather than a new array allocation.

Different matching and eviction algorithms have been experimented with, see section III-C for further details. Note that only full allocations are returned from the victim-cache, we do not try to use partial arrays or merge arrays to find a match, or in any other way attempt conventional heap management.

The logic behind this very naÃ¯ve approach is fairly simple as well; scientific applications are most often comprised of dense loops any temporary array allocation is due to this very likely to be observed again very soon after being released. In addition, the temporary arrays that are allocated for different operations on the same user defined array are likely to be of identical dimensions as well.

A. Temporary Arrays

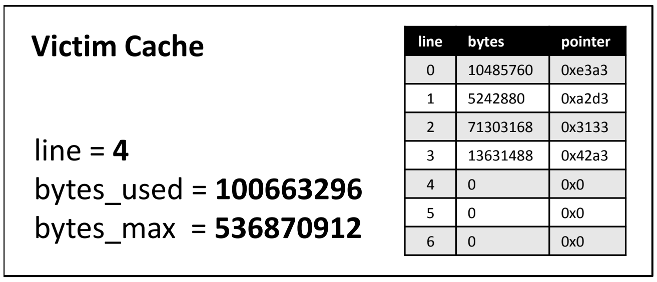
Temporary arrays are instantiated by NumPy whenever an intermediate result is needed. The general case is; an expression consisting of more than a single operator and thereby creating a complex expression. As an example, assume we wish to calculate the distance from $(0, 0)$ for a set of (X, Y) coordinates in NumPy we write:

```
distance = numpy.sqrt(X**2 + Y**2)
```

This operation will create three temporary arrays, plus a non temporary which is returned to distance, the X^2, Y^2, and $+$ operations will each allocate a temporary array which is discarded after the line is executed, the square root operation also allocates an array, which is returned to the distance array. In this case, the first two temporary arrays are released once the fourth allocation is called, and one of the first two could be used since they match the allocation perfectly.

B. Implementation

The implementation is interface-compatible with malloc. This allows for a very low-intrusion integration by only changing 10 lines of code in the NumPy codebase. The implementation of the victim cache itself, including all matching and eviction strategies mention in section III-C, is a total of 237 lines of code, where the simple version with only one strategy is 81 lines. Figure 1 illustrates the data-structures, which are maintained.


Victim Cache

line = **4**
bytes_used = **100663296**
bytes_max = **536870912**

line	bytes	pointer
0	10485760	0xe3a3
1	5242880	0xa2d3
2	71303168	0x3133
3	13631488	0x42a3
4	0	0x0
5	0	0x0
6	0	0x0

Fig. 1: Illustration of the victim-cache data-structures for a victim-cache with seven cache-lines, a maximum size of 512MB and currently populated with four entries consuming 96MB.

The simplest implementation maintains the currently consumed *bytes_used* of the victim-cache, the *bytes_max* maximum number of bytes allowed for consumption, and the currently used *line* in the victim-cache. The following section describes different strategies of using the victim-cache. The implementation is available as a github-fork[1] of NumPy 1.7 on the branches *victim_cache* and *victim_cache_clean*. The branch *victim_cache* contains the implementation featuring the multiple algorithms which are described in the following section. The branch *victim_cache_clean* contains the cleaned up, less than 100 source-lines of code, implementation featuring a single strategy and the possibility of enabling/disabling the victim cache via environment-options.

C. Algorithms

Buffer management algorithms are a well researched area[7]. However, for many scenarios a simple solution is as good, or better, than advanced adaptive algorithms. For that reason, we limit the experiments in this work to six well-known algorithms. Three for matching buffers to requirements and three for selecting a buffer to eliminate when the allocated buffer space is saturated.

For matching buffers, we use three very simple algorithms, Exact, First, and Best. Exact will only return a fit if the requested buffer-size is exactly the same size as the buffer in the victim cache. First will return the first of the tested buffers large enough to hold the requested buffer. Best will search all buffers in the victim-cache and return the buffer that is as large as the requested size, and with as little extra space as possible. If an exact match is found, it is returned immediately.

If the maximum allowed buffer size would be exceeded by adding an allocation, an existing buffer in the cache must be evicted. Choosing one can also be done in many ways including Round-Robin, Second-Chance, and Random. Round-Robin will evict buffers from the victim-cache in the order they are added. If a buffer is selected for reuse, it will be added to the end once it is released again. It could also be described as evicting the oldest cache-line first. Second-chance is well known from demand-paging in operating systems and is aptly named. If a buffer is next to be evicted it will be marked at ready-to-evict, but the algorithm will in-fact move on in the list, only if a buffer is revisited, i.e. when a buffer is marked as ready and is next to be evicted, will it actually be selected for eviction. The worst case scenario is that all buffers must be visited once before one can be chosen for eviction, but in reality is will be more like round-robin, but with a second chance for some buffers. Random selection is extremely simple; a random buffer in the list is selected for eviction, while this may appear as a strange choice this approach has the advantage that it will not fall into a pattern where the same set of buffers are continuously evicted.

As the applications that NumPy is commonly used for, are highly regular in their execution pattern, we expect the simplest algorithms to perform very well, i.e. Exact-fit for matching and round-robin for eviction. If this is the case, there is no reason to keep the more advanced algorithms in a final version and the codebase can be kept very small indeed.

IV. Comparison

To evaluate the performance of the victim-cache, we have chosen 15 different benchmarks, that use a broad range of NumPy functionality. We have chosen some benchmarks that

[1]http://github.com/cphhpc/numpy/

operate on one-dimensional arrays and perform typical Monte Carlo simulations. For two-dimensional benchmarks, we use a selection of classic physics based computational kernels, for higher dimensions, we use a 3D Lattice-Boltzmann simulation and an n-body simulation. To show that the approach is also valid in other scenarios, we also use naïve implementations of FFT, LU, and matrix multiplication. The source-code for the benchmarks are available for closer inspection in the github-fork [2] on the *victim_cache* branch in the *benchmark/Python/* folder.

A. One-dimensional benchmarks

For testing with one-dimensional applications, we have chosen three different Monte Carlo based implementations. The simplest version is the original Monte Carlo Pi simulation that derives the value of π through a series of simulated dart throws. The other two benchmarks are taken from financial analysis domain and attempt to price a set of stock options, using the Black-Scholes model for European pricing and swaptions in the LIBOR market model, respectively. The Monte Carlo Pi simulation generates only a few temporary arrays in each iteration, whereas the Black-Scholes implementation generates as much as 67 temporary arrays in an iteration. The number of temporary arrays generated by the Swaption implementation varies with the input data and the amount of elements in each temporary array is relatively small.

B. Two-dimensional benchmarks

For two-dimensional applications we have chosen a common Jacobi five-point stencil application, a successive over relaxation (SOR), a shallow water simulation, a WireWorld simulation, a Lattice-Boltzmann simulation and a cloth physics simulation. The Jacobi stencil is chosen for its simplicity, where the others are chosen because they are larger applications, which would be hard to optimize by hand. The SOR simulation essentially does the same as the Jacobi stencil, but implemented with a red/black update scheme, and includes a global delta calculation. The Lattice-Boltzmann, shallow water, and cloth simulations all simulate movement in a two-dimensional space with different models for force propagation. The Jacobi stencil code is fairly compact but still generates 9 temporary arrays in each iteration. The other benchmarks generate a larger number of temporary arrays that are candidates for optimization from the victim cache.

C. Higher-dimensional benchmarks

To show the effects of the victim-cache with problems that have multiple dimensions, we have chosen a some classic computational kernels, namely a naïve n-body simulation, a k-nearest-neighbor search, and a Lattice-Boltzmann simulation in 3D space. The k-nearest-neighbors search has a low amount of temporary arrays, and the n-body simulation and Lattice-Boltzmann simulations have a moderate amount of temporary arrays.

[2]http://github.com/cphhpc/numpy/

Benchmark	Problemsize	Iterations
Black Scholes	$8 \cdot 10^6$	5
Bolzmann 3D	$120 \times 100 \times 100$	5
Bolzmann D2Q9	800×800	5
Cloth	3000×3000	1
FFT	18	N/A
Jacobi Stencil	$10000 \times 4000 \times 10$	10
KNN	$2 \cdot 10^6 \times 10$	3
LU Factor.	500×500	N/A
Matrix Mul	800	N/A
Monte Carlo PI	$2 \cdot 10^7$	10
NBody	3000×1	1
Shallow Water	3000×3000	5
SOR	4000×4000	5
Swaption	1000	N/A
Wire World	5000×5000	5

TABLE I: Overview of benchmarks and problem sizes.

D. Kernel benchmarks

To broaden the experiment we have chosen a set of kernels that are traditionally implemented in external libraries and implemented them in NumPy. The kernels comprise naïve versions of matrix multiplication, LU factorization, and Fast Fourier Transformation (FFT). The FFT kernel generates a low amount of temporary arrays. The where the matrix multiplication and LU kernels generate a large amount of temporary arrays, where the arrays generated by the LU kernel are small, and the ones generated by the matrix multiplication are large.

Table I provides the full list of benchmarks along with the parameters for their execution.

E. Results

As the victim cache mitigates the work related to allocating array memory from the operating system, there is a clear relation between the number of temporary arrays and the gained speedup. Figure 2 shows the speedups obtained from running the same NumPy code with three different sizes of the victim cache. Each benchmark has been set up with parameters that cause the benchmarks to run around 10 seconds with no victim cache. Each benchmark is then executed with the same input data, and varying sizes of the victim cache and the wall-clock times are used to compute the speedup. All benchmarks are executed on a AMD Opteron 6272 CPU with 128 GB of memory, running with Ubuntu 12.04.2 LTS. The execution times were stable with a maximum wall-clock time deviation of 0.08 seconds.

We can see that some experiments gain no speedup at all, but none of the experiments show any slow down from the victim cache. For our problem sizes, a moderate size victim cache of 512 MB is sufficient to gain the maximum performance speed up, except for the Jacobi example, which shows a large speedup when utilizing 1GB of victim cache memory.

The SOR, shallow water, and Jacobi benchmarks show as much as 2.3 times speedup from using the victim cache, which we consider a significant result. The following section provides further analysis of the results.

F. Analysis

The results are quite convincing, while a few benchmarks do not show any improvement in performance most do, and

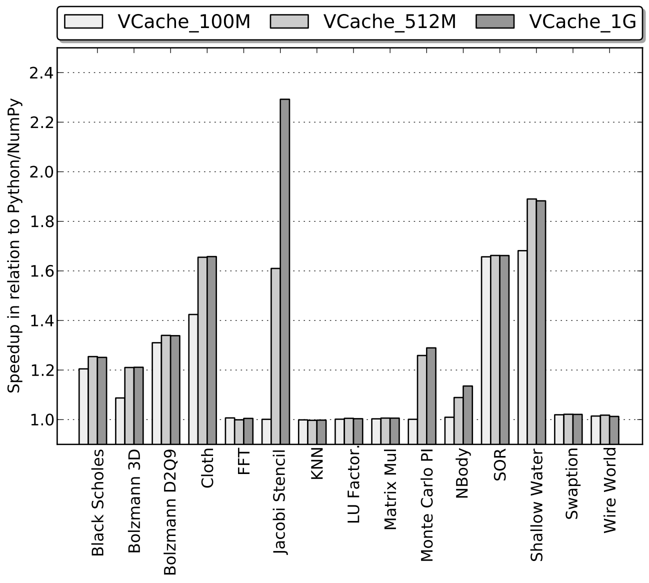


Fig. 2: Speedup of victim-cache in relation to unmodified Python/NumPy of the 15 benchmark applications.

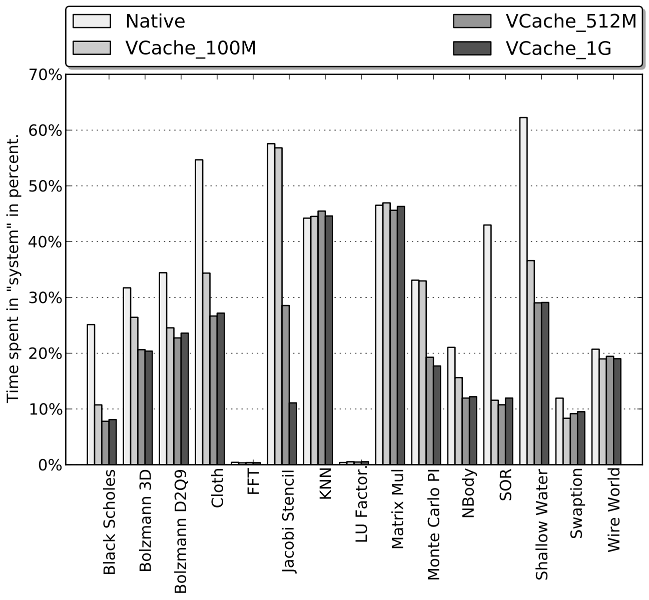


Fig. 3: Time spent in system/kernel-level reported in percentage of total wall-clock time.

no less than four show a performance increase of more than 50%, two of more then 100%. At a first glance, it is hard to see why a simple victim-cache can improve performance that much. The explanation can be found in the way glibc under Linux handles memory allocations. Any allocation to an area that is larger than 128KB[3], is allocated using mmap rather than sbrk[8]. The consequence is that memory that is allocated with mmap, will be released to the operating system, when free is called. This means that the memory is actually returned to Linux as opposed to memory that is allocated with sbrk, which is kept for reuse by glibc. The consequence of this is that the many, large, temporary array allocations in NumPy are moved back and forth between user space and kernel space. The actual call to the operating system represents an overhead in itself, but the majority of the time is spent zeroing the memory to stop information from leaking between processes. Thus the big advantage of the victim cache model it that we save a write to the temporary memory, which in effect doubles the cost of simple array operations.

To verify that the above description is in fact the reason for the observed performance improvements, the benchmarks were repeated using the time tool in order measure time spend in user-level and kernel-level respectively. Figure 3 show the time spent in kernel-level for each benchmark, for standard NumPy (Native), and the victim-cache implementation for three different cache sizes. In this figure, lower means less time spent in system.

There is an obvious correlation between the benchmark where we observed improvement in the overall runtime, and the drop in time spent in kernel-level. Moreover, going from 512MB to 1GB of victim-cache only shows a significant impact in the Jacobi Stencil benchmark where time spent in kernel-level is reduced by more than half. A manual experiment to increase the victim-cache to 2GB showed no further improvement in runtime.

[3]by default but may be changed by the user

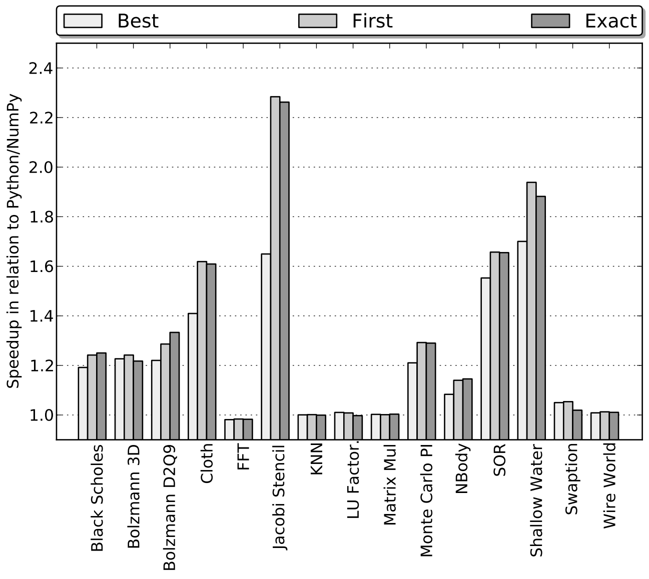


Fig. 4: Comparison of fitting strategies.

We go on experimenting with the three different matching algorithms, best-fit, first-fit, and exact-fit. Best-fit requires an inspection of each element of the cache for each victim-cache lookup, the consequence of which is clearly shown in figure 4. The difference between first-fit and exact-fit is marginal and varies across the benchmark suite.

Figure 5 illustrates the results of experiments with different eviction strategies. The random eviction-scheme, which is known to work well for page-replacement in the operating system, clearly does not apply for the victim-cache. This result was expected since most benchmarks demonstrate a high degree of regularity. Second change and oldest first are

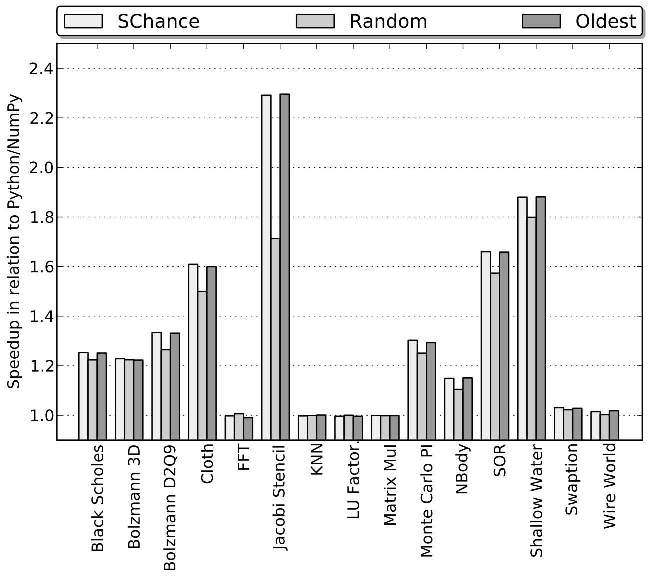


Fig. 5: Comparison of eviction strategies.

indistinguishable.

Given the interface compatibility and the previous description of malloc, a valid question is: "why not just parameterize malloc correctly?". Because the zeroing memory only affects allocations above `MMAP_THRESHOLD` bytes, one could consider simply increasing `MMAP_THRESHOLD`. However, glibc malloc does not allow for `MMAP_THRESHOLD` to go beyond `MMAP_THRESHOLD_MAX`, which on 64-bit systems is only 4MB. Changing the maximum value would require recompiling glibc and is thus a fairly intrusive procedure. A more drastic approach would be to re-compile the Linux kernel with `__GFP_ZERO` defined to zero, which would completely eliminate the zeroing of pages across the operating system.

V. Future Work

The solution itself is quite simple, but the implementation may be subjected to further refinements. In the current implementation, a rather simple list-based lookup search is performed, yield a runtime complexity of $O(n)$ over the number of entries. Many better strategies exist, such as tree-based lookups that can reduce this overhead.

The results presented in this article show that there is indeed an overhead involved in the generation of temporary arrays and that a victim cache can reduce the overhead by using some extra memory. However, a more thorough approach is to avoid creating the temporary arrays completely.

However, unlike the victim cache, such a change requires changes in many places within the NumPy libraries. We are actively investigating this approach as part of the Bohrium runtime system[1].

We have produced a cleaned up version of the changes, which only supports exact matching and round-robin eviction. This cleaned up version is reduced to 81 lines of C-code, combined with the ten lines in the NumPy multiarraymodule brings the total lines of code in the victim-cache implementation up to 91. We plan to submit this patch to NumPy upstream developers so NumPy user can reap the benefits discovered.

VI. Conclusion

We have implemented a very simple and non-intrusive victim-cache in NumPy, and evaluated the effects on a variety of different benchmarks. The experiments clearly show that the victim-cache is able to reduce much of the overhead that occurs when NumPy allocates memory from the operating system. In no case did see an actual slowdown. Generally we see an average improvement of 32% accross the benchmark suite, if we cherry-pick only the benchmarks where we see an improvement the average speedup is 52%. The best observed speedup is 230% of the Jacobi Stencil benchmark. A victim-cache of 512MB was sufficient to harvest all the gains of victim-caching in all benchmarks except one, the Jacobi Stencil.

We experimented with three different matching strategies, and three different eviction strategies, however the high degree of regularity in the benchmark suite meant that the simplest algorithms exact-fit and oldest-eviction first performed as good as or better than the more advanced strategies.

Overall we conclude that the victim-cache is a nearly cost-free optimization, that potentially benefits more than half of all NumPy applications, without any requirements towards to programmer. The implementation comprises ten lines of changes to the NumPy multiarraymodule and 81 lines for the victim-cache itself, in total 91 lines of code.

Acknowledgments

The Cloth simulation is ported to Python from a Java based implementation written by Jared Counts[4]. The Black-Scholes simulation is vectorized based on an example from Espen Haug[5]. The 2D Lattice-Boltzmann simulation is based on code by Jonas Latt[6]. The 3D Lattice-Boltzmann simulation is based on code written by Iain Haslam[7]. The Swaption pricing in LIBOR market is written by Lykke Rasmussen[8]. The LU factorization is based on code in the SciMark2 suite.

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (hiperfit.dk) under contract number 10-092299.

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the 'High Performance High Productivity' project under contract number 09-067060.

References

[1] M. Kristensen, S. Lund, K. Skovhede, T. Blum, and B. Vinter, "Bohrium: a virtual machine approach to portable parallelism," *in submission to ICPADS13*, 2013.

[4]http://gamedev.tutsplus.com/tutorials/implementation/simulate-fabric-and-ragdolls-with-simple-verlet-integration/
[5]http://www.espenhaug.com/black_scholes.html
[6]http://wiki.palabos.org/_media/numerics:cylinder.pdf
[7]http://www.exolete.com/images/lbm3d.m
[8]http://quantess.net/

[2] D. A. Wheeler, “Sloccount, a set of tools for counting physical source lines of code (sloc),” *URL http://www. dwheeler. com/sloccount*, 2004.

[3] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*. IEEE, 1990, pp. 364–373.

[4] P. R. Wilson, “Some issues and strategies in heap management and memory hierarchies,” *ACM SIGPLAN Notices*, vol. 26, no. 3, pp. 45–52, 1991.

[5] T. Lang, C. Wood, and E. B. Fernández, “Database buffer paging in virtual storage systems,” *ACM Transactions on Database Systems (TODS)*, vol. 2, no. 4, pp. 339–351, 1977.

[6] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, “Dynamic storage allocation: A survey and critical review,” in *Memory Management*. Springer, 1995, pp. 1–116.

[7] ——, “Dynamic storage allocation: A survey and critical review,” in *Memory Management*. Springer, 1995, pp. 1–116.

[8] G. Insolvibile, “Advanced memory allocation,” *Linux Journal*, vol. 2003, no. 109, p. 7, 2003.

6.4 Bypassing the Conventional Software Stack Using Adaptable Runtime Systems

Bypassing the Conventional Software Stack Using Adaptable Runtime Systems

Simon A. F. Lund, Mads R. B. Kristensen, Brian Vinter, and Dimitrios Katsaros

Niels Bohr Institute, University of Copenhagen, Denmark
{safl/madsbk/vinter}@nbi.dk
Computer Science Department, University of Copenhagen, Denmark
rth738@alumni.ku.dk

Abstract. High-level languages such as Python offer convenient language constructs and abstractions for readability and productivity. Such features and Python's ability to serve as a steering language as well as a self-contained language for scientific computations has made Python a viable choice for high-performance computing. However, the Python interpreter's reliance on shared objects and dynamic loading causes scalability issues that at large-scale consumes hours of wall-clock time just for loading the interpreter.
The work in this paper explores an approach to bypass the conventional software stack, by replacing the Python interpreter on compute nodes with an adaptable runtime system capable of executing the compute intensive portions of a Python program. Allowing for a single instance of the Python interpreter, interpreting the users' program and additionally moving program interpretation off the compute nodes. Thereby avoiding the scalability issue of the interpreter as well as providing a means of running Python programs on restrictive compute notes which are otherwise unable to run Python.
The approach is experimentally evaluated through a prototype implementation of an extension to the Bohrium runtime system. The evaluation shows promising results as well as identifying issues for future work to address.

Keywords: Scalability, Python, import problem, dynamic loading

1 Introduction

Python is a high-level, general-purpose, interpreted language. Python advocates high-level abstractions and convenient language constructs for readability and productivity. The reference implementation of the Python interpreter, CPython, provides rich means for extending Python with modules implemented in lower-level languages such as C and C++. Lower-level implementations can be written from scratch and conveniently map to Python data-structures through Cython[4], function wrappers to existing libraries through SWIG[3, 2], or using the Python ctypes[1] interface.

[1] http://docs.python.org/2/library/ctypes.html

The features of the language itself and its extensibility make it attractive as a steering language for scientific computing, which the existence of Python at high-performance compute sites confirms. Furthermore, there exists a broad range of Python wrappers to existing scientific libraries and solvers[11, 20, 13, 8, 9].

Python transcends its utilization as a steering language. SciPy[2] and its accompanying software stack[17, 18, 12] provides a powerful environment for developing scientific applications. The fundamental building block of SciPy is the multidimensional arrays provided by NumPy[17]. NumPy expands Python by providing a means of doing array-oriented programming using array-notation with slicing and whole-array operations. The array-abstractions offered by NumPy provides the basis for a wealth of existing[6] and emerging[19, 21, 14] approaches that increases the applicability of Python in an HPC environment. Even though advances are made within these areas, a problem commonly referred to as the the *import problem*[1, 15, 22] still persists at large-scale compute sites. The problem evolves around dynamic loading of CPython itself, built-in modules, and third party modules. Recent numbers reported on Hopper[22] state linear scale with the number of cores, which amount to a startup time of 400 seconds on 1024 cores and one hour for 8000 cores.

The approach in this paper explores a simple idea to avoid such expensive startup costs: execute one instance of the Python interpreter regardless of the cluster size. Furthermore, we allow the Python interpreter to run on an external machine that might not be part of the cluster. The machine can be any one of; the user's own laptop/workstation, a frontend/compile node, or a compute node, e.g. any machine that is accessible from the compute-site.

A positive complementary effect, as well as a goal in itself, is that the Python interpreter and the associated software stack need not be available on the compute nodes.

The work in this paper experimentally evaluates the feasibility of bypassing the conventional software stack, by replacing the Python interpreter on the compute nodes with an adaptable runtime system capable of executing the computationally heavy part of the users' program. The approach facilitates the use of Python at restrictive compute-sites and thereby broadens application of Python in HPC.

2 Related Work

The work within this paper describes, to the authors knowledge, a novel approach for handling the Python *import problem.* This section describes other approaches to meeting the same end.

Python itself support a means for doing a user-level override of the import mechanism[3] and work from within the Python community has improved upon

[2] `http://www.scipy.org/stackspec.html`
[3] `http://legacy.python.org/dev/peps/pep-0302/`

the import system from version 2.6 to 2.7 and 3.0. In spite of these efforts, the problem persists.

One aspect of the *import problem* is the excessive stress on the IO-system caused by the object-loader traversing the filesystem looking for Python modules. Path caching through collective operations is one approach to lowering overhead. The mpi4py[7] project implements multiple techniques to path caching where a single node traverses the file-system and broadcasts the information to the remaining $N-1$ nodes. The results of this approach show significant improvements to startup times from hours to minutes but relies on the *mpi4py* library and requires maintenance of the Python software-stack on the compute-nodes.

Scalable Python[4], first described in[9], addresses the problem at a lower level. Scalable Python, a modification of CPython, seeks to address the *import problem* by implementing a parallel IO layer utilized by all Python import statements. By doing so only a single process, in contrast to N processes, perform IO. The result of the IO operation is broadcast to the remaining $N-1$ nodes via MPI. The results reported in[9] show significant improvements towards the time consumed by Python import statements at the expense of maintaining a custom CPython implementation.

Relying on dynamically loaded shared objects is a general challenge for large-scale compute-sites with a shared filesystem. SPINDLE[10] provides a generic approach to the problem through an extension to the GNU Loader.

The above described approaches apply different techniques for improving performance of dynamic loading. A different strategy which in this respect is thematically closer to the work within this paper is to reduce the use of dynamic loading. The work in[15] investigate such strategy by replacing as much dynamic loading with statically compiled libraries. Such technique in a Python context can by applied through the use of Python freeze[5] and additional tools[6] exists to support it.

3 The Approach

The previous sections describe and identify the CPython import system as the culprit guilty of limiting the use of Python / NumPy at large-scale compute sites. Dynamic loading and excessive path searching are accomplices to the havoc raised. The crime committed is labelled as the Python *import problem.*

Related work let the culprit run free and implement techniques to handling the havoc raised. The work within this paper focuses on restricting the culprit and thereby preventively avoiding the problem.

The idea is to run a single instance of the Python interpreter, thereby keeping the overhead constant and manageable. The remaining instances of the interpreter are replaced with a runtime system capable of efficiently executing the portion of the Python / NumPy program responsible for communication and

[4] `https://gitorious.org/scalable-python`
[5] `https://wiki.python.org/moin/Freeze`
[6] `https://github.com/bfroehle/slither`

computation. Leaving the task of interpreting the Python / NumPy program, conditionals, and general program flow up to the interpreter. The computationally heavy parts are delegated to execution on the compute nodes through the runtime system.

3.1 Runtime System

The runtime used in this work is part of the Bohrium[19] project[7]. The Bohrium runtime system (BRS) provides a backend for mapping array operations onto a number of different hardware targets, from multi-core systems to clusters and GPU enabled systems. It is implemented as a virtual machine capable of making runtime decisions instead of a statically compiled library. Any programming language can use BRS in principle; in this paper though, we will use the Python / NumPy support exclusively.

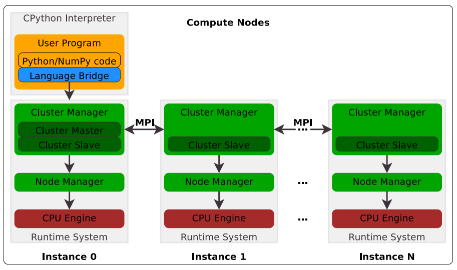


Fig. 1. Illustration of communication between the runtime system components *without* the use of the proxy component.

The fundamental building block of BRS is the representation of programs in the form of *vector bytecode*. A vector bytecode is a representation of an operation acting upon an array. This can be one of the standard built-in operations such as element-wise addition of arrays, function promotion of trigonometric functions over all elements of an array, or in functional terms: map, zip, scan and reduction, or an operation defined by third party.

BRS is implemented using a layered architecture featuring a set of interchangeable *components*. Three different types of components exist: *filters*, *managers*, and *engines*. Figure 1 illustrates a configuration of the runtime system configured for execution in a cluster of homogenous nodes. The arrows represent vector bytecode sent through the runtime system in a top-down fashion, possibly altering it on its way.

Each component exposes the same C-interface for initialization, shutdown, and execution thus basic component interaction consists of regular function calls. The component interface ensures isolation between the language bridge that runs the CPython interpreter and the rest of Bohrium. Thus, BRS only runs a single instance of the CPython interpreter no matter the underlying architecture – distributed or otherwise.

Above the runtime, a language bridge is responsible for mapping language constructs to vector bytecode and passing it to the runtime system via the C-interface.

Managers manage a specific memory address space within the runtime system and decide where to execute the vector bytecode. In figure 1 a node man-

[7] `http://www.bh107.org`

ager manages the local address space (one compute-node) and a cluster-manager which handles data distribution and inter-node communication through MPI. At the bottom of the runtime system, we have the execution engines, which are responsible for providing efficient mapping of array operations down to a specific processing unit such as a CPU or a GPU.

3.2 Proxy Manager

Currently, all Bohrium components communicate using local function calls, which translates into shared memory communication. Figure 1 illustrates the means of communication within the BRS prior to the addition of the proxy component. As a result, the language bridge, which runs a CPython interpreter, must execute on one of the cluster-nodes. In order to circumvent this problem, we introduce a new *proxy* component.

This new component acts as a network proxy that enables Bohrium components to exchange vector bytecode across a network. Figure 2 illustrates the means for communication which the Proxy component provides. By using this functionality, separation can be achieved between the implementation of any application using Bohrium and the actual hardware on which it runs. This is an important property when considering cases of supercomputers or clusters, which define specific characteristics for the execution of tasks on them.

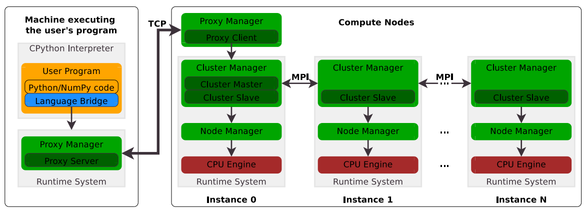


Fig. 2. Illustration of communication between the runtime system components *with* the use of the proxy component.

The proxy component is composed of two parts – a server and a client. The server exposes the component interface (init, execute, and shutdown) to its parent component in the hierarchy whereas the client uses its child component interface. When the parent component calls execute with a list of vector bytecodes, the server serialize and sends the vector bytecodes to the client, which in turn uses its child component interface to push the vector bytecodes further down the Bohrium hierarchy. Besides the serialized list of vector bytecodes, the proxy component needs to communicate array-data in two cases.

When the CPython interpreter introduces existing NumPy arrays and Python scalars to a Bohrium execution. Typically, this happens when the user application loads arrays and scalars initially. When the CPython interpreter access the result of a Bohrium execution directly. Typically, this happens when the user application evaluates a loop-condition based on some array and scalar data.

Both the server and the client maintain a record of array-data locations thus avoiding unnecessary array-data transfers. Only when the array-data is involved in a calculation at the client-side will the server send the array-data. Similarly, only when the CPython interpreter request the array-data will the client send the array-data to the server.

In practice, when the client sends array-data to the server it is because the CPython interpreter needs to evaluate a scalar value before continuing. In this case, the performance is very latency sensitive since the CPython interpreter is blocking on the scalar value. Therefore, it is crucial to disable Nagle's TCP/IP algorithm[16] in order achieve good performance. Additionally, the size of the vector bytecode lists is significantly greater than the TCP packet header thus limiting the possible advantage of Nagle's TCP/IP algorithm. Therefore, when the proxy component initiates the TCP connection between server and client it sets the `TCP_NODELAY` socket option.

4 Evaluation

The basic idea of the approach is to have a single instance of CPython interpreting the user's program, such as figure 2 illustrates. With a single isolated instance of the interpreter the *import problem* is solved by design. The second goal of the approach is to facilitate execution of a Python program in a restricted environment where the Python software stack is not available on the compute nodes.

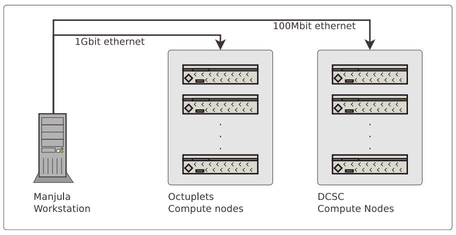


Fig. 3. Octuplets and DCSC two physically and administratively disjoint clusters of eight and sixteen nodes. Octuplets is a small-scale research-cluster managed by the eScience group at the Niels Bohr Institute. DCSC is a larger compute-site for scientific computation in Denmark. Gbit ethernet facilitate the connection between Manjula and the octuplet cluster and 100Mbit ethernet between Manjula and DCSC.

The potential *Achilles heel* of the approach is in its singularity, with a single *remote* instance of the interpreter network latency and bandwidth limitations potentially limit application of the approach.

Network latency can stall execution of programs when the round-trip-time of transmitting vector bytecode from the interpreter-machine to the compute node exceeds the time spent computing on previously received vector bytecode. Bandwidth becomes a limiting factor when the interpreted program needs large amounts of data for evaluation to proceed interpretation and transmission of vector bytecode. The listing below contains descriptions of the applications used as well as their need for communication between interpreter and runtime. The sourcecode is available for closer inspection in the Bohrium repository[8].

Black Scholes implements a financial pricing model using a partial differential equation, calculating price variations over time[5]. At each time-step

[8] `http://bitbucket.org/bohrium/bohrium`

the interpreter reads out a scalar value from the runtime representing the computed price at that time.

Heat Equation simulates the heat transfer on a surface represented by a two-dimensional grid, implemented using jacobi-iteration with numerical convergence. The interpreter requires a scalar value from the runtime at each time-step to evaluate whether or not simulation should continue. Additionally when executed with visualization the entire grid is required.

N-Body simulates interaction of bodies according to the laws of Newtonian physics. We use a straightforward algorithm that computes all body-body interactions, $O(n^2)$, with collisions detection. The interpreter only needs data from the runtime at the end of the simulation to retrieve the final position of the bodies. However, the interpreter will at each time-step, when executed for visualization purposes, request coordinates of the bodies.

Shallow Water simulates a system governed by the Shallow Water equations. The simulation initates by placing a drop of water in a still container. The simulation then proceeds, in discrete time-steps, simulating the water movement. The implementation is a port of the MATLAB application by Burkardt [9]. The interpreter needs no data from the runtime to progress the simulation at each time-step. However, the interpreter will at each time-step, when executed for visualization purposes, request the current state of the simulated water.

We benchmark the above applications on two Linux-based clusters (Fig. 3). The following subsections describe the experiments performed and report the performance numbers.

4.1 Proxy Overhead

We begin with figure 4 which show the results of running the four benchmark applications on the octuplet cluster using eight compute nodes and two different configurations:

With Proxy The BRS configured with the proxy component and the interpreter is running on Manjula. This configuration is equivalent to the one illustrated in figure 2.

Without Proxy The BRS configured without the proxy component. The interpreter is running on the first of the eight compute nodes. This setup is equivalent to the one illustrated in figure 1.

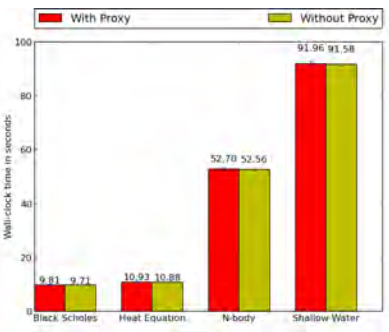


Fig. 4. Elapsed wall-clock time in seconds of the four applications on the octuplet compute nodes with and without the proxy component.

[9] `http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_2d/`

We cannot run Python on the DCSC cluster for the simple reason that the software stack is too old to compile Python 2.6 on the DCSC compute nodes. Thus, it is not possible to provide comparable results of running with and without the Proxy component.

The purpose of this experiment is to evaluate the overhead of introducing the proxy component in a well-behaved environment. There were no other users of the network, filesystem, or machines. Round-trip-time between Manjula and the first compute node was average at 0.07ms during the experiment. The error bars show two standard deviations from the mean. The overhead of adding the proxy component is within the margin of error and thereby unmeasurable.

4.2 Latency Sensitivity

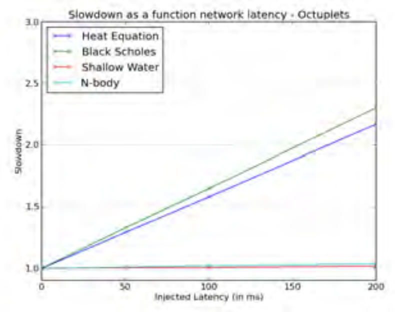


Fig. 5. Slowdown of the four applications as a function of injected latency between Manjula and octuplet compute node.

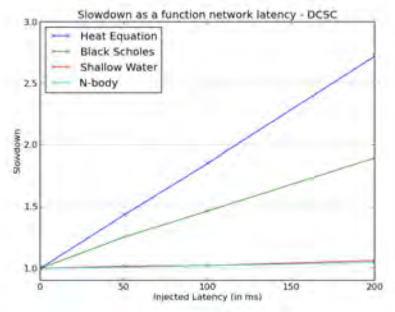


Fig. 6. Slowdown of the four applications as a function of injected latency between Manjula and DCSC compute node.

We continue with figures 5 and 6. The BRS configured with the proxy component, running the interpreter on Manjula. Figure 2 illustrates the setup. The purpose of the experiment is to evaluate the approach' sensitivity to network latency. Latencies of 50, 100, 150, and 200ms are injected between Manjula and the compute node running the proxy client. The figures show slowdown of the applications as a function of the injected latency.

The applications Shallow Water and N-body are nearly unmeasurably affected by the injected latency. The observed behavior is as expected since the interpreter does not need any data to progress interpretation. It is thereby possible to overlap transmission of vector bytecode from the interpreter-machine with computation on the compute nodes.

The injected latency does, however, affect the applications Heat Equation and Black Scholes. The observed behavior is as expected since the interpreter requires a scalar value for determining convergence criteria for Heat Equation and sampling the pricing value for Black Scholes. Network latency affects the results from the DCSC cluster the most, with a worst-case of a 2.8 slowdown. This is due to the elapsed time being lower when using the sixteen DCSC compute nodes. Since less time is spent computing more time is spent waiting and thereby a relatively larger sensitivity to network latency.

4.3 Bandwidth Sensitivity

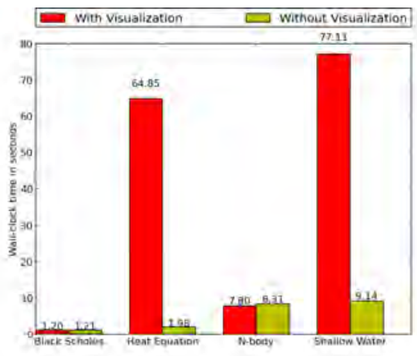


Fig. 7. Elapsed wall-clock time of the four applications with and without visualization on the octuplet compute nodes.

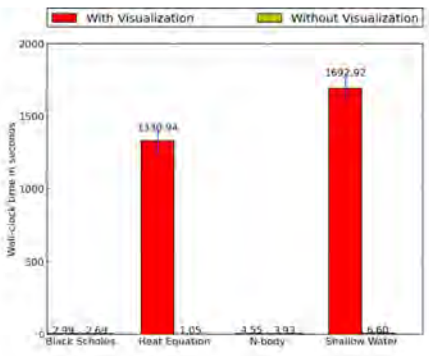


Fig. 8. Elapsed wall-clock time of the four applications with and without visualization on the DCSC compute nodes.

The last experiment sought to evaluate the sensitivity to high network bandwidth utilization. Figures 7 and 8 show the results of an experiment where the four applications were running with visualization updated at each time-step. The BRS configured with the proxy component; Manjula is running the Python interpreter. Figure 2 illustrates the setup.

When executing with visualization, the interpreter requires a varying (depending on the application) amount of data to be transmitted from the compute nodes to the interpreter-machine at each time step. Thereby straining the available bandwidth between the interpreter-machine and the compute node running the proxy-client.

Black-Scholes although sensitive to latency due to the need of transmitting the computed price at each time-step, does not require any significant amount of data to be transferred for visualization, neither does the N-Body simulation. However, the two other applications Heat Equation and Shallow Water require transmission of the entire state to visualize dissipation of heat on the plane and the current movement of water. These two applications are sufficient to observe a key concern of the approach.

We observe a slowdown of about $\times 1260$ (Heat Equation) and $\times 257$ (Shallow Water) when running on the DCSC nodes. We observe a slowdown of about $\times 32.8$ (Heat Equation) and $\times 8.5$ (Shallow Water) when running in the octuplet nodes. These results clearly show that network bandwidth becomes a bottleneck, with disastrous consequences in terms of execution time and thus a limiting factor for applying the approach for such use.

The slowdown is much worse when running on the DCSC compute nodes compare to the slowdown on the octuplet nodes. This is due to the interconnect being 100Mbit ethernet to the DCSC in relation to the 1Gbit ethernet connection to the octuplet nodes.

5 Future Work

The evaluation revealed bandwidth bottlenecks when the machine running the interpreter requests data for purposes such as visualization. The setup in the evaluation was synthetic and forced requests of the entire data-set at each time-step without any transformation of the data, it can, therefore, be regarded as a worst-case scenario.

One could argue that the setup is not representative for user behaviour and instead assume that the user would only need a snapshot of data at every $timestep/K$ iteration and with lowered resolution such as every I'th datapoint and thus drastically lowering the bottleneck. However, to address the issue future work will involve compressed encoding of data transmitted as well as suitable downsampling for the visualization purpose.

The primary focus point for future work is now in progress and relates to the effective throughput at each compute-node. The current implementation of the execution engine uses a virtual-machine approach for executing array operations. In this approach the virtual machine delegate execution of each vector bytecode to statically compiled routine. Within this area, a wealth of optimizations are applicable by composing multiple operations on the same data and hereby *fusing* array operations together.

Random-number generators, linear spaces of data, and iotas, when combined with reductions are another common source for optimization of memory utilization and locality. Obtaining such optimizations within the runtime require the use of JIT compilation techniques and potentially increase the use dynamic loading of optimized codes. The challenge for this part of future work involves exploration of how to get such optimization without losing the performance gained to runtime and JIT compilation overhead.

6 Conclusions

The work in this paper explores the feasibility of replacing the Python interpreter with an adaptable runtime system, with the purpose of avoiding the CPython scalability issues and providing a means of executing Python programs on restrictive compute nodes which are otherwise unable to run the Python interpreter.

The proxy component, implemented as an extension to the Bohrium runtime system (BRS), provides the means for the BRS to communicate with a single *remote* instance of the Python interpreter. The prototype implementation enabled evaluation of the proposed approach of the paper.

Allowing the interpreter to execute on any machine, possibly users' own workstations/laptops, leverages a Python user to utilize a cluster of compute nodes or a supercomputer with direct realtime interaction. However, it also introduces concerns with regards to the effect of network latency and available bandwidth, between the machine running the interpreter and the compute node running the proxy client, on program execution. These concerns were the themes for the conducted evaluation.

Results showed that the overhead of adding the proxy component and thereby the ability for the BRS to use a remote interpreter was not measurable in terms of elapsed wall-clock time, as results were within two standard deviations of the measured elapsed wall-clock. The results additionally showed a reasonable tolerance to high network latency, at $50ms$ round-trip-time, slowdown ranged from not being measurable to $\times 1.3 - \times 1.4$. In the extreme case of $200ms$ latency ranged from not being measurable to a slowdown of $\times 1.9 - \times 2.8$.

The primary concern, and focus for future work, presented itself during evaluation of bandwidth requirements. If the Python program requests large amounts of data then the network throughput capability becomes a bottleneck, severely impacting elapsed wall-clock as well as saturating the network link, potentially disrupting other users.

The results show that the approach explored within this paper does provide a possible means to avoid the scalability issues of CPython, allowing direct user interaction and enabling execution of Python programs in restricted environments that are otherwise unable to run interpreted Python programs. The approach is, however, restricted to transmission of data such as vector bytecode, scalars for evaluation of convergence criteria, boolean values, and low-volume data-sets between the interpreter-machine and runtime. This does, however, not restrict processing of large-volume datasets within the runtime on and between the compute nodes.

7 Acknowledgments

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (hiperfit.dk) under contract number 10-092299.

References

1. A. Ahmadia. Solving the import problem: Scalable Dynamic Loading Network File Systems. Technical report, Talk at SciPy conference, Austin, Texas, July 2012, 2012. `http://pyvideo.org/video/1201/solving-the-import-problem-scalable-dynamic-load`.
2. D. M. Beazley. Automated scientific software scripting with SWIG. volume 19, pages 599–609. Elsevier, 2003.
3. D. M. Beazley et al. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk workshop*, pages 129–139, 1996.
4. S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.
5. F. Black and M. Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, pages 637–654, 1973.
6. J. Daily and R. R. Lewis. Using the global arrays toolkit to reimplement numpy for distributed computation. In *Proceedings of the 10th Python in Science Conference*, 2011.

7. L. Dalcin, R. Paz, M. Storti, and J. D Elia. MPI for Python: Performance improvements and MPI-2 extensions. *Journal of Parallel and Distributed Computing*, 68(5):655–662, 2008.
8. L. Drummond, V. Galiano, V. Migallon, and J. Penades. High-Level User Interfaces for the DOE ACTS Collection. In B. Kagstrom, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 251–259. Springer Berlin Heidelberg, 2007.
9. J. Enkovaaraa, M. Louhivuoria, P. Jovanovicb, V. Slavnicb, and M. Rännarc. Optimizing GPAW. *Partnership for Advanced Computing in Europe*, September 2012. Online: `http://www.prace-ri.eu/IMG/pdf/Optimizing_GPAW.pdf`.
10. W. Frings, D. H. Ahn, M. LeGendre, T. Gamblin, B. R. de Supinski, and F. Wolf. Massively Parallel Loading. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 389–398, New York, NY, USA, 2013. ACM.
11. K. Gawande and C. Webers. PyPETSc User Manual (Revision 1.0). Technical report, NICTA, 2009. `http://elefant.developer.nicta.com.au/documentation/userguide/PyPetscManual.pdf`.
12. J. D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.
13. D. I. Ketcheson, K. T. Mandli, A. J. Ahmadia, A. Alghamdi, M. Quezada de Luna, M. Parsani, M. G. Knepley, and M. Emmett. PyClaw: Accessible, Extensible, Scalable Tools for Wave Propagation Problems. *SIAM Journal on Scientific Computing*, 34(4):C210–C231, Nov. 2012.
14. M. R. B. Kristensen and B. Vinter. Numerical Python for scalable architectures. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 15:1–15:9, New York, NY, USA, 2010. ACM.
15. P. Marion, A. Ahmadia, and B. M. Froehle. Import without a filesystem: scientific Python built-in with static linking and frozen modules. Technical report, Talk at SciPy conference, Austin, Texas, July 2012, 2013. `https://www.youtube.com/watch?v=EOiEIWMYkwE`.
16. J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, Jan. 1984.
17. T. E. Oliphant. Python for Scientific Computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.
18. F. Prez and B. E. Granger. IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering*, 9(3):21–29, 2007.
19. M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter. Bohrium: a Virtual Machine Approach to Portable Parallelism. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2014 IEEE 28th International*. IEEE, 2014.
20. M. Sala, W. Spotz, and M. Heroux. PyTrilinos: High-Performance Distributed-Memory Solvers for Python. *ACM Transactions on Mathematical Software (TOMS)*, 34, March 2008.
21. K. Smith, W. F. Spotz, and S. Ross-Ross. A Python HPC Framework: PyTrilinos, ODIN, and Seamless. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 593–599. IEEE, 2012.
22. Z. Zhao, M. Davis, K. Antypas, Y. Yao, R. Lee, and T. Butler. Shared Library Performance on Hopper. In *CUG 2012, Greengineering the Future, Stuttgart, Germany*, 2012.

6.5 Bohrium: a Virtual Machine Approach to Portable Parallelism

Bohrium: a Virtual Machine Approach to Portable Parallelism

Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter
Niels Bohr Institute, University of Copenhagen, Denmark
{madsbk/safl/blum/skovhede/vinter}@nbi.dk

Abstract—
In this work we introduce, Bohrium, a runtime-system for mapping vector operations onto a number of different hardware platforms, from simple multi-core systems to clusters and GPU enabled systems. In order to make efficient choices Bohrium is implemented as a virtual machine that makes runtime decisions, rather than a statically complied library, which is the more common approach. In principle, Bohrium can be used for any programming language but for now, the supported languages are limited to Python, C++ and the .Net framework, e.g. C# and F#.

The primary success parameters are to maintain a complete abstraction from low-level details and to provide efficient code execution across different, current and future, processors.

We evaluate the presented design through a setup that targets a single-core CPU, an eight-node Cluster, and a GPU, all preliminary prototypes. The evaluation includes three well-known benchmark applications, *Black Sholes*, *Shallow Water*, and *N-body*, implemented in C++, Python, and C# respectively.

I. Introduction

Finding the solution for computational scientific and engineering problems often requires experimenting with various algorithms and different parameters with feedback in several iterations. Therefore, the ability to quickly prototype the solution is critical to timely and successful scientific discovery.

In order to accommodate these demands, the scientific community makes use of high-productivity programming languages and libraries. Particularly of interest are languages and libraries that support a declarative vector programming style; such as HPF[1], MATLAB[2], NumPy[3], Blitz++[4], and ILNumerics.Net[5].

In this context declarative means the ability to specify an operation, e.g. addition of two vectors, as a full-vector operation, $a + b$, instead of explicitly specifying looping and element-indexing: **for** i **in** $n : a[i]+b[i]$. Vector programming, also know as array programming, is of particular interest since full-vector operations are closer to the domain of the application-programmer.

The performance of a high-productivity programming language and/or library is often insufficient to handle problem sizes required in the research. Thus, we see the scientific community reimplement the prototype using another more high-performance framework, which exposes both the complexity and the performance-potential of the underlying hardware. This reimplementation is very time-consuming and a source of errors in the scientific code. Especially, when the computing environments are highly heterogeneous and require both parallelism and hardware architecture expertise.

Bohrium is a framework that circumvents the need for reimplementation completely. Instead of manually parallelizing the scientific applications for a specific hardware component, the Bohrium framework seamlessly interprets several high-productivity languages and libraries while transparently utilizing the parallel potential of the underlying hardware. The expressed goal of Bohrium is to achieve 80% of the achievable performance compared to a highly optimized implementation.

The version of Bohrium we present in this paper is a proof-of-concept that supports three languages; Python, C++, and Common Intermediate Language (CIL)[1], and three computer architectures, CPU, Cluster, and GPU. Bohrium defines an intermediate vector bytecode language specialized for the declarative vector programming model and provides a runtime environment for executing the bytecode. The intermediate vector bytecode makes Bohrium a retargetable framework where the front-end languages and the back-end architectures are fully interchangeable.

II. Related Work

The key motivation for Bohrium is to provide a framework for the utilization of diverse and complex computing systems with the goal of obtaining high-performance, high-productivity and high-portability, HP^3. Systems such as pyOpenCL/pyCUDA[6] provides tools for interfacing a high abstraction front-end language with kernels written for specific potentially exotic hardware. In this case, lowering the bar for harvesting the power of modern GPU's, by letting the user write only the GPU-kernels as text strings in the host language Python. The goal is similar to that of Bohrium – the approach however is entirely different. Bohrium provides a means to hide low-level target specific code behind a programming model and providing a framework and runtime environment to support it.

Bohrium is more closely related to the work described in [7], here a compilation framework, unPython, is provided for execution in a hybrid environment consisting of both CPUs and GPUs. The framework uses a Python/NumPy based front-end that uses Python decorators as hints to do selective optimizations. Bohrium performs data-centric optimizations on vector operations, which can be viewed as akin to selective optimizations, in the respect that we do *not* optimize the program as a whole. However, the approach used in the

[1]also known as Microsoft .NET

Bohrium Python interface we find much less obtrusive. All arrays are by default handled by Bohrium – No decorators are needed or used. This approach provides the advantage that any existing NumPy program can run unaltered and take advantage of Bohrium without changing a single line of code. Whereas unPython requires the user to manually modify the source code by applying hints in a manner similar to that of OpenMP. The proposed non-obtrusive design at the source level is to the author's knowledge novel.

Microsoft Accelerator [8] introduces ParallelArray, which is similar to the utilization of the NumPy arrays in Bohrium but there are strict limitations to the utilization of ParallelArrays. ParallelArrays does not allow the use of direct indexing, which means that the user must copy a ParallelArray into a conventional array before indexing. Bohrium instead allows indexed operations and additionally supports *vector-views*, which are vector-aliases that provide multiple ways to access the same chunk of allocated memory. Thus, the data structure in Bohrium is highly flexible and provides elegant programming solutions for a broad range of numerical algorithms. Intel provides a similar approach called Intel Array Building Blocks (ArBB) [9] that provides retargetability and dynamic compilation. It is thereby possible to utilize heterogeneous architectures from within standard C++. The retargetability aspect of Intel ArBB is represented in Bohrium as a plain and simple configuration file that define the Bohrium runtime environment. Intel ArBB provides a high performance library that utilizes a heterogeneous environment and hides the low-level details behind a declarative vector programming model similar to Bohrium. However, ArBB only provides access to the programming model via C++ whereas Bohrium is not limited to any one specific front-end language.

On multiple points Bohrium is closely related in functionality and goals to the SEJITS [10] project. SEJITS takes a different approach towards the front-end and programming model. SEJITS provides a rich set of computational kernels in a high-productivity language such as Python or Ruby. These kernels are then specialized towards an optimality criteria. The programming model in Bohrium does not provide this kernel methodology instead Bohrium deduce computational kernels at runtime by inspecting the flow of vector bytecode.

Bohrium provides in this sense a virtual machine optimized for execution of vector operations, previous work [11] was based on a complete virtual machine for generic execution whereas Bohrium provides an optimized subset.

III. Front-end Languages

To hide the complexities of obtaining high-performance from the diverse hardware making up modern compute systems any given framework must provide a meaningful high-level abstraction. This can be realized in the form of domain specific languages, embedded languages, language extensions, libraries, APIs etc. Such an abstraction serves two purposes: (1) It must provide meaning for the end-user such that the goal of high-productivity can be met with satisfaction. (2) It must provide an abstraction that consists of a sufficient amount of information for the system to optimize its utilization.

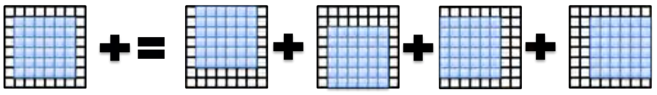


Fig. 1: A computation that makes use of views to implement a 5-point stencil.

Bohrium is not biased towards any specific choice of abstraction or front-end technology as long as it is compatible with the declarative vector programming model. This provides means to use Bohrium in functional programming languages, provide a front-end with a strict mathematic notation such as APL [12], or a more relaxed syntax such as MATLAB.

The declarative vector programming model encourages expressing programs in the form of high-level vector operations, e.g. by expressing the addition of two vectors using one high-level function instead of computing each element individually. Combined with vector slicing, also known as vector or matrix slicing [1], [2], [13], [14], the programming model is very powerful as a high-level, high-productive programming model (Fig. 1).

In this work, we will not introduce a whole new programming language, instead we will introduce *bridges* that integrate existing languages into the Bohrium framework. The current prototype implementation of Bohrium supports three popular languages: C++, .NET, and Python. Thus, we have three bridges – one for each language.

The C++ and .NET bridge provides a new array library for their respective languages that utilizes the Bohrium framework by mapping array operations to vector bytecode. The new array libraries do not attempt to be compatible with any existing libraries, but rather provide an intuitive interface to the Bohrium functionality. The Python bridge make use of NumPy, which is the de facto library for scientific computing in Python. The Python bridge implements a new version of NumPy that uses the Bohrium framework for all N-dimensional array computations.

Brief descriptions on how each one of the three bridges can be used is given in the following. The jacobi-stencil expression from Figure 1 is used as a running example for each language.

A. C++

The C++ bridge provides an interface to Bohrium as a domain-specific embedded language (DSEL) providing a declarative, high-level programming model. Related libraries and DSELs include Armadillo[15], Blitz++[4], Eigen[16] and Intel Array Building Blocks[9]. These libraries have similar traits; declarative programming style through operator-overloading, template metaprogramming and lazy evaluation for applying optimizations and late instantiation.

A key difference is that the C++ bridge applies lazy evaluation at runtime by delegating all operations on arrays to the Bohrium runtime environment. Whereas the other libraries apply lazy evaluation at compile-time via expression-templates. This is a general design-choice in Bohrium – evaluation is improved by a single component and not in every language-bridge. A positive side-effect of avoiding expression-templates

```
double solve(multi_array<double> grid,
             size_t iterations)
{
  multi_array<double> center,north,south,east,west;
  center  = grid[_(1,-1,1)][_(1,-1,1)];
  north   = grid[_(0,-2,1)][_(1,-1,1)];
  south   = grid[_(2, 0,1)][_(1,-1,1)];
  east    = grid[_(1,-1,1)][_(2, 0,1)];
  west    = grid[_(1,-1,1)][_(0,-2,1)];
  for(size_t i=0; i<iterations; ++i)
   center( 0.2*(center+north+east+west+south) );
}
```

Fig. 2: Jacobi stencil computation expressed in Bohrium C++.

in the C++ bridge are better compile-time error-messages for the application programmer.

Figure 2 illustrates the Jacobi-Stencil expressed in Bohrium/C++, a brief clarification of the semantics follow. Arrays along with the type of their containing elements are declared as "`multi_array<T>`". The function "`_(start, end, skip)`" creates a slice of every `skip` element from `start` to (but not including) `end`. The generated slice is then passed to the overloaded "`operator[]`" to create a segmented view of the operand. Overload of "`operator=`" creates aliases to avoid copying. To explicitly copy an operand the programmer must use a "`copy(...)`" function. Overload of "`operator()`" allows for updating an existing operand; as can been seen in the loop-body.

B. CIL

The NumCIL library introduces the declarative vector programming model to the CIL languages[17] and, like ILNumerics.Net, provides an array class that supports full-array operations. In order to utilize Bohrium, the CIL bridge extents NumCIL with a new Bohrium back-end.

The Bohrium extension to NumCIL, and NumCIL itself, is written in C# but with consideration for other languages. Example benchmarks are provided that shows how to use NumCIL with other popular languages, such as F# and IronPython. An additional IronPython module is provided which allows a subset of Numpy programs to run unmodified in IronPython with NumCIL. Due to the nature of the CIL, any language that can use NumCIL can also seamlessly utilize the Bohrium extension. The NumCIL library is designed to work with an unmodified compiler and runtime environment and supports Windows, Linux and Mac. It provides both operator overloads and function based ways to utilize the library.

Where the NumCIL library executes operations when requested, the Bohrium extension uses both lazy evaluation and lazy instantiation. When a side-effect can be observed, such as accessing a scalar value, any queued instructions are executed. To avoid problems with garbage collection and memory limits in CIL, access to data is kept outside CIL. This allows lazy instantiation, and allows the Bohrium runtime to avoid costly data transfers.

The usage of NumCIL with the C# language is shown in Figure 3. The `NdArray` class is a typed vesion of a general multidimensional array, from which multiple views can be extracted. In the example, the `Range` class is used to extract

```
using NumCIL.Double;
using R = NumCIL.Range;

double Solve(NdArray grid, int iterations)
{
  var center = grid[R.Slice(1,-1), R.Slice(1,-1)];
  var north  = grid[R.Slice(0,-2), R.Slice(1,-1)];
  var south  = grid[R.Slice(2, 0), R.Slice(1,-1)];
  var east   = grid[R.Slice(1,-1), R.Slice(2, 0)];
  var west   = grid[R.Slice(1,-1), R.Slice(0,-2)];

  for (var i = 0; i < iterations; i++)
   center[R.All]=0.2*(center+north+east+west+south);
}
```

Fig. 3: Jacobi stencil computation expressed in NumCIL C#.

```
solve(grid, iterations):
  center = grid[1:-1,1:-1]
  north  = grid[-2:,1:-1]
  south  = grid[2:,1:-1]
  east   = grid[1:-1,:2]
  west   = grid[1:-1,2:]
  for i in xrange(iterations):
    center[:] += 0.2*(north+south+east+west)
```

Fig. 4: Jacobi stencil computation expressed in Python/Numpy.

views on a common base. The notation for views is influenced by Python, in which slices can be expressed as a three element tuple of offset, length and stride. If the stride is omitted, as in the example, it will have the default value of one. The length will default to zero, which means “the rest”, but can also be set to negative numbers which will be intepreted as “the rest minus N elements”. The benefits of this notation is that it becomes possible to express views in terms of relative sizes, instead of hardcoding the sizes.

In the example, the one line update, actually reads multiple data elements from same memory region and writes it back. This use of views simplifies concurrent access and removes all problems related to handling boundary conditions and manual pointer arithmetics. The special use of indexing on the target variable is needed to update the contents of the variable, instead of replacing the variable.

C. Python

The Python Bridge is an extension of the scientific Python library, NumPy version 1.6 (Fig. 4). The Bridge seamlessly implements a new array back-end for NumPy and uses *hooks* to divert function call where the program access Bohrium enabled NumPy arrays. The hooks will translate a given function into its corresponding Bohrium bytecode when possible. When it is not possible, the hooks will feed the function call back into NumPy and thereby forcing NumPy to handle the function call itself. The Bridge operates with two address spaces for arrays: the Bohrium space and the NumPy space. The user can explicitly assign new arrays to either the Bohrium or the NumPy space through a new array creation parameter. In two circumstances, it is possible for an array to transfer from one address space to the other implicitly at runtime.

1) When an operation accesses an array in the Bohrium address space but it is not possible for the bridge to translate the operation into Bohrium bytecode. In this

case, the bridge will synchronize and move the data to the NumPy address space. For efficiency no data is actually copied instead the bridge uses the `mremap` function to re-map the relevant memory pages when the data is already present in main memory.

2) When an operations access arrays in different address spaces the Bridge will transfer the arrays in the NumPy space to the Bohrium space.

In order to detect direct access to arrays in the Bohrium address space by the user, the original NumPy implementation, a Python library or any other external source, the bridge protects the memory of arrays that are in the Bohrium address space using `mprotect`. Because of this memory protection, subsequently accesses to the memory will trigger a segmentation fault. The Bridge can then handle this kernel signal by transferring the array to the NumPy address space and cancel the segmentation fault. This technique makes it possible for the Bridge to support all valid Python/NumPy application since it can always fallback to the original NumPy implementation.

Similarly to the other Bridges, the Bohrium Bridge uses lazy evaluation where it records instruction until a side-effect can be observed.

IV. The Bohrium Runtime System

The key contribution in this work is a framework, Bohrium, that significantly reduces the costs associated with high-performance program development. Bohrium provides the mechanics to couple a programming language or library with an architecture-specific implementation seamlessly.

Bohrium consists of a number of components that communicate by exchanging a *Vector Bytecode*. Components are allowed to be architecture-specific but they are all interchangeable since all uses the same communication protocol. The idea is to make it possible to combine components in a setup that match a specific execution environment. Bohrium consist of the following three component types (Fig. 5):

Bridge The role of the Bridge is to integrate Bohrium into existing languages and libraries. The Bridge generates the Bohrium bytecode that corresponds to the user-code.

Vector Engine Manager (VEM) The role of the VEM is to manage data location and ownership of vectors. It also manages the distribution of computing jobs between potentially several Vector Engines, hence the name.

Vector Engine (VE) The VE is the architecture-specific implementation that executes Bohrium bytecode.

When using the Bohrium framework, at least one implementation of each component type must be available. However, the exact component setup depend on the runtime system and what hardware to utilize, e.g. executing NumPy on a single machine using the CPU would require a Bridge implementation for NumPy, a VEM implementation for a machine node, and a VE implementation for a CPU. Now, in order to utilize a GPU instead, we can exchange the CPU-VE with a GPU-VE without having to change a single line of code in the NumPy application. This is exactly the key contribution of Bohrium – the ability to change the execution hardware without changing the user application.

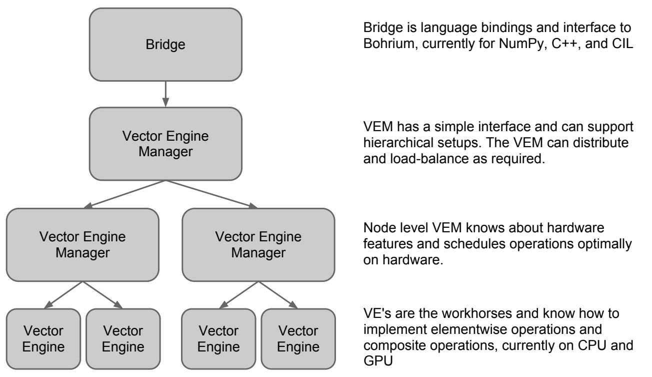

Bridge
Vector Engine Manager
Vector Engine Manager
Vector Engine Manager
Vector Engine
Vector Engine
Vector Engine
Vector Engine
Bridge is language bindings and interface to Bohrium, currently for NumPy, C++, and CIL
VEM has a simple interface and can support hierarchical setups. The VEM can distribute and load-balance as required.
Node level VEM knows about hardware features and schedules operations optimally on hardware.
VE's are the workhorses and know how to implement elementwise operations and composite operations, currently on CPU and GPU

Fig. 5: Bohrium Overview

```
# Bridge for NumPy
[numpy]
type = bridge
children = node

# Vector Engine Manager for a single machine
[node]
type = vem
impl = libbh_vem_node.so
children = gpu

# Vector Engine for a GPU
[gpu]
type = ve
impl = lbbh_ve_gpu.so
```

Fig. 6: This example configuration provides a setup for utilizing a GPU on one machine by instructing the Vector Engine Manager to use the GPU Vector Engine implemented in the shared library `lbhvb_ve_gpu.so`.

A. Configuration

To make Bohrium as flexible a framework as possible, we manage the setup of all the components at runtime through a configuration file. The idea is that the user or system administrator can specify the hardware setup of the system through a ini-file (Fig. 6). Thus, it is just a matter of editing the configuration file when changing or moving to a new hardware setup and there is absolutely no need to change the user applications.

B. Vector Bytecode

A vital part of Bohrium is the *Vector Bytecode* that constitute the link between the high-level user language and the low-level execution engine. The bytecode is designed with the declarative vector programming model in mind where the bytecode instructions operate on input and output vectors. To avoid excessive memory copying, the vectors can also be shaped into multi-dimensional vectors. These reshaped vector views are then not necessarily comprised of elements that are contiguous in memory. Each dimension is described with a stride and size, such that any regularly shaped subset of the

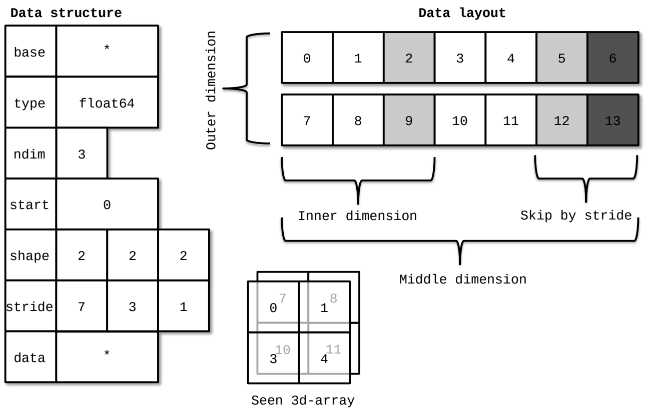


Fig. 7: Descriptor for n-dimensional vector and corresponding interpretation

underlying data can be accessed. We have chosen to focus on a simple, yet flexible, data structure that allows us to express any regularly distributed vectors. Figure 7 shows how the shape is implemented and how the data is projected.

The aim is to have a vector bytecode that support data parallelism implicitly and thus makes it easy for the bridge to translate the user language into the bytecode efficiently. Additionally, the design enables the VE to exploit data parallelism through SIMD[2] and the VEM through SPMD[3].

In the following we will go through the four types of vector bytecodes in Bohrium.

1) Element-wise: Element-wise bytecodes performs a unary or binary operation on all vector elements. At the moment Bohrium supports 53 element-wise operations, e.g. addition, multiplication, square root, equal, less than, logical and, bitwise and, etc. For element-wise operations, we only allow data overlap between the input and the output vectors if the access pattern is the same, which, combined with the fact that they are all stateless, makes it straightforward to execute them in parallel.

2) Reduction: Reduction bytecodes reduce an input dimension using a binary operator. Again, we do not allow data overlap between the input and the output vectors and the operator must be associative. At the moment Bohrium supports 10 reductions, e.g. addition, multiplication, minimum, etc. Even though none of them are stateless, the reductions are all straightforward to execute in parallel because of the non-overlap and associative property.

3) Data Management: Data Management bytecodes determines the data ownership of vectors. It consists of a synchronization operator that orders a child component to place the vector data in the address space of its parent component; a free operator that orders a child component to free the data of a given vector in the global address space; and a discard operator that orders a child component free the meta-data associated a given vector, and signals that any local copy of the data is invalidated.

The three bytecodes enable lazy allocation where the actual vector data allocation is delayed until it is used. Often vectors are created with a generator (e.g. random, constants) or with no data (e.g. temporary), which may exist on the computing device exclusively. Thus, lazy allocation may save several memory allocations and copies.

4) User-extensions: The above three types of bytecode makes up the bulk of a Bohrium execution. However, the three types of bytecode do not constitute a Turing complete instruction set. It is clear that the instruction set produces executions with a deterministic finite executing time and memory use. Thus, it cannot be Turing complete.

In order to handle operations that would otherwise be impossible, we introduce the fourth type of bytecode: user-extensions. We impose no restrictions to this generic operation; the extension writer has total freedom. However, Bohrium do not guarantee that all components support the operation. Initially, the user registers the user-extension with paths to all component-specific implementations of the operation. The user then receives a new handle for this *user-defined* bytecode and may use it subsequently. Matrix multiplication is a special example of a user-extension. A CPU specific implementation could simply call a native BLAS library and a Cluster specific implementation could call the ScaLAPACK library[18].

C. Bridge

The Bridge component is the *bridge* between the programming interface, e.g. Python/NumPy, and the VEM. The Bridge is the only component that is specifically implemented for the user programming language. In order to add Bohrium support to a new language or library, only the bridge component needs to be implemented. The bridge component generates bytecode based on the user application and sends them to the underlying VEM.

D. Vector Engine Manager

Rather than allowing the Bridge to communicate directly with the Vector Engine, we introduce a Vector Engine Manager into the design. The VEM is responsible for one memory address space in the hardware configuration. The current version of Bohrium implements two VEMs: the Node-VEM that handles the local address space of a single computer and the Cluster-VEM that handles the global distributed address space of a computer cluster.

The Node-VEM is very simple since the hardware already provides a shared memory address space; hence, the Node-VEM can simply forward all instruction from its parent to its child component. The Cluster-VEM, on the other hand, has to distribute all vectors between Node-VEMs in the cluster.

1) Cluster Architectures: In order to utilize scalable architectures fully, distributed memory parallelism is mandatory. The current Cluster-VEM implementation is quite naïve; it uses the bulk-synchronous parallel model[19] with static data decomposition and no communication latency hiding.

Bohrium implements all communication through the MPI-2 library and use a process hierarchy that consists of one master-process and multiple slave-processes. The master-process ex-

[2]Single Instruction, Multiple Data
[3]Single Program, Multiple Data

ecutes a regular Bohrium setup with the Bridge, Cluster-VEM, Node-VEM, and VE. The slave-processes, on the other hand, execute the same setup but without the Bridge and thus without the user applications. Instead, the master-process will broadcast vector bytecode and vector meta-data to the slave-processes throughout the execution of the user application.

Bohrium use a data-centric approach where a static decomposition dictates the data distribution between the MPI-processes. Because of this static data decomposition all processes have full knowledge of the data distribution and need not exchange data location meta-data. Furthermore, the task of computing vector operations is also statically distributed which means that all processes can calculate exactly what it needs to send, receive, and compute. Meta-data communication is only needed when broadcasting vector bytecode and creating new vectors – a task that has a asymptotes complexity of $O(\log_2 n)$

E. Vector Engine

The Vector Engine (VE) is the only component that actually does the computations, specified by the user application. It has to execute instructions it receives in an order that comply with the dependencies between instructions. Furthermore, it has to ensure that its parent VEM has access to the results as governed by the Data Management bytecodes.

1) CPU: The CPU-VE utilizes a single core on a regular CPU using a straightforward implementation that handles instructions in a sequential manner without any instruction reordering. The implementation is in C++ and uses *templated* for-loops to implement vector operations. A precompiled switch interprets the vector bytecode and invokes the for-loop with the relevant type signature. All operations are inlined as *functors* thus removing the need for function calls within the for-loop.

2) GPU: To harness the computational power of the modern GPU we have created the GPU-VE for Bohrium. Since Bohrium imposes a vector oriented style of programming on the user, which directly maps to data-parallel execution, Bohrium byte code is a perfect match for a modern GPU.

We have chosen to implement the GPU-VE in OpenCL over CUDA. This was the natural choice since one of the major goals of Bohrium is portability, and OpenCL is supported by more platforms.

The GPU-VE currently uses a simple kernel building and code generation scheme: It will keep adding instructions to the current kernel for as long as the shape of the instruction output matches that of the current kernel. Input parameters are registered so they can be read from global memory. Similarly, output parameters are registered to be written back to global memory.

The GPU-VE implements a simple method for temporary vector elimination when building kernels:

- If the instruction input is already read by the kernel, or it is generated within the kernel it will not be read from global memory.
- If the instruction output is not need later in the instruction sequence – signaled by a discard – it will not be written back to global memory.

Machine:	8-node Cluster	GPU Host
Processor:	AMD Opteron 6272	AMD Opteron 6274
Clock:	2.1 GHz	2.2 GHz
L3 Cache:	16MB	16MB
Memory:	128GB DDR3	128GB DDR3
Compiler:	GCC 4.6.3	GCC 4.6.3 & OpenCL 1.1
GPU:	N/A	Nvidia GeForce GTX 680 (2GB DDR5)
Software:	Linux 3.2, Mono Compiler 2.10, Python 2.7, NumPy 2.6, Blitz++ 0.9	

TABLE I: Machine Specifications

This simple scheme has proven very efficient. However, the efficiency is closely linked to the ability of the bridge to send discards close to the last usage of an vector.

The code generation we have in the GPU-VE simply translates every Bohrium instruction into exactly one line of OpenCL code.

V. Preliminary Results

In order to demonstrate our Bohrium design we have implemented a basic Bohrium setup. This concretization of Bohrium is by no means exhaustive but only a proof-of-concept implementation. It supports the three popular languages: C++, CIL, and Python, and the three computer architectures: CPU, GPU, and Cluster. All of which are preliminary implementations that have a high degree of further optimization potential. We conduct a preliminary performance study of the implementation that consists of the following three representative scientific application kernels:

Shallow Water A simulation that simulates a system governed by the shallow water equations. A drop is placed in a still container and the water movement is simulated in discrete time steps. It is a Python/NumPy implementation of a MATLAB application by Burkardt [20]. We use this benchmark for studying the Python/NumPy performance in Bohrium. We compare the performance of the same implementation using the Bohrium back-end and the native NumPy back-end.

Black Scholes The Black-Scholes model is a partial differential equation, which is used in finance for calculating price variations over time. In order to study the performance of Bohrium in C++, we compare two C++ implementations of this benchmark, one using Bohrium and one using Blitz++. The two implementations are very similar and both uses vector operations almost exclusively.

N-Body A Newtonian N-body simulation is one that studies how bodies, represented by a mass, a location, and a velocity, move in space according to the laws of Newtonian physics. We use a straightforward algorithm that computes all body-body interactions, $O(n^2)$, with collisions detection. It is a C# implementation that uses the NumCIL vector library[17]. We compare the performance of the implementation using the Bohrium back-end and the native NumCIL back-end.

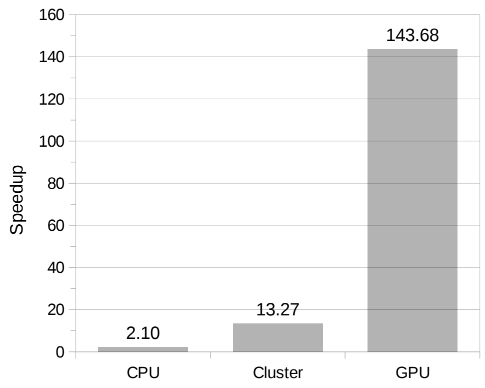


Fig. 8: Relative speedup of the Shallow Water application. For the CPU and Cluster, the application simulates a 2D domain with 25k^2 value points in 10 iterations. For the GPU, it is a 4k^2 domain in 100 iterations.

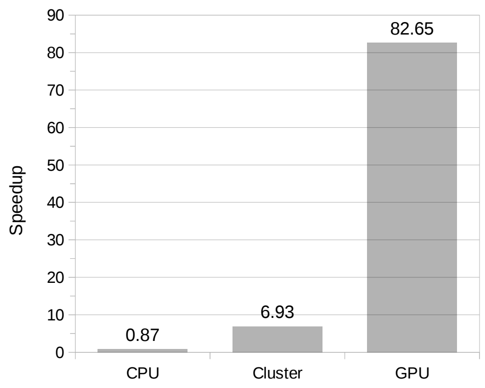


Fig. 9: Relative speedup of the Black Scholes application. For the CPU and Cluster, the application generates 10m element vectors using 10 iterations. For the GPU, it generates 64m element vectors using 50 iterations.

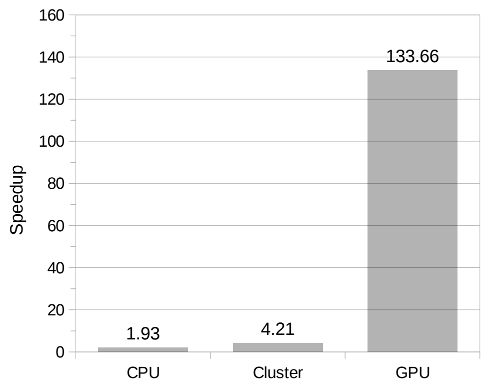


Fig. 10: Relative speedup of the N-Body application. For the CPU and Cluster, the application simulates 15k bodies in 10 iterations. For the GPU, it is 3200 bodies and 50 iterations.

We execute all three applications using three different hardware setups: one using a single CPU, one using an eight cluster-nodes, and one using a GPU. The single CPU setup uses one of the cluster-nodes whereas the GPU setup uses a new machine (Table I). For each benchmark/language we compare the Bohrium execution with the baseline execution and calculate the speedup based on the average wall clock time of five executions. When executing on the single CPU, we use one CPU-core likewise when executing on the eight-node cluster, we use one CPU-core per node. The input and output data is 64bit floating point for all executions.

A. Discussion

The Shallow Water application is memory intensive and uses many temporary vectors. This is clear when comparing the Bohrium execution with the Native NumPy execution on a single CPU. The Bohrium execution is 2.10 times faster than the Native NumPy execution primarily because of memory allocation reuse. The Cluster setup demonstrates good scalable performance as well. Even without communication latency hiding, it achieves a speedup of 6.37 compared to the single CPU setup and 13.27 compared to Native NumPy. Finally, the GPU shows an impressive 143.68 speedup, which demonstrates the efficiency of parallelizing vector operations on a vector machine.

The Black Scholes application is computation intensive and embarrassedly parallel, which is evident in the benchmark result. The cluster setup achieve a speedup of 6.93 compared to the Blitz++ and an almost linearly speedup of 7.93 compared to the single CPU. The GPU achieves a slightly lower speedup mainly due to the fact, that it is compared to a faster baseline i.e. Blitz++ is faster than NumPy. Resulting in a speedup of 82.65 on the GPU.

The Cluster performance of the N-Body application is not very impressive – a speedup of 4.21 compared to NumCIL and 2.18 compared to Bohrium using a single CPU. The problem is the use of communication intensive transpose operations that translate into *all-to-all* communication and hurts scalability. Especially, since the Cluster does not implement communication latency hiding. The Bohrium execution shows a speedup of 1.93 compared to the NumPy execution on a single CPU. This is because Bohrium uses inline function calls when traversing a computation loop – a technique that is not possible in the CIL framework. Finally, the GPU demonstrate a good speedup of 133.66 compared to NumCIL.

VI. Future Work

From the experiments we can see that the performance is generally quite good. There is room for improvement when distributing transposed data in the cluster, which is a deficiency that stems from the need to broadcast all elements to all nodes. This problem is not trivial to solve but we have previously shown an efficient solution in the DistNumPy[21], [22] project.

Despite the good results, we are convinced that we can improve these results significantly. We are currently working on an internal representation for bytecode dependencies, which will enable us to rearrange the instructions and eliminate the use of temporary storage. In the article describing Intel Array Building Blocks, they report that the removal of temporary arrays is the single optimization that yields the greatest performance improvement. Internal testing with manual removal of temporary storage shows an order of magnitude improvement, even for simple benchmarks.

The GPU vector engine already uses a simple scanning algorithm that detects some instances of temporary vectors usage, as that is required to avoid exhausting the limited GPU memory. However, the internal representation will enable a better detection of temporary storage, but also enable loop detection and improve kernel generation and kernel reusability.

This internal representation will also allow pattern matching, that will allow selective replacement of parts of the instruction stream with optimized versions. This can be used to detect cases where the user is calculating a scalar sum, using a series of reductions, or detect matrix multiplications. By implementing efficient micro-kernels for known computations, we can improve the execution significantly.

As these kernels are implemented, it is simple to offer them as function calls in the bridges. The bridge implementation can then simply implement the functionality by sending a precoded sequence of instructions.

We are also investigating the possibility of implementing a Bohrium Processing Unit, BPU, on FPGAs. With a BPU, we expect to achieve performance that rivals the best of todays GPUs, but with a lower power consumption. As the FPGAs come with a built-in Ethernet socket, they can also provide significantly lower latency, possibly providing real-time data analysis.

Finally, the ultimate goal of the Bohrium project is to support clusters of heterogeneous computation nodes where components specialized for GPUs, NUMA[4] aware multi-core CPUs, and Clusters, work together seamlessly.

VII. Conclusion

The declarative vector-programming model used in Bohrium provides a framework for high-performance and high-productivity. It enables the end-user to execute vectorized applications on a broad range of hardware architectures efficiently without any hardware specific knowledge. Furthermore, the Bohrium design supports scalable architectures such as clusters and supercomputers. It is even possible to combine architectures in order to exploit hybrid programming where multiple levels of parallelism exist, which is essential when fully utilizing supercomputers such as the Blue Gene/P[23].

In this paper, we introduce a proof-of-concept implementation of Bohrium that supports three front-end languages – Python, C++ and the .Net – and three back-end hardware architectures – single-core CPUs, distributed memory Clusters, and GPUs. The preliminary results are very promising – a *Shallow Water* simulation achieves 143.68 speedup when comparing a Native NumPy execution and a Bohrium execution that utilize the GPU back-end.

[4]Non-Uniform Memory Access

References

[1] D. Loveman, "High performance fortran," *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 1, no. 1, pp. 25–42, 1993.

[2] W. Yang, W. Cao, T. Chung, and J. Morris, *Applied numerical methods using MATLAB*. Wiley-Interscience, 2005.

[3] T. Oliphant, *A Guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[4] T. Veldhuizen, "Arrays in Blitz++," in *Computing in Object-Oriented Parallel Environments*, ser. Lecture Notes in Computer Science, D. Caromel, R. Oldehoeft, and M. Tholburn, Eds. Springer Berlin Heidelberg, 1998, vol. 1505, pp. 223–230.

[5] "Ilnumerics," http://ilnumerics.net/, [Online; accessed 12 March 2013].

[6] A. Klckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A scripting-based approach to GPU runtime code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157 – 174, 2012.

[7] R. Garg and J. N. Amaral, "Compiling python to a hybrid execution environment," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 19–30.

[8] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program gpus for general-purpose uses," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 325–335, Oct. 2006.

[9] C. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, "Intel's array building blocks: A retargetable, dynamic compiler and embedded language," in *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, 2011, pp. 224–235.

[10] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox, "Sejits: Getting productivity and performance with selective embedded jit specialization," *Programming Models for Emerging Architectures*, 2009.

[11] R. Andersen and B. Vinter, "The scientific byte code virtual machine," in *GCA'08*, 2008, pp. 175–181.

[12] K. E. Iverson, *A programming language*. New York, NY, USA: John Wiley & Sons, Inc., 1962.

[13] B. Mailloux, J. Peck, and C. Koster, "Report on the algorithmic language algol 68," *Numerische Mathematik*, vol. 14, no. 2, pp. 79–218, 1969. [Online]. Available: http://dx.doi.org/10.1007/BF02163002

[14] S. Van Der Walt, S. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[15] C. Sanderson *et al.*, "Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments," Technical report, NICTA, Tech. Rep., 2010.

[16] "Eigen," http://eigen.tuxfamily.org/, [Online; accessed 12 March 2013].

[17] K. Skovhede and B. Vinter, "NumCIL: Numeric operations in the Common Intermediate Language," *Journal of Next Generation Information Technology*, vol. 4, no. 1, 2013.

[18] L. S. Blackford, "ScaLAPACK," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing 96 Supercomputing 96*, 1996, p. 5.

[19] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[20] J. Burkardt, "Shallow water equations," people.sc.fsu.edu/\~jburkardt/m_src/shallow_water_2d/, [Online; accessed March 2010].

[21] M. R. B. Kristensen and B. Vinter, "Numerical python for scalable architectures," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10. New York, NY, USA: ACM, 2010, pp. 15:1–15:9.

[22] M. R. B. Kristensen, Y. Zheng, and B. Vinter, "Pgas for distributed numerical python targeting multi-core clusters," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 680–690, 2012.

[23] M. Kristensen, H. Happe, and B. Vinter, "GPAW Optimized for Blue Gene/P using Hybrid Programming," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009, pp. 1–6.

6.6 Just-In-Time Compilation of NumPy Vector Operations

DOI 10.7603/s40601-013-0021-1

GSTF Journal on Computing (JoC), Vol. 3 No. 3, Dec 2013

Just-In-Time Compilation of NumPy Vector Operations

Johannes Lund, Mads R. B. Kristensen, Simon A. F. Lund, and Brian Vinter
Niels Bohr Institute, University of Copenhagen, Denmark
jolu@diku.dk and {madsbk/safl/vinter}@nbi.dk

***Abstract*—In this paper, we introduce JIT compilation for the high-productivity framework Python/NumPy in order to boost the performance significantly. The JIT compilation of Python/NumPy is completely transparent to the user – the runtime system will automatically JIT compile and execute the NumPy instructions encountered in a Python application. In other words, we introduce a framework that provides the high-productivity from Python while maintaining the high-performance of a low-level, compiled language.**

We transforms NumPy vector instruction into an Abstract Syntax Tree representation that creates the basis for further optimizations. From the AST we auto-generate C code which we compile into computational kernels and execute. These incorporate temporary array removal and loop-fusion which are main benefactors in the achieved speedups. In order to amortize the overhead of creation, we also implement a cache for the compiled kernels.

We evaluate the JIT compilation by executing several scientific computing benchmarks on an AMD. Compared to NumPy, we achieve speedups of a factor 4.72 for a N-Body application and 7.51 for a Jacobi Stencil application executing on a single CPU core.

Keywords*—*JIT, automatic, dynamic, runtime

I. Introduction

Many scientific algorithms can be expressed by using vector operation and linear algebra. These are easily expressed in specialized high-level languages such as the NumPy library for Python. However, their performance is often significantly lower than when implemented and computed in a low-level language. Using the high-level languages for prototyping and re-implementing the found solution in a low level language when required to run on actual-size data.

Expressing the data and calculations efficiently in a low-level language such as C is far from being a trivial task. It requires an in-depth understanding to implement this efficiently on heterogeneous hardware architectures.

We wish to bridge the gap between the two extremes, by allowing scientists to express their problems in a favorable high-level language and at the same time achieve the performance of a complex low-level language implementation. Thus, the goal of this paper is to improve the performance of Python/NumPy applications to a degree that makes it similar to low-level languages such as C or C++. We do not expect it to out-perform hand-optimized C code. As long as it demonstrates similar performance while retaining the high-productivity of the Python language, we are satisfied.

In order to improve the performance of Python/NumPy, we introduce a Just-In-Time (JIT) compiler backend for the NumPy library. In order to hook into the NumPy library we make us of the Bohrium runtime system [1], which translate NumPy vector operations into an intermediate vector bytecode suitable for JIT compilation. Because Python is an interpreted language, we use lazy evaluation of vector instructions in order to have multiple instructions available to analyze, optimize, and JIT compile.

The following methods constitute the key contributions for the performance improvement of Python/NumPy applications using out JIT compiler backend:

- Removal of temporary arrays
- Loop fusion
- Compiled kernel caching

II. Related Work

The key motivation for our JIT back-end is to automatically transform high-level Python/NumPy applications to complied executable kernels, with the goal of obtaining high-performance, high-productivity and high-portability, HP^3.

Our work is closely related to the work described in [2] where a compilation framework, unPython, complies Python code into C. The framework uses Python decorators as hints to do selective optimizations. Particularly, the user must annotate variables with C data types. Because of the Bohrium runtime system, our JIT backend does not require any modifications to the Python code.

Systems such as pyOpenCL/pyCUDA [3] provides tools for interfacing with the OpenCL and CUDA framework directly from Python. They lower the bar for harvesting the power of modern systems by letting the user write CPU or GPU kernels as text strings in Python. Still, the user need knowledge of the underlying hardware and must modify existing Python code in order to utilize them.

III. Analysis

In this section we present an analysis of the requirements and solutions for using JIT compilation of NumPy vector instruction. There are many different elements required in framework for JIT compilation, which all must be designed and implemented.

DOI: 10.5176/2251-3043_3.3.270

Published online: 28 February 2014

A. Creating composite expressions

At runtime we have a list of vector instructions available which contain information about the relation between the instruction. We can use this relational information to build composite expressions and create computational kernels based on these.

In this subsection we investigate methods to extract and analyze the information from the instruction list.

1) Naive approach: The naive approach involves folding the instruction into larger expression. Examining the instruction list inorder by comparing the output of a instruction with the input of the next it can be determined if the first is a subexpression of the later. As many expression are organized as a chain of instructions this naive method would work well on many of the expressions.

Creating composite expressions and computational kernels directly from the list would be straight forward. In the case where the output is not uesd a new composite expression is build. For each of the following expression which uses the previous output in its input the expression grows by substituting the new input with the expression. With this approach the code for the kernel could be created directly in the first passthrough of the instructionlist.

With the order of execution the same as for the instruction the data-dependencies between the instruction are not relevant.

This approach only handles relations in chains and would not combine expression where both the left and right inputs where a result of prior instruction. The only information used would be the one found between two instructions. We have information about the relation between all instruction in the list and we should use this.

2) Abstract Representation: The abstract approach targets the shortcomings of the naive approach and is the method used. This is initially done by splitting the creation of kernels from the data extraction and analysis. The instructions list is translated into a abstract representation where the information between all instructions are represented.

The expressions are created as Abstract Syntax Trees (AST) from on the mathematical expressions from the instruction list. The transformation from a batch of instruction results in a set of AST's which are later converted into computational kernels.

It is a more complex solution compared to the above and requires the use of compiler techniques such as Static Single Assignments (SSA) and creation of dependency graphs. We use the AST as our working representation of the instructions for the following reasons:

1) Not sensitive to the order of instructions but the semantic meaning. Semantically equal expression result in the same AST's or can easily be transformed to it.
2) The tree data structure is well known and easy to work with, analyze and optimize.
3) In the creation of an AST temporary arrays are syntacticly removed.
4) The general structure can be used to represent more complicated AST's, which ensure that later extensions to the form is possible.

B. Execution Orchestration

With the change of representation from a list of instruction into a set expression it is needed to determine how these are to be executed. In the list the order was given but with the AST's the choice is not as simple. We present different orchestration methods for the AST and the resulting kernels and argues for the methods used in our solution.

The orchestration of the AST's for execution can take the form of a list or a graph

- The list execution follows the sequential execution of the kernel of the AST's. The AST's are all rooted to array assignment, which originally is represented as a instruction. The AST's are arranged and executed in this order. This approach is well suited for single core execution as all operations must be performed in s sequential order. It can be seen a flattened graph, as the graph information is available within the AST's.
- Orchestration based on the dependencies between the AST, represented as a graph. The dependencies between AST results in sequential paths. The graph will represent the relation between the AST. From this it can be determined if AST's are independent each other and thus if they can be executed in parallel.

The List model is chosen for it simplicity and that the framework target is the a single CPU core.

The relation between the AST's have purposes in different optimization method which are relevant for both single- and multi-core scenarios. Within the AST's the relational information is used to discover dependency violations and to secure correctness among the AST's. The correct inital dependencies is vital om checking data-dependencies spanning multiple the AST's.

C. Representation of Kernel Function.

To execute the AST's we must represent them in the form of a programing language which we can execute. This could in principle be any language, but there is a clear demand for a highly efficient language, which narrows down the field of candidate. The considered options where the following:

- C/C++
- Assembler

The C languages was chosen as it is supported everywhere and can be very efficient. The choice of language is connected to the choice in compiler as well. For the C language there is a number of compilers available. The kernels are pretty simple and require no extended functionality for which C++ would be of value.

Using Assembler to create the kernels would move the implementation closer to the metal then with C and potentially perform better. The kernels are rather simple as they consists of a traversal of a multidimensional array and an equation. For the simple uses the assembler version would be fairly straightforward to implement.

Being close the metal also has its drawbacks. In order to take advantage of the different architectures their special instructions must be used, which requires multiple implementation to work efficiently in a heterogeneous hardware environment.

The machine code produced by modern compilers is very efficient. The newest advantages in CPU design is integrated in these optimization which can include the use of vendor specific optimization, SSE instruction, function in-lining or even loop-unrolling. These and many other optimization are available in modern compilers such as GCC [4] and C-LANG/LLVM [5]. Achieving these benefits with machine code implemented kernels would is not practical solution.

An alternative to native code is using Intermediate Language (IL) as used internally in LLVM. The AST's and traversal would be expressed in IL language and compiled with LLVM. With this comes the possibilities to apply specific optimization to the code in the compilation phase.

A second alternative could be OpenCL code which would be targeted the CPU. The language is based on C99 with a few extensions and can be compiled to both CPU's and GPU's. The OpenCL framework target Multi- and Many-Core architectures where concurrency and parallelism is in focus.

C is chosen for the kernel representation it is best suited for the task and it will be reasonable fast to investigate future optimizations to the kernels.

D. Kernel Compilation

The choice of compiler is strongly coupled with the choice of language. With C chosen there are still different approaches to take on compilation.

- Command-line compilation and dynamic linking: Write the kernel program to a file which is then used compiled with a compiler from the command-line,
- In memory compilation: Compile from a library function where the kernel function code is read as strings and the results is a function-pointer.

The command-line method is simple approach as most linux systems has access to a C compiler such as GCC. It is required to write the function code to a file as this is the input. By compiling the code into a shared object file it can be linked into the running program. This is done with the ldopen() function.

The method of using a in memory compiler eliminated the need to perform disk I/O operations and system calls. It is all handled in memory and within the program execution. The Tiny C Compiler [6] (TCC), is such a library which offers the ability to compile a string of C code into a machine code and return a function-pointers to the functions compiled. TCC is very small and very simple library which is easy to use. Unfortunately the quality of the resulting machine code is far from that GCC.

The library for C-lang with is part of LLVM. Here the C code would be read and compiled into the LLVM IL language and from this into machine code using the LLVM backend to do the compilation. The LLVM and C-lang libraries are both very large and complex API's to work with.

TCC was initially used but replaced with the GCC as it became clear that the performance was an issue. It here became clear that the quality of code is more important then the compile time. The initial investigation into LLVM revealed a large and complex framework, of which only the compilation part was of interest. As GCC and LLVM generally produces code of equal quality [7] the expected outcome of using LLVM over GCC is to reduce the compilation time and make the implementation prettier.

The GCC method is chosen as the kernel compiler due to its simple approach, availability and execution performance.

E. Cache

Caching in the JIT framework is related to the computational kernels. Many of the same instructions are reoccurring, as a result of loops in the host programs, the same AST's are created and thus the same kernels. The use of caches is based on assumption that each kernel is multiple times, which is the case in most scientific NumPy applications.

The reason to use a cache for the JIT compiled kernels is to reduce the time required in creation and compilation of the kernels for every AST. There will be a overhead of creating the kernels but the overall effect can be reduced significantly by using a cache for the kernels.

The number of unique kernels depends on the number of uniqe AST's created. We dont expect a large number of kernels as many of the scientific applications uses the same computations multiple times. Running the Shallow Water benchmark, which produces the most kernels of the benchmarks used, only 11 kernels are created in total.

We have decided to use a non-persited cache for the kernels where the kernels are created and used as the program executes. The on-the-fly strategy fits the needs of the JIT framework very well. Creating the kernels is fast compared to the execution time and only a few must be created. The small time difference between loading the kernels or creating them is insignificant compared to the runtime.

The benifit from the cache is the large number of identical kernels fetched from memory apposed to being created.

IV. Design and Implementation

The goal of the system is to transform the list of instructions unary og binary instructions into computational kernels and execute these instead a series of individual instructions.

We described the various parts in the analysis section and here define four phases in which we organized the various parts. The phases, which are illustrated in Figure 1, are:

1) AST creation: Information gathering, analysis and creation of composite expressions.
2) Optimize and Orchestrate: Determining the flow of the execution and perform optimizations on its abstract form.
3) Kernel creation: Code generation and compilation.
4) Execution and caching.

In the first phase the instruction list is transformed into a set of AST's which are organized in a Nametable. The initial AST's are analyzed for basearray dependencies as these not reflected in the Nametable after the initial creation. After the dependency corrections have been performed the collection of AST's is a valid representation of the instruction list.

In phase two the forest of AST's are orchestrated into an execution list. This phase would be place for AST-based analysis

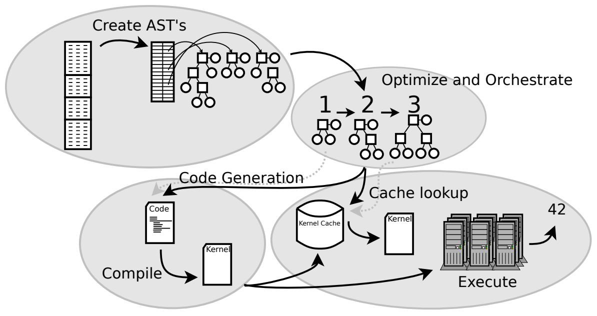


Fig. 1. JIT Framework design overview.

```
Assignment: Array = Expr

Exp : Array
    : Constant
    : UnaryOperation
    : BinaryOperation

UnaryOperation : Opcode Exp
BinaryOperation: Opcode Exp Exp
UserFunc       : Exp ... Exp
```

Fig. 2. AST definition for NumPy instructions

and optimization such as grouping of multiple AST's into one, dead-code elimination or other optimizations. Preparations for more advanced code generation methods would be part of this phase as well.

In Phase three we transform the AST into a kernel function which are compiled and linked into the running program. This involves code generation and compilations.

Phase four handles the execution of the kernels either directly handed down from phase three or extracted from the cache.

A. Abstract Syntax Tree for the vector expressions.

AST are generally used in compilers and interpreters to represent the abstract syntax of the program. This representation follows the Concrete Syntax Tree (CST) which is representation of the program text.

We focus the AST for the JIT framework on representing the mathematical expressions found in the instruction lists and uses a list to represent the order of the execution.

We defined the AST used to represent the mathematical expressions as depicted in the figure 2. We formally defined the AST to include the following two types of components, the Statement and Expressions. The Statements assigns the value of an Expression to an array. The Expression can take many forms as it is the case with mathematical expressions.

The simplest form is the array or constant. These are used with unary and binary operator as well as userdefined functions. These operations defines the recursive nature of the data structure as they themselves are Expressions and takes expressions as input. The Opcode is a basic mathematical operators, such as add, multiply or sinus.

With this definition we are able to express the mathematical expressions of the bytecode.

An Assignment is an instruction and thus a program consists of series of assignments which assigns the value of a simple array, constant or unary or binary expressions to an array. In the case of the constant assignment it would be broadcasted to all elements of the array.

We view the bytecode instruction as an assignment with a left and right side. The right side is the Expr and the array the left. When an expression consists of more then a single unary or binary operations we label it as a composite expression, as it composed of multiple expressions. To manage the assignment of expressions to arrays a set of data structures are used.

1) Data structures to manage the AST: We present the three data structures we use to create and manage the AST-representation of an list of instructions:

- BaseUsageTable
- SSAMap
- Nametable

The BaseUsageTable is used to register when a base array are written to. With this information we can determine dependency violation with in the AST's and ensure correct execution.

In Numpy the use of slices of data is represented af view of the original data. This view is called an array and will always be present. Multiple arrays can reference the same underlying data, called a base-array. Operations on the different arrays can thus alter the same base-array. We register which arrays use the same base-arrays to ensure that data is written to the base-array before it is used in a new expression.

The BaseUsageTable is implemented as a Map of lists, {array: [nt_index,...],..}, where there for each basearray is a list of references to the Nametable of where the array is used.

We use the Static Single Assignment map (SSAMap) in the creation phase of the Nametable and AST's. We build the Nametable in SSA form where each assignments as its unique name to eases later analysis.

The SSAMap registers all arrays in an version list, {array: [nt_index,..],..}, which works as translation table from name to array version.

The Nametable is used to store the AST representation and associated meta-information, such as traversal states and dependencies. With the SSA map all arrays are assigned a new name, an integer. These names are assigned in order of appearances while the Nametable is being build.

The Nametable can be seen as a mapping between Name, Array and AST and holds the following information:

- A Reference to AST
- The array the AST is assigned to, a Target Array (TArray)
- A reference to the instruction in the instruction batch
- A List of Depend-ON and Dependent-TO references (DON and DTO)
- When the TArray is discarded and freed.
- If the name points to a userdefined function, additional information is kept.

All the information from the instruction list kept in the Nametable (see Figure 3).

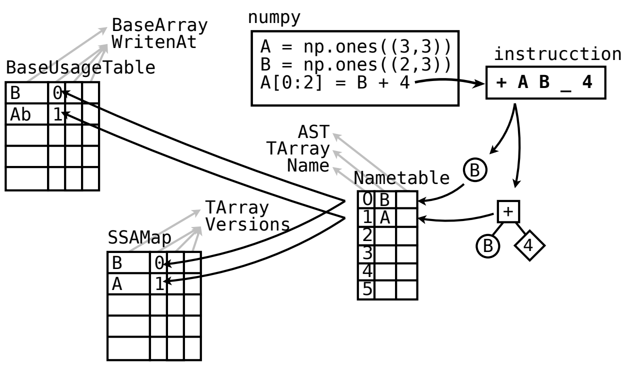


Fig. 3. Nametable relation to instruction

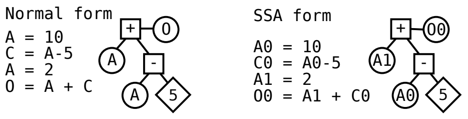


Fig. 4. Static Single Assignment

We do not register constants in the nametable as these cannot be assigned an array and have no importance without a relation to an array. Constants are inserted directly into the autogenerated code and is only used here.

The Nametable is implemented as a vector where the name corresponds to the index. We register all array assignment in the Nametable in the same order they appear in the instruction batch. This means that we preserve the execution order of the created AST in the Nametable structure. When performing the later dependency analysis the order can determined by a comparison on names.

a) Static Single Assignment: SSA is used as a step to encode relationships between variables in code in the naming. It is done by only allowing assignment to a variable once. In the literature [8] if a variable is assigned more then once, a new variable is created with a subscripted number. If this the second time, it is subscripted with a 1, second a 2 and so forth. The relationship to previously used variables are thus encoded into the naming since the name change of a variable is done to the remaining variables in the list of operations. There is no such thing as an overwrite of a variable.

Let us consider the example listed in Figure 4, which can be viewed as a list of instructions. This involves a reassignment of A in the same equation which we need to represent in the AST.

We could do this by only keeping references to the arrays by name, but we would be required to find the correct value of A through a liveliness analysis. We use SSA form for the Nametable to remove this necessity.

In this form all assignment are made to new variables which makes determining the origin of a value much easier as this is now encoded into the name. In traditional languages SSA form handled control flow by introducing phi-functions to represent a changes performed in a branch. As there is no such control flow elements in the bytecode, the SSA form for AST's are very simple.

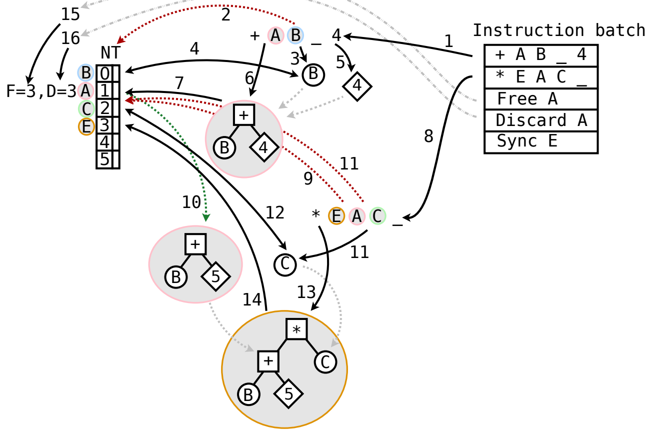


Fig. 5. AST creation illustrated of $\bar{O} = (\bar{B}+5)*\bar{C}$. Underscore (_) indicates a empty value. The arrow numbering show the of the creation process from 1 to 16. The illustration does not cover the all the elements of the creation process. SSAMap, BaseUsageTable and Dependency Graph updates among others, are not included.

We introduce integers as new names where all assignments are assigned a new incremented number. We thus loose the direct relation between names in the naming scheme. In the SSA Map we keep the information on version and their mapping to the new names. This is also used to determine which version of an array should used when referenced.

2) Creating the AST's: As most AST's the AST's in the JIT framework are build bottom up. Starting with the lowest expression and using these as sub trees in the following expressions.

In short, this is done by iterating through the instruction list in the order of execution, creating AST's from the instructions and building larger and larger AST while registering the relations in the Nametable and BaseUsageTable.

This subsection describes the design and implementation of the algorithms used in this process of creating the AST's and filling the Nametable with them.

The instructions are transformed from start to end and analyzed in this order. This results in a bottom-up approach to AST creation, where arrays used in an AST is either new or referencing a existing AST.

While building the Nametable and the AST's we only look back, appending to the existing structure and preserving the order through the naming scheme.

The creation steps of the AST's are best described through an example. Depicted in figure 5 we show the creation of a very simple program which performs the following equation: $\bar{O} = (\bar{B}+5)*\bar{C}$. The program is described by the add and multiply instruction along with a Free, Discard and Sync instruction.

(1) We start from the top of the instruction list looking at the first instruction. (2-3) Extract the left operand $\bar{B}$. As this is and array we check if have a reference to it. (4) We store $\bar{B}$ in the Nametable as 0. (5) We extract the second operand. As this is a constant we do nothing else. (6) We then extract the operator and creates the AST. (7) We store AST in the

nametable as a assignment to $\bar{A}$, as 1.

(8) We are now done with the first instruction and move on to the next in the batch. (9) We lookup the first operand $\bar{A}$ in the Nametable, which is one we previously created and (10) extract the corresponding AST. (11) We lookup and create $\bar{C}$ as it is a new array and (12) stores it in the Nametable. (13) we extact the multiply operator and create the composite AST by combining the AST's of $\bar{A}$ and $\bar{C}$.

(15) we extract free operation for $\bar{A}$, which we register in the Nametable. (16) We do the same for the Discard operation. The Sync is ignored as we execute everything in batch.

As the example shows many elements are in play to transform the instruction batch to a naive forest of AST's. It is not the end of the creation phase as their are still base array dependencies to handle as well as sub expression elimination to perform. This ties into the orchestration phase as dependencies may results in spilling AST's into multiples.

3) Expression Orchestration: The orchestration of AST requires knowledge of dependencies between the AST's. In the rather simple process of building the AST's we do not check for dependencies. As part if the orchestration phase the AST's are dependency validated and violatoins are resolved.

The resulting set of AST are a result of the following reasons:

1) Different expressions used in the program.
2) Varying sizes of arrays used in the computations as a result slicing.
3) Use of arrays across instruction batches result in a unknown case of multiple use.
4) Base array dependency violation.
5) AST subexpressions.

The program is a list of batches of instructions. A batch is thus a sublist og instructions. These batches is a result of the interpreter which at the end of a batch required the evaluation of the expressoin. This could be the result of a print statement or other points in the Python code where evaluation is required. With the Free and Discard intructions we know when the interpreter no longer has a reference. In the case where the free/discard operation for a array is not in the batch we treat the array as having multiple dependencies in the following batch.

The result of the Nametable creation is a set of distinct AST's with a internal relationship. The orchestration is highly influences by the single core target as the AST are arranged in a list structure. This is done by the order of the Nametable which in effect is a sort of the AST by name. The AST root with the smallest name is executed as the first, continuing upwards to the AST with the highest name.

We know that the dependencies are acyclic, meaning that dependencies between AST's are only lower names ones. There is no dependency violation as these have been resolved by splitting AST's into smaller ones. The set of AST's can be viewed as a graph of expression dependent on each other.

Analysis of this set prior to code generation could hold possibilities to group AST's into even larger kernels, further exploiting the loop fusion benefits. Converting the dependency information from a single AST, into a single unit would be done by using the root nodes dependent-to and the AST's and the depend-on dependencies from all leaf-nodes, as the dependencies of the AST.

Analysis of this dependency graph could be used to parallelize independent AST on different CPU cores or to group different sets of dependent AST's into single larger kernels. This is touches more in the future work section.

B. Code Generation

This section describes the transformation from AST's to autogenerated C code and computational kernels. To achieve performance close to that of optimized C code the generated code must be of equal quality. We will in this section cover the information extraction and code generation.

We will take a look at the possibilities for further with runtime generation of kernels.

1) Kernel Function: A kernel function is a C function with a defined signature that perform a series of operations corresponding to one or more instructions and is created from a AST. When compiled it is defined as a computational kernel or just kernel.

A kernel function consist of three logical elements:

- The Input
- Traversal method
- The computation

The input consists of all the distinct arrays and constants used by in the AST.

The reason we pas array distinct is to reduce the number of arrays used in the computation. The computation requires computing the element-position in the matrix's to retrive the correct values. By removing the dublicate arrays and using the same element-position computations multiple time we reduce the number of calculations required.

For constants this is not an issue, and as they are used only once. We pass these to the kernel as a array.

We know in which context the kernel functions are to be used and thus we have no need for size parameters for the input arrays. We use arrays as we must handle different size inputs depending on the kernel and they must all have the same function signature.

Kernel functions are named by the hash of the AST they are based on. This create unique names based on the signature of the AST and used both in naming and later caching. The hash is based on a AST Signature, retrieved by performing a depth-first search, left to right, where the opcode of the node and leafs along with the Type is used to create the signature. Extending the signiture with information on

The traversal method is how the arrays are travered as part of the computataion. The traversal used in the kernels is pointer incrementation where only additions are used to determin the position of data to work in the matrices.

The computation part is the calculation perfomed on the arrays. In the creation phase this is called the computestring and is the computation of a series of values which produces a result.

2) Input extraction and equation creation: The AST's are traversed in a depth-first search left to right. This is the case for all AST traversals in the JIT framework.

The recursive traversal method used in the creational phase of the kernel, extracts the distinct arrays, all constant and builds the compute string. The compute string is created by combining one or two inputs with a operator. This is based on the opcode of the AST nodes which combined with the stringnames for the inputs are merged into the final composite expression. The left-hand side of the created equation is retrieved from the target array of AST, defined in the Nametable.

The computational order from the instruction is kept in the AST structure and no further actions are needed to ensure the order in the kernel creation phase. To ensure the order in the created equation parenthesis are added around each unary or binary expression.

C. Execution and Kernel Cache

Caching is an important part of ensuring a reasonable runtime for the JIT as we will show in section V-A1.

We perform caching to reduce the number of kernel we create, in effect caching the JIT Optimizations for later use.

The compiled kernel is just a function pointer which must be called with a specific number of arguments. We store the array and constant array's used as input with the compiled kernel function in a execution kernel data structure. This structure can hold both compiled kernels, instructions or userdefined function instruction and is thus a wrapper around a element to execute.

The caching model starts with a orchestrated set of AST's.

- A hash of the AST is created and checked against the cache.
- The AST are compiled into a computational kernel.
- The kernels inputs is filled based on a traversal of the AST.
- The kernel is executed.
- The kernel is cached with the hash of the AST as key.

The kernels are directly compiled and executed in the order they have been orchestrated in. This means that when the same kernels are used multiple times only a single kernel is created.

The hash of the AST are done based on a left to right, depth-first-search, which produces a vector of the structure. This can be viewed as a flattening of the operators, types and expression-types which forms the input for a cryptographic hashing algorithm.

V. Evaluation

In this section, we present performance evaluation of our JIT implementation running on a AMD machine (See table I). The framework testing is as follows:

- Each benchmark is the average of three run for each configuration.
- The benchmark scripts are written in Python
- We use the system GCC compiler for both compilation of the C/C++ implementations and the Computational Kernels. All compilations use the -O2 as optimization flag.

CPU	AMD Opteron(TM) Processor 6274
Feq	2.20 GHz
Layout	2 CPU's. 16 Cores per CPU
RAM	128 GB
Software	Linux version 3.2.0-25-generic Python 2.7.3, GCC version 4.6.3

TABLE I. Benchmark Configuiration

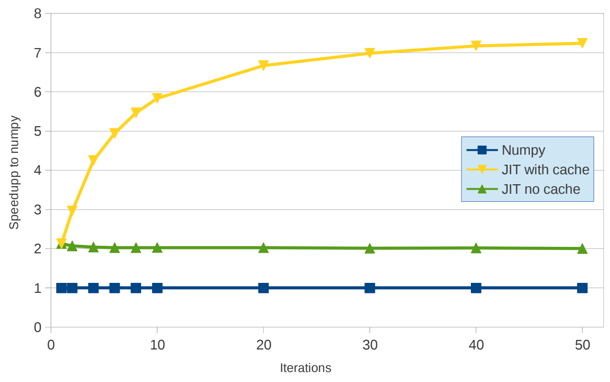


Fig. 6. Effect of kernel cache when running the Jacobi benchmark. The matrix input and output size is fixed at 4k by 4k elements.

- In the benchmarks we measure the computation time only. The time to initialize the arrays are not included.

A. Jacobi

The Jacobi benchmark is a implementation which solves the heat equation iteratively using the Jacobi Method. We use this benchmark to show the effect of kernel caching, relation to problem size and comparison with C/C++ implementations.

1) Caching : We evaluate the effect of kernel caching by comparing the execution with and without caching enabled. By disabling the cache all AST's result in the creation and compilation of a kernel.

The runtime graph depicted in figure 6, clearly shows the overhead of creation and compilation of the kernels and the how this overhead is armotized over time.

2) Problem Size: Figure 7 show the runtime of the first iteration in the Jacobi benchmark. The graphs show that there is a strong correlation between the benefit of the JIT methods and the size of data. As the data grows the runtime follow in a similar quadratic way and illustrates the significant difference in growth between the JIT and the Numpy execution.

With problems larger then 2k-by-2k elements, JIT is faster then Numpy even in the first iteration. This will for the most part be the case for the Jacobi or programs with similar computational complexity as this includes the creation of all the computation kernels. The first iterations will be the most expensive, as caching removed the overhead of kernel creation in the later iterations.

3) C/C++ Comparison: We wish to bridge the gap between low-level languages and and thus we compare the Numpy implementation with a range of different C/C++ implementations as the methods used in have great impact on the performance.

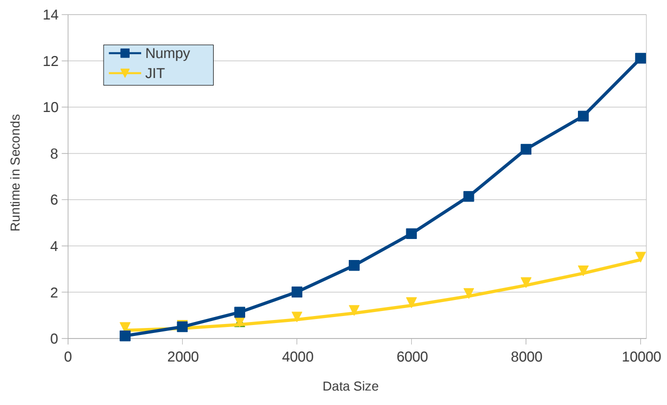


Fig. 7. Effect of the problem size running the first iteration of the Jacobi benchmark. The matrix input and output size grows from 1k by 1k to 1-kby 10k.

Depicted in Figure 8 is the speedup graph of the Jacobi implementations. The C/C++ versions are:

- Naive: A naive implementation where indexing into the matrices are done multiplying a column count with the row length to get the index for an element.
- Tuned: Only pointers are used to index the matrices. For each row iteration only pointer incremention is need. To change columns a second add is done.
- VTuned: Optimization of the Tuned, thus VeryTuned, where columns are handled slightly more efficient.
- Boost: Use of the Boost 2D array data structure.

These four implementation can be seen as four different approaches or stages of a C implementation based on a Matlab or Numpy prototype. The Naive approach or using libraries as Boost would be a common first step and for many the only step. Using pointers incrementation instead of coordinate calculations requires a thorough understanding of C and could be seen as a next step. The Tuned implementation divides the normal programmer from the specialist and the step further to VTuned pushes the expertise needed even further.

We observe a surprising ordering where the C version are not gathered in the top. The JIT implementation is significantly faster then the Naive approach aswell as the implementation using the Boost library. The Pointer based implementation are grouped in top with very high execution speeds.

In figure 8 the higher speeds of the pointer based solution is clearly visible. The fluctuations of Tuned and VTuned is a result of normal noise, as the difference between them are in the +/-0.2 second range.

B. Black Scholes

The Black Scholes method is used to determine the price of European options. The algorithm is run over a time series which is the represent the iterations done.

The input data is a single dimensional vector and the operations performed are highly dependent on scalars. The C version is implemented in a double for loop summing the intermediate results together.

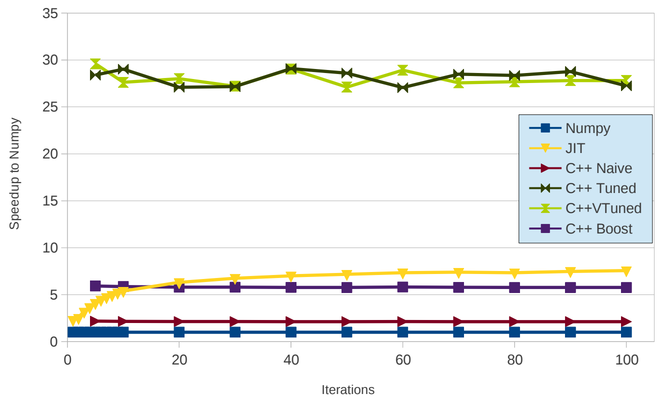


Fig. 8. Comparison of C/C++ vs NumPy implementations of the Jacobi benchmark. The matrix input and output size is fixed at 4k by 4k elements.

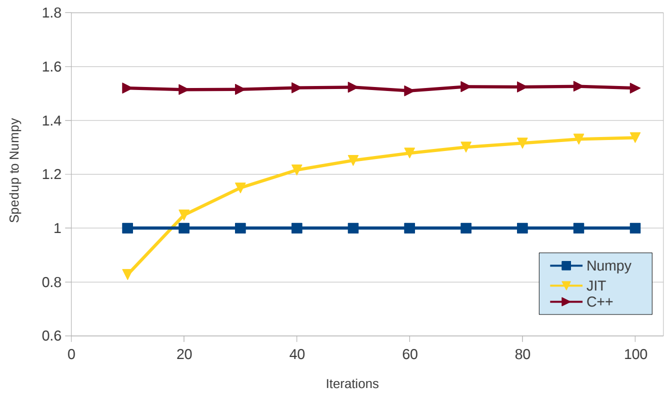


Fig. 9. Comparison of C/C++ vs NumPy implementations of the Black Scholes benchmark using a 200k elements data set.

The execution consists of 200K elements and uses from 10 to 100 iterations. Figure 9 shows the speedup compared to NumPy. Comparing the result of the NumPy implementation that uses JIT with the C implementation, we observe that the C execution is much faster at few iterations. However, at 100 iterations the C implementaion is only slightly faster.

C. K-Nearest Neighbor

The K-Nearest Neighbor is an algorithm which find the closest K closest neighbors to a given point. This means that the distance between all points must be calculated to determine which are the closes. This is learning algorithm is often used to determine classification of elements in a dataset, which can consists of multi-dimensional data elements. The implementation is made in Numpy without any loops, resulting 4 kernels of which 2 is userdefined functions and the remaining is computational kernels. The test is run on a varying number of elements K ranging from 10 to 100. Each elements can be seen as a point in a 50000 dimensional space.

Figure 10 shows the same trend of increasing speedup over iterations. At 140 iterations we see a speedup greater than 7.

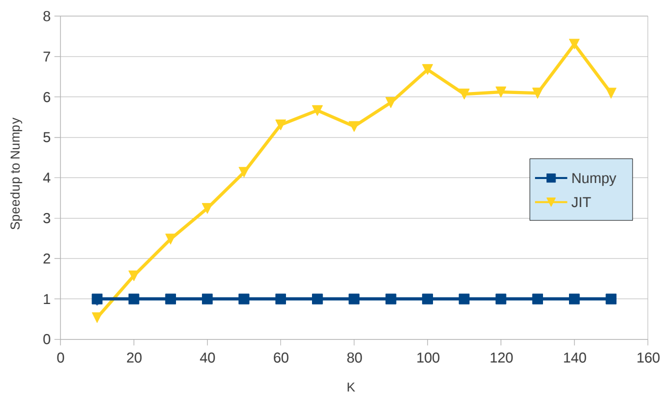


Fig. 10. Comparison of two NumPy executions – one using the regular NumPy implementation and one using our JIT backend – that runs the KNN benchmark. The data set consist of 50k elements.

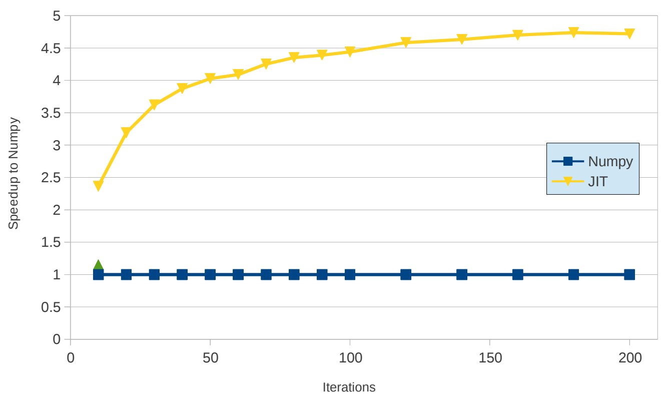


Fig. 11. Comparison of two NumPy executions – one using the regular NumPy implementation and one using our JIT backend – that runs the N-Body benchmark. The data set consist of 100 bodies.

D. *N-Body*

The N-Body test is a simulation of elements and how the gravitational forces effects the movement of them. We used a naive method where the effect of all elements are applied to all elements. The simulation is run with 1000 elements for 100 iterations, where each iteration is a timestep in the algorithm. As the algorithm contains no natural batching a manual flush have been inserted in the Numpy to break the instruction list into batches.

In figure 11 we show the results of running the N-Body benchmark. We see the same trends again, where the JIT methods performance increase over time as the initial overhead is amortized along with a small difference between the JIT methods.

E. *Summery*

Our benchmark results show a solid speedup across all test. This a result of both loop-fusion and temporary array removal, as well as the implemented kernel cache. This combination shows significant improvement.

We clearly see that the number of iterations have a significant impact on the speedup. Performance decreases are seen in the first iteration of all test and shows that a large part of the base speedup is a result of the cache. With this we see the overhead of the initial kernel creations amortized over the iterations.

We see a correlation between the problem size and complexity, which both effect the potential speedup. As the problem size, complexity or both, rises the speedup to compared to normal Numpy follows. This is reasonable as the effect temporary arrays removal and loop-fusion has a per-element effect, where the complexity of the program is reflected in number of arrays fused together in the kernels. A large and complex problem will get the largest benefit from the JIT Framework.

We are very close or better the naive C implementation, but as showed in the Jacobi examples there is still room for improvements. We do not expect to reach the computation times of optimized C code due the overhead of Numpy and the JIT framework. Comparing with the naive and/or Boost based C implementation shows that we clearly are within range of these.

VI. Future Work

In this section we take a look into the future of the JIT framework. This includes interesting areas for further investigation as well as possible optimizations. This section follows the phases of the Framework as there are paths to investigate in most of the JIT stages.

A. *AST*

The AST's are now focused on the mathematical equation and represent these very well but there are other elements related to the vector operations which could be represented in the same structure. In many other bytecode formats control operations are part of the representation. Operations for Reduction and Broadcasting could be added, allowing for AST's to include different shapes and provide more information about their use.

Introduce optimizations based on the AST's prior to kernel creation. This could be dead code elimination or redefinitions of the equations which could lead to reduced computational complexity.

B. *Code Generation*

In the code generation phase there is a range of areas to investigate further. The implementation presented has focused on building the framework and investigating many areas of the JIT kernel creation. It clear that optimizations to the kernel code and the method kernels are created is a significant part of nearing the runtime of optimized C.

The following optimization can be applied to the kernel to achieve a increased performance on the single core architecture:

- Loop unrolling: Perform multiple operation in the most inner loop to reduce the number of index calculations needed in the traversal.

- Use machine specific information: Utilize available information such as cache size or architecture in the creation of the code. This could be to use special libraries for certain operation, to take better advantage of the cache or use special compiler flags or specific compilers.

The first optimization are straightforward to implement in the existing framework by extending code generation. Taking advantage of architecture specifics will requires substantially more work, as the optimization will be build on more advanced technologies. This could be Multicore, NUMA, cache-tileing or SSE instructions.

Creating larger kernels based on multiple AST. Combining multiple AST's into a single kernel will have multiple advantages. By grouping AST's which output is used in multiple other AST's together aditional tempoary arrays can be removed. Using traversal calculations on multiple unrelated AST's in parallel will reduce reduce the time spend on index calculations taking further advantage of loop-fusion.

This would also be the case for reduction as these could become part of the execution loop. In case of a reduction it could be directly applied to the result of the computation eliminating the need to store the result in a temp array, only to perform a reduce afterwards.

The use of LLVM and C-Lang to compile the kernel function should be investigated. This would enable the kernel creation to be done in memory by using the available library. Apposed to creating C code and compiling this, it could be more beneficial to create the intermediate language of LLVM and use their compiler to generator executable code. This could bring down the overhead from the compilation making the approach of JIT compilation more attractive for programs which is translated into many distinct kernels.

VII. Conclusion

We have implemented a JIT framework for Python/NumPy that allow NumPy instructions to be expressed in an abstract form using Abstract Syntax Tree's. This has allowed for a set of optimizations to the computations of Numpy vector operations and enables further optimizations.

Our approach of transforming the NumPy instructions into AST's is well suited to compose bytecode instructions into composite expressions. We show that this composition results in loop-fusion and temporary array removal when transformed into computational kernels.

The process of creating ASTs is a non-trivial task because arrays may share the same underlying data structure and dependencies. By performing dependency analysis and breaking initial AST's into smaller trees, we transform the instruction batch into a forest of connected tree, which express the syntax of the batch much cleaner than a single instruction list.

We show that the use of a kernel cache provides a significant increase in performance in real world tests. This effect is largest for small problem sizes, where the overhead of kernel creation is expensive, but at larger problem sizes this become insignificant as shown in fig 8.

The combined effect of temporal array removal, loop-fusion and caching show significant speedups. Our benchmark of Jacobi and N-Body present speedups compared to Numpy of 7.51 and 4.72 respectively. Comparing to C version we observe that we are close to or achieve better performance then naive implementations with or without the Boost library, but we are still orders of magnitude slower than optimized C.

We achieve these result with a combined set of unoptimized and in many cases naive implementations. The JIT framework allow many more interesting optimizations that have yet to be applied. In the future work section, we outlined a set of optimization that bridge the performance even closer to an optimized C implementation.

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (hiperfit.dk) under contract number 10-092299.

References

[1] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter, "Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster," in *Python for High Performance and Scientific Computing (PyHPC 2013)*, 2013.

[2] R. Garg and J. N. Amaral, "Compiling python to a hybrid execution environment," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 19–30.

[3] A. Klckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157 – 174, 2012.

[4] "Gnu c copiler."

[5] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[6] F. Bellard, "Tcc: Tiny c compiler," *URL: http://fabrice. bellard. free. fr/tcc*, 2003.

[7] http://openbenchmarking.org/result/1204215 SU-LLVMCLANG23, "Llvm clang 3.1 gcc 4.7 intel core i7 benchmarks," 2012.

[8] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

Johannes Lund Master in Computer Science from The Department of Computer Science, University of Copenhagen (DIKU). The main focus of the Master's has been in Distributed Systems and High-Performance Computing. The Master theses investigated methods to improve performance for matrix operation in Bohrium on a single CPU core.

Mads R. B. Kristensen PostDoc at the Niels Bohr Institute, University of Copenhagen. His primary research areas are High Performance Computing and PGAS languages/libraries. He has developed algorithms and frameworks targeting supercomputers, including Cray XE6 and Blue Gene/P.

Simon A. F. Lund PhD student at the Niels Bohr Institute, at the University of Copenhagen. The title for his PhD project is "A High-Performance Backend for Computational Finance on Next-Generation Processing Units" which evolves around mapping high-level language constructs to hardware, with a focus on efficient utilization of SIMD-units, and Multi-Core architectures.

Brian Vinter Professor at the Niels Bohr Institute, University of Copenhagen. His primary research areas are Grid Computing, Supercomputing, and Many-core architectures. He has done research in the field of High Performance Computing since 1994. Current research includes methods for seamlessly utilization of parallelism in scientific application.

6.7 Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster

Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster

Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter
Niels Bohr Institute, University of Copenhagen, Denmark
{madsbk/safl/blum/skovhede/vinter}@nbi.dk

Abstract—**In this paper we introduce Bohrium, a runtime-system for mapping array-operations onto a number of different hardware platforms, from multi-core systems to clusters and GPU enabled systems. As a result, the Bohrium runtime system enables NumPy code to utilize CPU, GPU, and Clusters. Bohrium integrates seamlessly into NumPy through the implicit data parallelization of array operations, which are called Universal Functions in NumPy. Bohrium requires no annotations or other code modifications besides changing the original NumPy import statement to: "`import bohrium as numpy`".**

We evaluate the presented design through a setup that targets a multi-core CPU, an eight-node Cluster, and a GPU, all implemented as preliminary prototypes. The evaluation includes three well-known benchmark applications, *Black Sholes*, *Shallow Water*, and *N-body*, implemented in Python/NumPy.

I. INTRODUCTION

The popularity of the Python programming language is growing in the HPC community. Python is a high-productivity programming language that focus on high-productivity rather than high-performance thus it might seem paradoxical that such a language would gain popularity in HPC. However, Python is easily extensible with libraries implemented in high-performance languages such as C and FORTRAN, which makes Python a great tool for gluing high-performance libraries together[1].

NumPy is the de-facto standard for scientific applications written in Python[2]. It provides a rich set of high-level numerical operations and introduces a powerful array object. NumPy supports a declarative vector programming style where numerical operations operate on full arrays rather than scalars. This programming style is often referred to as vector or array programming and is commonly used in programming languages and libraries that target the scientific community, e.g. HPF[3], MATLAB[4], Armadillo[5], and Blitz++[6].

A major shortcoming of Python/NumPy is the lack of thread-based concurrency. The de-facto Python interpreter, CPython, uses a Global Interpreter Lock to serialize concurrent execution of Python bytecode thus parallelism in restricted to external libraries. Similarly, NumPy does not parallelize array operations but might use external libraries, such as BLAS or FFTW, that do support parallelism.

The result is that Python/NumPy is great for gluing HPC code together, but often it cannot stand by itself. In this paper, we introduce a framework that addresses this issue. We introduce a runtime system, Bohrium, which seamlessly executes NumPy array operations in parallel. Through Bohrium, it is possible to utilize CPU, GPU, and Clusters without changing the original Python/NumPy code besides adding the import statement: "`import bohrium as numpy`".

In order to couple NumPy with the execution back-end, Bohrium uses an intermediate vector bytecode that correspond to the NumPy array operations. The execution back-end is then able to execute the intermediate vector bytecode without any Python/NumPy knowledge, which also makes Bohrium usable for any programming language. Additionally, the intermediate vector bytecode solves the Python *import problem* where the "`import numpy`" instruction overwhelms the file-system in supercomputers[7], [8]. With Bohrium, only a single node needs to run the Python interpreter, the remaining nodes execute the intermediate vector bytecode directly.

The version of Bohrium we present in this paper is a proof-of-concept implementation that supports the Python programming language through a Bohrium implementation of NumPy[1]. However, the Bohrium project also supports additional languages, such as C++ and Common Intermediate Language (CIL)[2], which we have described in previous work [**?**]. The proof-of-concept implementation supports three computer architectures: CPU, GPU, and Cluster.

II. RELATED WORK

The key motivation for Bohrium is to provide a framework for the utilization of diverse and complex computing systems, with the goal of obtaining high-performance, high-productivity and high-portability, *HP^3*. Systems such as pyOpenCL/pyCUDA[9] provides tools for interfacing a high abstraction front-end language with kernels written for specific potentially exotic hardware. In this case, lowering the bar for harvesting the power of modern GPU's, by letting the user write only the GPU-kernels as text strings in the host language Python. The goal is similar to that of Bohrium – the approach however is entirely different. Bohrium provides a means to hide low-level target specific code behind a programming model and providing a framework and runtime environment to support it.

Bohrium is more closely related to the work described in [10], where a compilation framework, unPython, is provided for execution in a hybrid environment consisting of both CPUs and GPUs. The framework uses a Python/NumPy based front-end that uses Python decorators as hints to do selective optimizations. Bohrium performs data-centric optimizations on vector operations, which can be viewed as akin to selective optimizations, in the respect that we do *not* optimize the

[1]The implementation is open-source and available at www.bh107.org
[2]also known as Microsoft .NET

program as a whole. However, we find that the approach used in the Bohrium Python interface is much less intrusive. All arrays are by default handled by Bohrium – no decorators are needed or used. This approach provides the advantage that any existing NumPy program can run unaltered and take advantage of Bohrium without changing a single line of code. In contrast, unPython requires the user to modify the source code manually, by applying hints in a manner similar to that of OpenMP. The proposed non-obtrusive design at the source level is to the author's knowledge novel.

Microsoft Accelerator [11] introduces ParallelArray, which is similar to the utilization of the NumPy arrays in Bohrium but there are strict limitations to the utilization of ParallelArrays. ParallelArrays does not allow the use of direct indexing, which means that the user must copy a ParallelArray into a conventional array before indexing. Bohrium instead allows indexed operations and additionally supports *vector-views*, which are vector-aliases that provide multiple ways to access the same chunk of allocated memory. Thus, the data structure in Bohrium is highly flexible and provides elegant programming solutions for a broad range of numerical algorithms. Intel provides a similar approach called Intel Array Building Blocks (ArBB) [12] that provides retargetability and dynamic compilation. It is thereby possible to utilize heterogeneous architectures from within standard C++. The retargetability aspect of Intel ArBB is represented in Bohrium as a simple configuration file that defines the Bohrium runtime environment. Intel ArBB provides a high performance library that utilizes a heterogeneous environment and hides the low-level details behind a declarative vector-programming model similar to Bohrium. However, ArBB only provides access to the programming model via C++ whereas Bohrium is not limited to any one specific front-end language.

On multiple points, Bohrium is closely related in functionality and goals to the SEJITS [13] project, but takes a different approach towards the front-end and programming model. SEJITS provides a rich set of computational kernels in a high-productivity language such as Python or Ruby. These kernels are then specialized towards an optimality criterion . The programming model in Bohrium does not provide this kernel methodology, but deduces computational kernels at runtime by inspecting the flow of vector bytecode.

Bohrium provides, in this sense, a virtual machine optimized for execution of vector operations. Previous work [14] was based on a complete virtual machine for generic execution whereas Bohrium provides an optimized subset.

III. The Front-end Language

To hide the complexities of obtaining high-performance from the diverse hardware making up modern computer systems any given framework must provide a meaningful high-level abstraction. This can be realized in the form of domain specific languages, embedded languages, language extensions, libraries, APIs etc. Such an abstraction serves two purposes: (1) It must provide meaning for the end-user such that the goal of high-productivity can be met with satisfaction. (2) It must provide an abstraction that consists of a sufficient amount of information for the system to optimize its utilization.

```
import bohrium as numpy
solve(grid, epsilon):
  center = grid[1:-1,1:-1]
  north  = grid[-2:,1:-1]
  south  = grid[2:,1:-1]
  east   = grid[1:-1,:2]
  west   = grid[1:-1,2:]
  delta  = epsilon+1
  while delta > epsilon:
    tmp = 0.2*(center+north+south+east+west)
    delta = numpy.sum(numpy.abs(tmp-center))
    center[:] = tmp
```

Fig. 1: Python/NumPy implementation of the heat equation solver. The `grid` is a two-dimensional NumPy array and the `epsilon` is a Python scalar. Note that the first line of code imports the Bohrium module instead of the NumPy module, which is all the modifications needed in order to utilize the Bohrium runtime system.

Bohrium does not introduce a new programming language and is not biased towards any specific choice of abstraction or front-end technology. However, the front-end must be compatible with the declarative vector programming model and support vector slicing, also known as vector or matrix slicing [3], [4], [15], [16]. Bohrium introduces *bridges* that integrate existing languages into the Bohrium runtime system.

The Python Bridge is an extension of NumPy version 1.6, which seamlessly implements a new array back-end that inherits the manipulation features, such as *slice*, *reshape*, *offset*, and *stride*. As a result, the user only needs to modify the import statement of NumPy (Fig. 1) in order to utilize Bohrium.

The Python Bridge uses *hooks* to divert function call where the program accesses Bohrium enabled NumPy arrays. The hooks will translate a given function into its corresponding Bohrium bytecode when possible. When it is not possible, the hooks will feed the function call back into NumPy and thereby forces NumPy to handle the function call itself. The Bridge operates with two address spaces for arrays: the Bohrium space and the NumPy space. The user can explicitly assign new arrays to either the Bohrium or the NumPy space through a new array creation parameter. In two circumstances, it is possible for an array to transfer from one address space to the other implicitly at runtime.

1) When an operation accesses an array in the Bohrium address space but it is not possible for the bridge to translate the operation into Bohrium bytecode. In this case, the bridge will synchronize and move the data to the NumPy address space. For efficiency, no data is actually copied. Instead, the bridge uses the `mremap` function to re-map the relevant memory pages when the data is already present in main memory.
2) When an operations accesses arrays in different address spaces the Bridge will transfer the arrays in the NumPy space to the Bohrium space.

In order to detect direct access to arrays in the Bohrium address space by the user, the original NumPy implementation, a Python library, or any other external source, the bridge protects the memory of arrays that are in the Bohrium address space using `mprotect`. Because of this memory protection, subsequently accesses to the memory will trigger a segmentation fault. The Bridge can then handle this kernel signal by

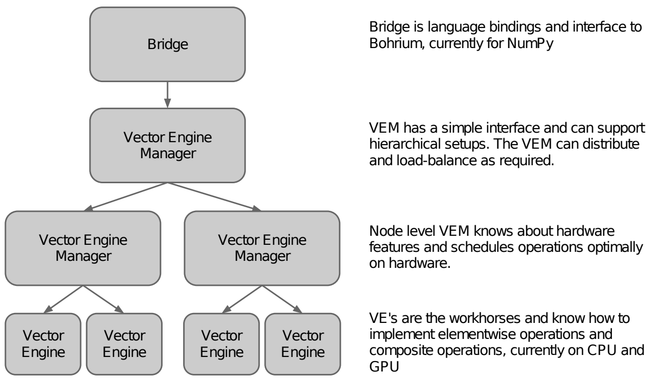

Bridge
Vector Engine Manager
Vector Engine Manager
Vector Engine Manager
Vector Engine
Vector Engine
Vector Engine
Vector Engine
Bridge is language bindings and interface to Bohrium, currently for NumPy
VEM has a simple interface and can support hierarchical setups. The VEM can distribute and load-balance as required.
Node level VEM knows about hardware features and schedules operations optimally on hardware.
VE's are the workhorses and know how to implement elementwise operations and composite operations, currently on CPU and GPU

Fig. 2: Bohrium Overview

transferring the array to the NumPy address space and cancel the segmentation fault. This technique makes it possible for the Bridge to support all valid Python/NumPy application, since it can always fall back to the original NumPy implementation.

To reduce the overhead related to generating and processing the bytecode, the Bohrium Bridge uses lazy evaluation for recording instruction until a side effect can be observed.

IV. The Bohrium Runtime System

The key contribution in this work is a framework, Bohrium, which significantly reduces the costs associated with high-performance program development. Bohrium provides the mechanics to couple a programming language or library with an architecture-specific implementation seamlessly.

Bohrium consists of a number of components that communicate by exchanging a *Vector Bytecode*[3]. Components are allowed to be architecture-specific but they are all interchangeable since all uses the same communication protocol. The idea is to make it possible to combine components in a setup that match a specific execution environment. Bohrium consist of the following three component types (Fig. 2):

Bridge The role of the Bridge is to integrate Bohrium into existing languages and libraries. The Bridge generates the Bohrium bytecode that corresponds to the user-code.

Vector Engine Manager (VEM) The role of the VEM is to manage data location and ownership of arrays. It also manages the distribution of computing jobs between potentially several Vector Engines, hence the name.

Vector Engine (VE) The VE is the architecture-specific implementation that executes Bohrium bytecode.

When using the Bohrium framework, at least one implementation of each component type must be available. However, the exact component setup depends on the runtime system and what hardware to utilize, e.g. executing NumPy on a single machine using the CPU would require a Bridge implementation for NumPy, a VEM implementation for a machine node, and a VE implementation for a CPU. Now, in order to utilize a GPU instead, we can exchange the CPU-VE with a GPU-VE without having to change a single line of code in the NumPy application. This is a key contribution of Bohrium: the ability to change the execution hardware without changing the user application.

[3]The name vector is roughly the same as the NumPy array type, but from a computer architecture perspective vector is a more precise term

```
# Bridge for NumPy
[numpy]
type = bridge
children = node

# Vector Engine Manager for a single machine
[node]
type = vem
impl = libbh_vem_node.so
children = gpu

# Vector Engine for a GPU
[gpu]
type = ve
impl = lbbh_ve_gpu.so
```

Fig. 3: This example configuration provides a setup for utilizing a GPU on one machine by instructing the Vector Engine Manager to use the GPU Vector Engine implemented in the shared library `lbhvb_ve_gpu.so`.

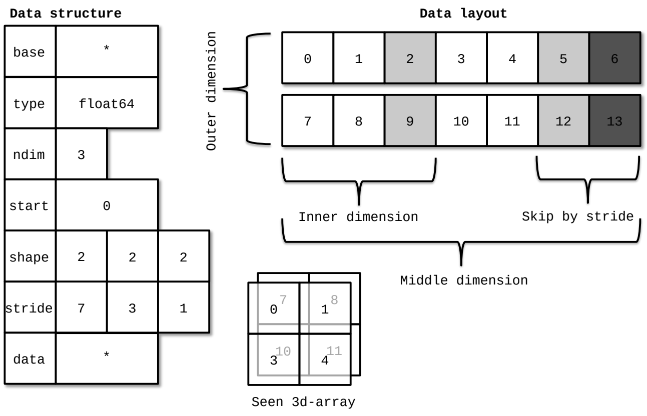


Fig. 4: Descriptor for n-dimensional array and corresponding interpretation

A. Configuration

To make Bohrium as flexible a framework as possible, we manage the setup of all the components at runtime through a configuration file. The idea is that the user or system administrator can specify the hardware setup of the system through an ini-file (Fig. 3). Thus, it is just a matter of editing the configuration file when changing or moving to a new hardware setup and there is no need to change the user applications.

B. Vector Bytecode

A vital part of Bohrium is the *Vector Bytecode* that constitutes the link between the high-level user language and the low-level execution engine. The bytecode is designed with the declarative array-programming model in mind where the bytecode instructions operate on input and output arrays. To avoid excessive memory copying, the arrays can also be shaped into multi-dimensional arrays. These reshaped array views are then not necessarily comprised of elements that are contiguous in memory. Each dimension comprises a stride and size, such that any regularly shaped subset of the underlying data can be accessed. We have chosen to focus on a simple, yet flexible, data structure that allows us to express any regularly distributed arrays. Figure 4 shows how the shape is implemented and how the data is projected.

The aim is to have a vector bytecode that support data parallelism implicitly and thus makes it easy for the bridge to translate the user language into the bytecode efficiently. Additionally, the design enables the VE to exploit data parallelism through SIMD[4] and the VEM through SPMD[5].

In the following, we will go through the four types of vector bytecodes in Bohrium.

1) Element-wise: Element-wise bytecodes performs a unary or binary operation on all array elements. Bohrium currently supports 53 element-wise operations, e.g. addition, multiplication, square root, equal, less than, logical and, bitwise and, etc. For element-wise operations, we only allow data overlap between the input and the output arrays if the access pattern is the same, which, combined with the fact that they are all stateless, makes it straightforward to execute them in parallel.

2) Reduction: Reduction bytecodes reduce an input dimension using a binary operator. Again, we do not allow data overlap between the input and the output arrays and the operator must be associative. Bohrium currently supports 10 reductions, e.g. addition, multiplication, minimum, etc. Even though none of them are stateless, the reductions are all straightforward to execute in parallel because of the non-overlap and associative properties.

3) Data Management: Data Management bytecodes determine the data ownership of arrays, and consists of three different bytecodes. The synchronization bytecode orders a child component to place the array data in the address space of its parent component. The free bytecode orders a child component to free the data of a given array in the global address space. Finally, the discard operator that orders a child component to free the meta-data associated with a given array, and signals that any local copy of the data is now invalid. These three bytecodes enable lazy allocation where the actual array data allocation is delayed until it is used. Often arrays are created with a generator (e.g. random, constants) or with no data (e.g. temporary), which may exist on the computing device exclusively. Thus, lazy allocation may save several memory allocations and copies.

4) Extension methods: The above three types of bytecode make up the bulk of a Bohrium execution. However not all algorithms may be efficiently implemented in this way. In order to handle operations that would otherwise be inefficient or even impossible, we introduce the fourth type of bytecode: extension methods. We impose no restrictions to this generic operation; the extension writer has total freedom. However, Bohrium do not guarantee that all components support the operation. Initially, the user registers the extension method with paths to all component-specific implementations of the operation. The user then receives a new handle for this *extension method* and may use it subsequently as a vector bytecode. Matrix multiplication and FFT are examples of a extension methods that are obviously needed. For matrix multiplication, a CPU specific implementation could simply call a native BLAS library and a Cluster specific implementation could call the ScaLAPACK library[17].

[4]Single Instruction, Multiple Data

[5]Single Program, Multiple Data

C. Bridge

The Bridge component is the *bridge* between the programming interface, e.g. Python/NumPy, and the VEM. The Bridge is the only component that is specifically implemented for the user programming language. In order to add Bohrium support to a new language or library, only the bridge component needs to be implemented. The bridge component generates bytecode based on the user application and sends them to the underlying VEM.

D. Vector Engine Manager

Rather than allowing the Bridge to communicate directly with the Vector Engine, we introduce a Vector Engine Manager into the design. The VEM is responsible for one memory address space in the hardware configuration. The current version of Bohrium implements two VEMs: the Node-VEM that handles the local address space of a single machine and the Cluster-VEM that handles the global distributed address space of a computer cluster.

The Node-VEM is very simple since the hardware already provides a shared memory address space; hence, the Node-VEM can simply forward all instruction from its parent to its child components. The Cluster-VEM, on the other hand, has to distribute all arrays between Node-VEMs in the cluster.

1) Cluster Architectures: In order to utilize scalable architectures fully, distributed memory parallelism is mandatory. The current Cluster-VEM implementation is currently quite naïve; it uses the bulk-synchronous parallel model[18] with static data decomposition and no communication latency hiding. We know from previous work than such optimizations are possible[19].

Bohrium implements all communication through the MPI-2 library and use a process hierarchy that consists of one master-process and multiple worker-processes. The master-process executes a regular Bohrium setup with the Bridge, Cluster-VEM, Node-VEM, and VE. The worker-processes, on the other hand, execute the same setup but without the Bridge and thus without the user applications. Instead, the master-process will broadcast vector bytecode and array meta-data to the worker-processes throughout the execution of the user application.

Bohrium use a data-centric approach where a static decomposition dictates the data distribution between the MPI-processes. Because of this static data decomposition, all processes have full knowledge of the data distribution and need not exchange data location meta-data. Furthermore, the task of computing array operations is also statically distributed which means that any process can calculate locally what needs to be sent, received, and computed. Meta-data communication is only needed when broadcasting vector bytecode and creating new arrays – a task that has an asymptotic complexity of $O(\log_2 n)$, where n is the number of nodes.

E. Vector Engine

The Vector Engine (VE) is the only component that actually does the computations, specified by the user application. It has to execute instructions it receives in an order that comply with the dependencies between instructions. Furthermore, it has to ensure that its parent VEM has access to the results as governed by the Data Management bytecodes.

1) CPU: The CPU-ve utilizes all cores available on the given CPU. The CPU-ve is implemented as a in-order interpreter of bytecode. It features dynamic compilation for single-expression just-in-time optimization. Which allows the engine to perform runtime-value-optimization, such as specialized interpretation based on the shape and rank of operands. As well as parallelization using OpenMP.

Dynamic memory allocation on the heap is a time-consuming task. This is particularly the case when allocating large chunks of memory because of the involvement of the system kernel. Typically, NumPy applications use many temporary arrays and thus use many consecutive equally sized memory allocations and de-allocations. In order to reduce the overhead associated with these memory allocations and de-allocations, we make use of a reusing scheme similar to a Victim Cache[20]. Instead of de-allocating memory immediately, we store the allocation for later reuse. If we, at a later point, encounter a memory allocation of the same size as the stored allocation, we can simply reuse the stored allocation. In order to have an upper bound of the extra memory footprint, we have a threshold for the maximum memory consumptions of the cache. When allocating memory that does not match any cached allocations, we de-allocate a number of cached allocations such that the total memory consumption of the cache is below the threshold. Previous work has proven this memory-reusing scheme very efficient for Python/NumPy applications[21].

2) GPU: To harness the computational power of the modern GPU we have created the GPU-VE for Bohrium. Since Bohrium imposes an array oriented style of programming on the user, which directly maps to data-parallel execution, Bohrium byte code is a perfect match for a modern GPU.

We have chosen to implement the GPU-VE in OpenCL over CUDA. This was the natural choice since one of the major goals of Bohrium is portability, and OpenCL is supported by more platforms.

The GPU-VE currently use a simple kernel building and code generation scheme: It will keep adding instructions to the current kernel for as long as the shape of the instruction output matches that of the current kernel, and adding it will not create a data hazard. Input parameters are registered so they can be read from global memory. Similarly, output parameters are registered to be written back to global memory.

The GPU-VE implements a simple method for temporary array elimination when building kernels:

- If the kernel already reads the input, or it is generated within the kernel, it will not be read from global memory.
- If the instruction output is not need later in the instruction sequence – signaled by a discard – it will not be written back to global memory.

```
 1 ...
 2 ADD t1, center, north
 3 ADD t2, t1, south
 4 FREE t1
 5 DISCARD t1
 6 ADD t3, t2, east
 7 FREE t2
 8 DISCARD t2
 9 ADD t4, t3, west
10 FREE t3
11 DISCARD t3
12 MUL tmp, t4, 0.2
13 FREE t4
14 DISCARD t4
15 MINUS t5, tmp, center
16 ABS t6, t5
17 FREE t5
18 DISCARD t5
19 ADD_REDUCE t7, t6
20 FREE t6
21 DISCARD t6
22 ADD_REDUCE delta, t7
23 FREE t7
24 DISCARD t7
25 COPY center, tmp
26 FREE tmp
27 DISCARD tmp
28 SYNC delta
29 ...
```

Fig. 5: Bytecode generated in each iteration of the Jacobi Method code example (Fig. 1). Note that the `SYNC` instruction at line `28` transfers the scalar `delta` from the Bohrium address space to the NumPy address space in order for the Python interpreter to evaluate the condition in the Jacobi Method code example (Fig. 1, line `9`).

This simple scheme has proven fairly efficient. However, the efficiency is closely linked to the ability of the bridge to send discards close to the last usage of an array in order to minimize the active memory footprint since this is a very scarce resource on the GPU.

The code generation we have in the GPU-VE simply translates every Bohrium instruction into exactly one line of OpenCL code.

F. Example

Figure 5 illustrate the list of vector byte code that the NumPy Bridge will generate when executing one of the iterations in the Jacobi Method code example (Fig. 1). The example demonstrates the nearly one-to-one mapping from the NumPy vector operations to the Bohrium vector byte code. The code generates seven temporary arrays (`t1`,...,`t7`) that are not specified in the code explicitly but is a result of how Python interprets the code. In a regular NumPy execution, the seven temporary arrays translate into seven memory allocations and de-allocations thus imposing an extra overhead. On the other hand, a Bohrium execution with the Victim Cache will only use two memory allocations since six of the temporary arrays (`t1`,...,`t6`) will use the same memory allocation. However, no writes to memory are eliminated. In the GPU-VE the source code generation eliminates the memory writes all together. (`t1`,...,`t5`) are stored only in registers. Without this strategy the speedup gain would no be possible on the GPU due to the memory bandwidth bottleneck.

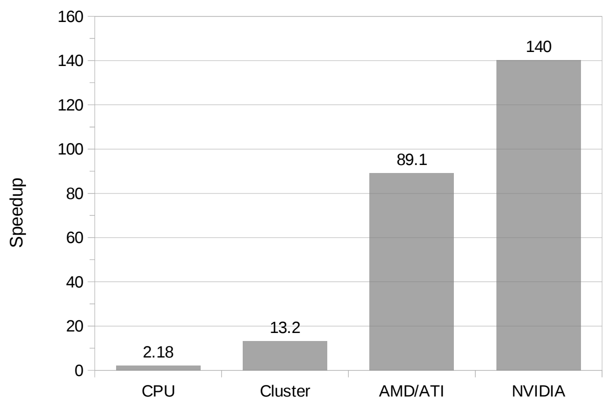


Fig. 6: Relative speedup of the Shallow Water application. For the CPU and Cluster, the application simulates a 2D domain with $25k^2$ value points in 10 iterations. For the GPUs, it is a 2k $\times$ 4k domain in 100 iterations.

Machine:	8-node Cluster	GPU Host
Processor:	AMD Opteron 6272	AMD Opteron 6274
Clock:	2.1 GHz	2.2 GHz
L3 Cache:	16MB	16MB
Memory:	128GB DDR3	128GB DDR3
Compiler:	GCC 4.6.3	GCC 4.6.3 & OpenCL 1.1
Network:	Gigabit Ethernet	N/A
Software:	Linux 3.2, Python 2.7, NumPy 1.6.1	

TABLE I: Machine Specifications

V. Preliminary Results

In order to demonstrate our Bohrium design we have implemented a basic Bohrium setup. This concretization of Bohrium is by no means exhaustive but only a proof-of-concept. The implementation supports Python/NumPy when executing on CPU, GPU, and Clusters. However, the implementation is preliminary and has a high degree of further optimization potential. In this section, we present a preliminary performance study of the implementation that consists of the following three representative scientific application kernels:

Shallow Water A simulation of a system governed by the shallow water equations. A drop is placed in a still container and the water movement is simulated in discrete time steps. It is a Python/NumPy implementation of a MATLAB application by Burkardt [22].

Black Scholes The Black-Scholes pricing model is a partial differential equation, which is used in finance for calculating price variations over time[23]. This implementation uses a Monte Carlo simulation to calculate the Black-Scholes pricing model.

N-Body A Newtonian N-body simulation is one that studies how bodies, represented by a mass, a location, and a velocity, move in space according to the laws of Newtonian physics. We use a straightforward algorithm that computes all body-body interactions, $O(n^2)$, with collisions detection.

We execute all three applications using four different hardware setups: one using a two CPUs, one using an eight-node cluster, one using a AMD GPU, and one using a NVIDIA GPU. The dual CPU setup uses one of the cluster-nodes whereas the two GPU setups use a similar AMD machine (Table I, II). For each benchmark/language, we compare the Bohrium execution with a native NumPy execution and calculate the speedup based on the average wall clock time of five executions. When executing on the PU, we use all CPU cores available likewise when executing on the eight-node cluster, we use all CPU cores available on the cluster-node. The input and output data is 64bit floating point for all executions. While measuring the performance, the variation of the timings did not exceed 1%.

The data set sizes are chosen to represent realistic workloads for a cluster and GPU setup respectively. The speedups reported are obtained by comparing the wall clock time of the original NumPy execution with the wall clock time for

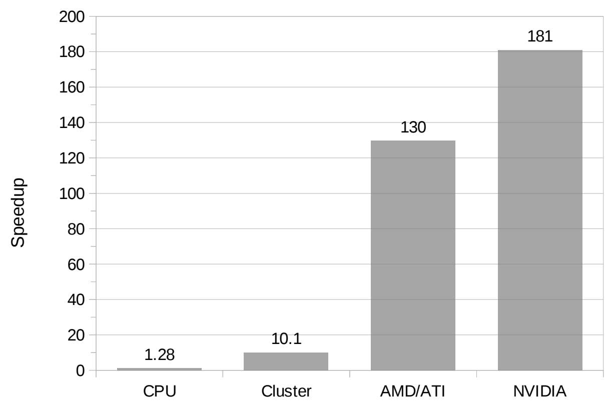


Fig. 7: Relative speedup of the Black Scholes application. For the CPU and Cluster, the application generates 10m element arrays using 10 iterations. For the GPUs, it generates 32m element arrays using 50 iterations.

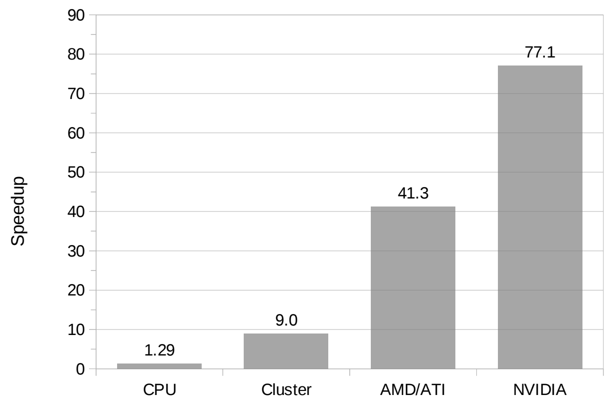


Fig. 8: Relative speedup of the N-Body application. For the CPU and Cluster, the application simulates 25k bodies in 10 iterations. For the GPUs, it is 1600 bodies and 50 iterations.

GPU:	AMD/ATI	NVIDIA
Processor:	ATI Radeon HD 7850	GeForce GTX 680
#Cores:	1024	1536
Core clock:	900 MHz	1006 MHz
Memory:	1GB DDR5	2GB DDR5
Memory bandwidth:	153 GB/s	192 GB/s
Peak (single-precision):	1761 GFLOPS	3090 GFLOPS
Peak (double-precision):	110 GFLOPS	128 GFLOPS

TABLE II: GPU Specifications

executing the same Python program with the same size of dataset.

A. Discussion

The Shallow Water application is memory intensive and uses many temporary arrays. This is clear when comparing the Bohrium execution with the Native NumPy execution on a single CPU. The Bohrium execution is 2.18 times faster than the Native NumPy execution primarily because of memory allocation reuse. The Cluster setup demonstrates good scalable performance as well. Even without communication latency hiding, it achieves a speedup of 6.07 compared to the CPU setup and 13.2 compared to Native NumPy. Finally, the two GPUs show an impressive 89 and 140 speedup, which demonstrates the efficiency of parallelizing array operations on a vector machine. NVIDIA is roughly one and a half times faster than AMD primarily because of the higher floating-point performance and memory bandwidth.

The Black Scholes application is computationally intensive and embarrassingly parallel, which is evident in the benchmark result. The cluster setup achieve a speedup of 10.1 compared to the Native NumPy and an almost linearly speedup of 7.91 compared to the CPU. Again, the performance of the GPUs is superior with a speedup of 130 and 181.

The N-Body application is memory intensive but does not use many temporary arrays thus the speedup of the CPU execution with the Native NumPy execution is only 1.29. However, the application scales well on the Cluster with a speedup of 9.0 compared to the Native NumPy execution and a speedup of 7.96 compared to the CPU execution. Finally, the two GPUs demonstrate a good speedup of 41.3 and 77.1 compared to the Native NumPy execution.

VI. Future Work

From the experiments, we can see that the performance is generally good. There is much room for further improvements when executing on the Cluster. Communication techniques, such as communication latency hiding and message aggregations, should improve performance[24], [25] further.

Despite the good results, we are convinced that we can still improve these results significantly. We are currently working on an internal representation for bytecode dependencies, which will enable us to rearrange the instructions and eliminate the use of temporary storage. In the article describing Intel Array Building Blocks, the authors report that the removal of temporary arrays is the single optimization that yields the greatest performance improvement. Informal testing with manual removal of temporary storage shows an order of magnitude improvement, even for simple benchmarks.

The GPU vector engine already uses a simple scanning algorithm that detects some instances of temporary vectors usage, as that is required to avoid exhausting the limited GPU memory. However, the internal representation will enable a better detection of temporary storage, but also enable loop detection and improve kernel generation and kernel reusability.

This internal representation will also allow pattern matching, which will allow selective replacement of parts of the instruction stream with optimized versions. This can be used to detect cases where the user is calculating a scalar sum, using a series of reductions, or detect matrix multiplications. By implementing efficient micro-kernels for known computations, we can improve the execution significantly.

Once these kernels are implemented, it is simple to offer them as function calls in the bridges. The bridge implementation can then simply implement the functionality by sending a pre-coded sequence of instructions.

We are also investigating the possibility of implementing a Bohrium Processing Unit, BPU, on FPGAs. With a BPU, we expect to achieve performance that rivals the best of todays GPUs, but with lower power consumption. As the FPGAs come with a built-in Ethernet support, they can also provide significantly lower latency, possibly providing real-time data analysis.

Finally, the ultimate goal of the Bohrium project is to support clusters of heterogeneous computation nodes where components specialized for GPUs, NUMA[6] aware multi-core CPUs, and Clusters, work together seamlessly.

VII. Conclusion

The declarative array-programming model used in Bohrium provides a framework for high-performance and high-productivity. It enables the end-user to execute regular Python/NumPy applications on a broad range of hardware architectures efficiently without any hardware specific knowledge. Furthermore, the Bohrium design supports scalable architectures such as clusters and supercomputers. It is even possible to combine architectures in order to exploit hybrid programming where multiple levels of parallelism exist, which is essential when fully utilizing supercomputers such as the Blue Gene/P[26].

In this paper, we introduce a proof-of-concept implementation of Bohrium that supports the Python programming language through a Bohrium implementation of NumPy and three computer architectures: CPU, GPU, and Cluster. The preliminary results are very promising – a *Black Scholes* computation achieves 181 times speedup for the same code, when comparing a Native NumPy execution and a Bohrium execution that utilize the GPU back-end.

The results are sufficiently good that we remain optimistic that we can reach a level where a pure Python/NumPy application offers sufficient performance on its own.

References

[1] G. van Rossum, "Glue it all together with python," in *Workshop on Compositional Software Architectures, Workshop Report, Monterey, California*, 1998.

[2] T. E. Oliphant, *A Guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[3] D. Loveman, "High performance fortran," *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 1, no. 1, pp. 25–42, 1993.

[4] W. Yang, W. Cao, T. Chung, and J. Morris, *Applied numerical methods using MATLAB*. Wiley-Interscience, 2005.

[5] C. Sanderson *et al.*, "Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments," Technical report, NICTA, Tech. Rep., 2010.

[6]Non-Uniform Memory Access

[6] T. Veldhuizen, "Arrays in Blitz++," in *Computing in Object-Oriented Parallel Environments*, ser. Lecture Notes in Computer Science, D. Caromel, R. Oldehoeft, and M. Tholburn, Eds. Springer Berlin Heidelberg, 1998, vol. 1505, pp. 223–230.

[7] J. Brown, W. Scullin, and A. Ahmadia, "Solving the import problem: Scalable dynamic loading network file systems," in *Talk at SciPy conference, Austin, Texas, July 2012*.

[8] J. Enkovaara, N. A. Romero, S. Shende, and J. J. Mortensen, "Gpaw-massively parallel electronic structure calculations with python-based software," *Procedia Computer Science*, vol. 4, pp. 17–25, 2011.

[9] A. Klckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157 – 174, 2012.

[10] R. Garg and J. N. Amaral, "Compiling python to a hybrid execution environment," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 19–30.

[11] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program gpus for general-purpose uses," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 325–335, Oct. 2006.

[12] C. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, "Intel's array building blocks: A retargetable, dynamic compiler and embedded language," in *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, 2011, pp. 224–235.

[13] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox, "Sejits: Getting productivity and performance with selective embedded jit specialization," *Programming Models for Emerging Architectures*, 2009.

[14] R. Andersen and B. Vinter, "The scientific byte code virtual machine," in *GCA'08*, 2008, pp. 175–181.

[15] B. Mailloux, J. Peck, and C. Koster, "Report on the algorithmic language algol 68," *Numerische Mathematik*, vol. 14, no. 2, pp. 79–218, 1969. [Online]. Available: http://dx.doi.org/10.1007/BF02163002

[16] S. Van Der Walt, S. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[17] L. S. Blackford, "ScaLAPACK," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing 96 Supercomputing 96*, 1996, p. 5.

[18] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[19] M. Kristensen and B. Vinter, "Managing communication latency-hiding at runtime for parallel programming languages and libraries," in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, 2012, pp. 546–555.

[20] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, may 1990, pp. 364 –373.

[21] S. A. F. Lund, K. Skovhede, M. R. B. Kristensen, and B. Vinter, "Doubling the Performance of Python/NumPy with less than 100 SLOC," in *4th Workshop on Python for High Performance and Scientific Computing (PyHPC'13)*, 2013.

[22] J. Burkardt, "Shallow water equations," people.sc.fsu.edu/\~jburkardt/m_src/shallow_water_2d/, [Online; accessed March 2010].

[23] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *The journal of political economy*, pp. 637–654, 1973.

[24] M. R. B. Kristensen and B. Vinter, "Numerical python for scalable architectures," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10. New York, NY, USA: ACM, 2010, pp. 15:1–15:9.

[25] M. R. B. Kristensen, Y. Zheng, and B. Vinter, "Pgas for distributed numerical python targeting multi-core clusters," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 680–690, 2012.

[26] M. Kristensen, H. Happe, and B. Vinter, "GPAW Optimized for Blue Gene/P using Hybrid Programming," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009, pp. 1–6.

6.8 NumCIL and Bohrium: High productivity and high performance

NumCIL and Bohrium: High productivity and high performance

Kenneth Skovhede and Simon Andreas Frimann Lund

Niels Bohr Institute, University of Copenhagen, Denmark
{skovhede,safl}@nbi.ku.dk

Abstract. In this paper, we explore the mapping of the NumCIL C# vector library where operations are offloaded to the Bohrium runtime system and evaluate the performance gains. By using a feature-rich language, such as C#, we argue that productivity can be increased. The use of the Bohrium runtime system allows all vector operations written in C# to be executed efficiently on multi-core systems.
We evaluate the presented design through a setup that targets a 32 core machine. The evaluation includes well-known benchmark applications, such as *Black Sholes*, *5-point stencil*, *Shallow Water*, and *N-body*.

Keywords: C# · NumCIL · Bohrium · High Performance · High Productivity · Vector Programming · Array Programming

1 Introduction

We have previously introduced the NumCIL library[13] for performing linear algebra in C#, using an approach known as vector programming, array programming or collection programming[12]. In such an approach, the programmer writes high-level operations on multidimensional vectors rather than looping over the individual elements. One of the primary benefits of such an approach is that it leaves the program more readable because it is more of a description of what should be done, rather than how it should be done. This approach can greatly speed up the development cycle, as the developer can focus on the structure of compact expressions, rather than explictily specify details such as loop indicies.

The Bohrium runtime system[10] is a related project aiming to deliver architecture specific optimizations. In Bohrium, a program will use the C or C++ interface to describe multidimensional vectors and request various operations on these. The execution of these operations is deferred until the program requires access to the result. This lazy evaluation approach enables the Bohrium runtime to collect a number of scheduled instructions and perform optimizations on these. The optimizations are an ongoing research project.

Since Bohrium uses a common intermediate representation of the scheduled operations, it is possible to apply different optimization strategies to different execution targets. The Bohrium intermediate representation also enables execution of Bohrium bytecode on multi-core CPU's, GPGPU's and even cluster setups.

In this article, we only evaluate the performance using a multi-core CPU. A more detailed description of the Bohrium system is available in *M. Kristensen et al.*[10].

By adding an extension to the NumCIL library, the vector operations expressed in C# can be forwarded to the Bohrium runtime system. This enables the programmer to have a rapid development cycle, without even having Bohrium installed. Once the program is tested for correctness, the unmodified program can then be executed with Bohrium support, such that all vector operations are executed with an efficient multi-core implementation.

2 Related Work

The array programming approach is in widespread use over a number of different programming languages, including Ada[5], CoArray Fortran[8], Chapel[3], NumPy[9] and numerous others. The NumPy approach differs in that it has no explicit support in Python but is implemented using Pythonic constructs in such a way that it seems *natural* to Python programmers. This approach means that nothing needs to change, in the Python programmers toolchain, to take advantage of the array programming found in NumPy. This non-intrusive approach with a natural language integration is the inspiration for the NumCIL library.

The idea of using language features to add support for vector programming instead of modifying the language is also found in the C++ libraries Armadillo[11] and Blitz++[16]. The Armadillo library leverages existing linear algebra systems to achieve high performance but does so at template instantiation time, rather than at runtime.

The RyuJIT[4] compiler adds support for smaller vectors by converting vector operations to SIMD instructions. This approach helps in handling memory access and accelerates the execution time, but does require changes to the runtime system and does not offer any features for larger arrays. The RuyJIT is scheduled to ship with Microsoft's .Net framework 5[4]. The Mono runtime[17] offers the Mono.Simd library with similar capabilities, implemented as a library with special support from the runtime[18].

The ideas for providing an intermediate representation of the requested operations, and performing optimizations on this, are also found in the, now discontinued, Intel Array Building Blocks (ArBB) project[7]. The ArBB system relies on a special compiler and an extended C++ syntax to describe computational kernels. When executing a batch of instructions, a number of optimization techniques are applied, such as removal of scratch memory, loop fusion, etc.

The Bohrium runtime system[10] is similar to ArBB and Chapel, in that the programmer uses vectors and describes *what* should be done, rather than *how* it is done. Internally this is achieved by means of a vector-oriented byte-code, i.e. simple instructions for a pseudo vector processing system. This abstraction allows Bohrium to be programming language agnostic, and is used to express a flat `C` API . With this API, it is possible to support a number of programming languages, such as Python, C++ and C#, in which the developer uses some array-library to interact with Bohrium.

The programming model used by Bohrium and NumCIL is very similar to the one found in NumPy[9], for which there also exists a Bohrium interface. In that sense, NumCIL fills the same role as NumPy, by providing an abstraction for interacting with Bohrium.

3 Implementation

The NumCIL library consists of three main item types: Multidimensional views, data storage and operators. The views are applied to the data storage to select a subset of the flat data storage, and project it into multiple dimensions, using offset, stride and skip values. Applied operators affect only the subset of the data that view projects, which greatly reduces the need for copying data into appropriately sized containers. The implementation of the multidimensional views found in NumCIL are compatible with NumPy's ndarrays[9] and also the Bohrium data views.

The primary design goal for the Bohrium extension to NumCIL has been to allow a non-intrusive addition. This allows code already written and tested with NumCIL to use the Bohrium runtime system without any changes. The non-intrusive design is achieved by hooking into the `DataAccessor` class, which is normally a simple wrapper for an array. By replacing the NumCIL factory instance that produces `DataAccessor` items, it becomes possible to provide Bohrium enabled data accessors.

Table 1 shows a simple multidimensional program written with NumCIL. It illustrates how a flat array can be projected into multiple dimensions, and how the data can be *broadcasted* into larger dimensions. The program can be executed in Bohrium, simply by adding the statement `NumCIL.Bohrium.Utility.Activate();` prior to running the code.

If the program in table 1 is executed with Bohrium loaded, the variable "`a`" will not be allocated until it is needed in the very last line. In that very last line, the allocation, multiplication, addition and summation is executed in Bohrium as a single instruction batch. Depending on the Garbage Collector, the batch may or may not contain instructions to deallocate the memory as well.

When a Bohrium enabled data accessor is created, it can be created with or without existing data. If there is no existing data, as with "`a`", an empty array is allocated by the Bohrium system and a handle for this is maintained by the data accessor. If existing data is already present, as with "`c`", the data accessor behaves as a non-Bohrium enabled data accessor facilitating access to the array data. This ensures that data is always kept where it is already allocated and not copied needlessly.

When an operation is applied to a multidimensional view that is referencing a Bohrium enabled data accessor, such as the multiplication, the views involved are created in Bohrium and an instruction matching the requested operation is emitted to the Bohrium runtime system. However, emitting the operation does

C# code	Resulting data
`using NumCIL.Float32;`	
`...`	
`var a = Generate.Range(3);`	`[0, 1, 2]`
`var b = a[Range.All, Range.NewAxis];`	`[[0], [1], [2]]`
`var data = new float[] { 2, 3 };`	`[2, 3]`
`var c = new NdArray(data);`	`[2, 3]`
`var d =`	`[[0, 0], [2, 3], [4, 6]] =`
`       b`	`    [[0 ,0], [1, 1], [2, 2]]`
`       *`	`    *`
`       c;`	`    [[2, 3], [2, 3], [2, 3]]`
`Console.WriteLine((d`	`[[0, 0], [2, 3], [4, 6]]`
`                     + 1)`	`    + [[1, 1], [1, 1], [1, 1]]`
`                         .Sum());`	`    = [[1, 1], [3, 4], [5, 7]] = 15`

Table 1: A simple vector program with NumCIL

nothing more than adding the operations to the current batch. Since the CLR is using a garbage collected approach, there is a chance that the GC will run before the operations are executed. If the GC runs, it can reuse the memory occupied by non-referenced items, and it may also choose to move existing data to a new location, and thus invalidating a pointer to the data. This problem is exacerbated by the introduction of temporary storage when compiling a composite statement as shown in table 2.

Composite expression	Expansion to single expressions
	`var e = Generate.Range(10);`
`var e = Generate.Range(10);`	`var t0 = e + 10;`
`var f = ((e + 10) * e) - 1;`	`var t1 = t0 * e;`
	`var f = t1 - 1;`

Table 2: A composite expression and the equivalent single expression version

All of the temporary variables shown in table 2 will be short lived and eliminated when the GC runs. In order to avoid issues with the GC, it is possible to `Pin` the memory when obtaining a pointer to the data. As long as the pointer is `Pinned`, the GC will not attempt to move or reuse the data. Since multiple multidimensional views may point to the same data, as with "`a`" and "`b`", a reference counting scheme is used to defer the Unpinning until the last reference is out of scope. This further ensures that data is not copied but used where it is located, with minimal overhead.

When a Bohrium enabled data accessor is created without existing data, only the view data is initialized, and the data storage is kept uninitialized. When the operations eventually execute, the Bohrium runtime will allocate only the needed data. This allows for using more memory than what is physically available on the machine with no side effects.

When data is requested by the CIL, i.e. for the summation operation which returns a scalar, all pending operations need to execute to ensure that the data observed by the application is seeing the expected results. This is accomplished by performing a Bohrium sync command on the target data, and then requesting a flush of all pending instructions.

In the case where the data being requested is not backed by a CIL array, an extra instruction is inserted that will copy the data allocated by Bohrium into a freshly created CIL array. This copy operation is done prior to the sync and flush commands, such that the intermediate storage can easily be eliminated by the Bohrium runtime system, thus allowing the results to be written directly to the CIL array.

If the user is only requesting a single element from the data, the entire data stays in the Bohrium allocated memory region, and only the requested element is copied into a CIL variable. This greatly reduces memory usage if only single elements are requested in a large array, such as when reading only the border values. If the user is writing to a single element in data that is not backed by a CIL array, all pending operations are flushed before writing the element directly into the memory region allocated by Bohrium. Table 3 shows the different states the `DataAccessor` goes through.

Created with		**Used in operation**		**Access element**		**Access array**
No data	→	create bh handle	→	flush read from pointer	→	create new array emit copy free bh handle flush convert to CIL return array
CIL array	→	pin create bh handle	→	flush unpin read array	→	flush unpin return array

Table 3: State flow for a Bohrium enabled NumCIL `DataAccessor`

4 Results

To evaluate the performance of the library, we have implemented a number of computational cores for classic simulations. The benchmarks are all implemented in C# and run using Mono 3.2.8 on Ubuntu 14.04.02. In order to provide a reasonable baseline, the benchmarks are also implemented with NumPy 1.8.2 and executed with Python 2.7.6. The hardware platform has two AMD Opteron 6272 CPUs with a total of 32 cores and 128 GB DDR3 memory with 4 memory busses. GCC version 4.8.2 was used to compile the Bohrium runtime. Source code is available for the Bohrium and NumCIL packages[15], as well as for the benchmarks[14].

Various options were used when executing the C# benchmarks. The basic `Managed` mode is using only C# and CIL functionality. The `Unsafe` configuration, utilizes the option to bypass array bounds checks within the CIL runtime, by accessing the data through memory pointers, with so-called `unsafe code`. The `Unsafe` configuration does not appear to influence the execution times on Mono, but is shown here as it does have an effect on the Microsoft .Net runtime[13]. When executing the benchmarks with Bohrium enabled, the number of utilized threads are varied to give an indication of the scalability.

The NumPy versions execute faster than the C# versions of the same code in general. There are two main reasons for this. Firstly, the NumPy implementation is written mostly in C, which means that none of the Python overhead is present. Secondly, the Mono JIT compiler does not perform various optimizations, such as efficient function inlining. When using the Microsoft .Net runtime, the execution times are roughly half of the Mono results, and approximately 20% faster than the NumPy code[13]. On the Windows platform, we would expect NumCIL by itself to perform roughly 50% faster than the reported Mono results, but when coupled with Bohrium, only metadata is handled by the .Net runtime, and thus the obtained execution times would be the same.

4.1 General observations

The speedup does not exceed a factor of 4, even when using 32 cores. This limitation stems from the current execution mode in Bohrium, where each operation is executed individually. This approach has the effect that each operation will read all memory inputs and write all memory outputs for each operation, even if the inputs or outputs are needed for other operations. As the inputs and outputs are vectors, the caches are not utilized, effectively limiting the output to the bandwidth of the memory system.

This issue, and many other performance issues, can be mitigated through a technique known as *loop fusion*, where loop traversals are transposed, such that less memory access is required. Even though these optimizations are not yet implemented in Bohrium, we still see speedups. Once these optimizations are fully implemented in Bohrium, the NumCIL library will automatically perform even better.

4.2 Black-Scholes model

The Black-Scholes model is a financial method for estimating the price of stock options[1]. It can be considered an embarrassingly parallel computation kernel, similar to Monte-Carlo π, but with a heavier computational workload. As shown in figure 1, the performance gains from the Bohrium runtime are fairly low, due to the current configuration not being able to efficiently fuse the operations, causing a high load on the memory system. As the memory system is saturated, adding execution units does not improve performance.

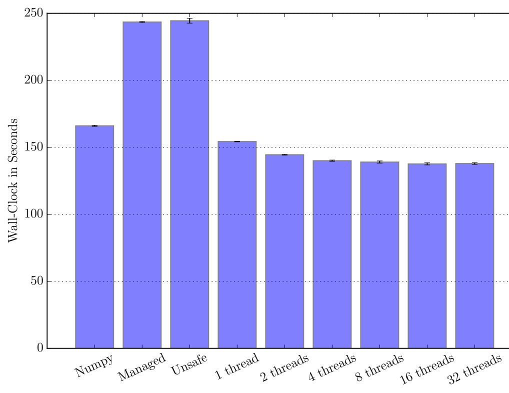


Fig. 1: BlackScholes 3200000, 36 iterations

4.3 Heat Equation

The Heat Equation benchmark is implemented as a 5-point stencil and simulates thermal dispersion in a material. The stencil is applied ten times to a 5000x5000 element array of single precision floating point numbers. The computation is simple additions, using multiple parallel accesses to the same memory. Even though the Mono implementation has some drawbacks and performs significantly slower than NumPy, the Bohrium runtime can re-use memory allocations, which allows for significant speedups[6]. Despite the low computational complexity, the Bohrium runtime can speed up execution when using all cores.

4.4 n-body simulation

The n-body simulation is implemented in a naive manner, yielding a $O = N^2$ complexity. For each time-step, the forces of all bodies on all bodies are computed, and their velocities and positions are updated. The Mono runtime slightly outperforms the NumPy version for this benchmark. When the Bohrium runtime is activated, it is capable of memory re-use and runs over twice as fast on a single core, with speedup on up to 16 threads.

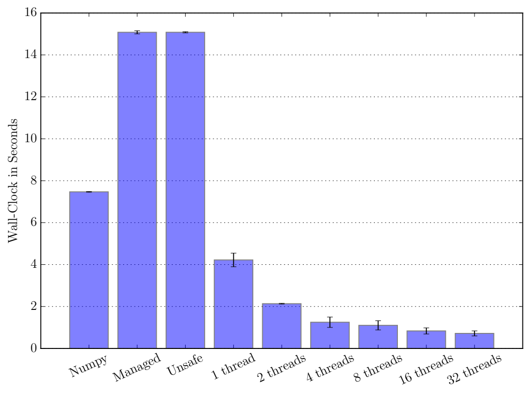


Fig. 2: HeatEquation 5000x5000, 10 iterations

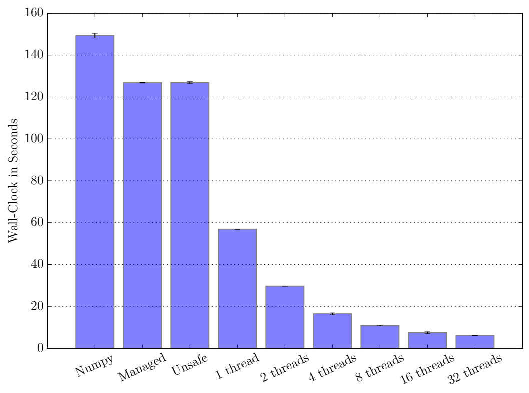


Fig. 3: nBody 5000, 10 iterations

4.5 Shallow water

The Shallow Water simulation[2] is performed on a grid of 5000 by 5000 single precision numbers, over ten discrete timesteps, simulating water movements. Many independent computations on each element dominate the computations, yielding irregular memory accesses. The NumPy implementation is more than twice as fast as the Mono version. The Bohrium runtime can improve this even further, by almost a factor of two, yielding a total speedup of four times, compared to the basic Mono performance.

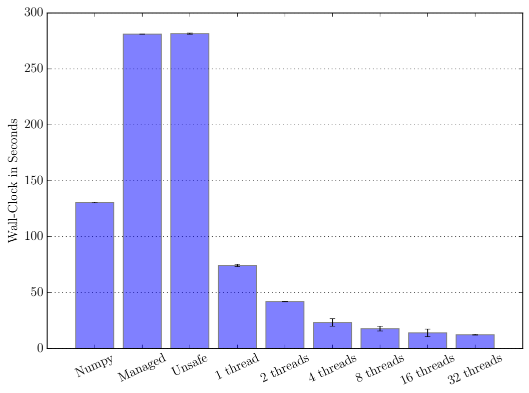


Fig. 4: Shallow water 5000x5000, 10 iterations

5 Conclusion

We have implemented and evaluated an extension to the NumCIL library, which enables completely transparent support for execution of existing programs with the Bohrium runtime system.

From the benchmarks, it is clear that even with the sub-par performance from the Mono JIT compiler, the Bohrium runtime system can deliver substantial speedups.

Given the high-level language features in C#, it is clear that the NumCIL library can be used for rapid development, and when paired with the Bohrium runtime, it also yields high performance.

Even with the speedups reported here, a number of additional optimizations are being developed for the Bohrium runtime, including loop fusion and NUMA-aware memory handling. Once these optimizations are implemented in Bohrium, the loosely coupled approach used in NumCIL will automatically give even greater performance boosts.

Acknowledgment

This research was supported by grant number 131-2014-5 from Innovation Fund Denmark. This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (hiperfit.dk) under contract number 10-092299.

References

1. Black, F., Scholes, M.: The pricing of options and corporate liabilities. The journal of political economy pp. 637–654 (1973)

2. Burkardt, J.: Shallow water equations. http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_2d/, [Online; accessed May 2015]
3. Chamberlain, B., Vetter, J.S.: An introduction to chapel: Cray cascades high productivity language. In: AHPCRC DARPA Parallel Global Address Space (PGAS) Programming Models Conference, Minneapolis, MN (2005)
4. Frei, K.: Ryujit ctp3: How to use simd. http://blogs.msdn.com/b/clrcodegeneration/archive/2014/04/03/ryujit-ctp3-how-to-use-simd.aspx, [Online; accessed May 2015]
5. Ichbiah, J.D., Krieg-Brueckner, B., Wichmann, B.A., Barnes, J.G., Roubine, O., Heliard, J.C.: Rationale for the design of the ada programming language. ACM Sigplan Notices 14(6b), 1–261 (1979)
6. Lund, S.A., Skovhede, K., Kristensen, M.R.B., Vinter, B.: Doubling the performance of python/numpy with less than 100 sloc. In: IEEE International Conference on Performance, Computing, and Communications (2013)
7. Newburn, C.J., So, B., Liu, Z., McCool, M., Ghuloum, A., Toit, S.D., Wang, Z.G., Du, Z.H., Chen, Y., Wu, G., et al.: Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In: Code generation and optimization (CGO), 2011 9th annual IEEE/ACM international symposium on. pp. 224–235. IEEE (2011)
8. Numrich, R.W., Reid, J.: Co-array fortran for parallel programming. In: ACM Sigplan Fortran Forum. vol. 17, pp. 1–31. ACM (1998)
9. Oliphant, T.E.: Python for scientific computing. Computing in Science & Engineering 9(3), 10–20 (2007), http://scitation.aip.org/content/aip/journal/cise/9/3/10.1109/MCSE.2007.58
10. R. B. Kristensen, M., A. F. Lund, S., Blum, T., Skovhede, K., Vinter, B.: Bohrium: a virtual machine approach to portable parallelism. In: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2014 IEEE 28th International. IEEE (2014)
11. Sanderson, C.: Armadillo: C++ linear algebra library. http://arma.sourceforge.net/, [Online; accessed May 2015]
12. Sipelstein, J.M., Blelloch, G.E.: Collection-oriented languages. Proceedings of the IEEE 79(4), 504–523 (1991)
13. Skovhede, K., Vinter, B.: Numcil: Numeric operations in the common intermediate language. Journal of Next Generation Information Technology 4(1), 9–18 (2013)
14. Team, B.: Benchpress source code. https://github.com/bh107/benchpress, [Online; accessed May 2015; used revision 349ce5c1a69bb723a76783f7720c6ff0874519af]
15. Team, B.: Bohrium source code. https://github.com/bh107/bohrium, [Online; accessed May 2015; used revision 6f27c1fb3ae46c9b2541ba6d15b44e4a02e2cb01]
16. Veldhuizen, T., Cummings, J.: Armadillo: C++ linear algebra library. http://blitz.sourceforge.net/, [Online; accessed May 2015]
17. Xamarin: Mono: Cross platform, open source .net framework. http://www.mono-project.com/, [Online; accessed May 2015]
18. Xamarin: Mono.simd namespace: Hardware accelerated simd-based primitives. http://api.xamarin.com/index.aspx?link=N%3AMono.Simd, [Online; accessed May 2015]

6.9 Separating NumPy API from Implementation

Separating NumPy API from Implementation

Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, and Kenneth Skovhede
Niels Bohr Institute, University of Copenhagen, Denmark
{madsbk/safl/blum/skovhede}@nbi.dk

Abstract—**In this paper, we introduce a unified backend framework for NumPy that combine a broad range of Python code accelerators with no modifications to the user Python/NumPy application. Thus, a Python/NumPy application can utilize hardware architecture such as multi-core CPUs and GPUs and optimization techniques such as Just-In-Time compilation and loop fusion without any modifications. The backend framework defines a number of primitive functions, including all existing ufuncs in NumPy, that a specific backend must implement in order to accelerate a Python/NumPy application. The framework then seamlessly translates the Python/NumPy application into a stream of calls to these primitive functions.**

In order to demonstrate the usability of our unified backend framework, we implement and benchmark four different backend implementations that use four different Python libraries: NumPy, Numexpr, libgpuarray, and Bohrium. The results are very promising with a speedup of up to 18 compared to a pure NumPy execution.

I. Introduction

Python is a high-level, general-purpose, interpreted language. Python advocates high-level abstractions and convenient language constructs for readability and productivity rather than high-performance. However, Python is easily extensible with libraries implemented in high-performance languages such as C and FORTRAN, which makes Python a great tool for gluing high-performance libraries together[1]. NumPy is the de-facto standard for scientific applications written in Python[2] and contributes to the popularity of Python in the HPC community. NumPy provides a rich set of high-level numerical operations and introduces a powerful array object. The array object is essential for scientific libraries, such as SciPy[3] and matplotlib[4], and a broad range of Python wrappers of external scientific libraries[5], [6], [7]. NumPy supports a declarative vector programming style where numerical operations applies to full arrays rather than scalars. This programming style is often referred to as vector or array programming and is commonly used in programming languages and libraries that target the scientific community, e.g. HPF[8], ZPL[9], MATLAB[10], Armadillo[11], and Blitz++[12].

NumPy does not make Python a high-performance language but through array programming it is possible to achieve performance within one order of magnitude of C. In contrast to pure Python, which typically is more than hundred if not thousand times slower than C. However, NumPy does not utilize parallel computer architectures when implementing basic array operations; thus only through external libraries, such as BLAS or FFTW, is it possible to utilize data or task parallelism.

In this paper, we introduce a unified NumPy backend that enables seamless utilization of parallel computer architecture such as multi-core CPUs, GPUs, and Clusters. The framework exposes NumPy applications as a stream of abstract array operations that architecture-specific computation backends can execute in parallel without the need for modifying the original NumPy application.

The aim of this new unified NumPy backend is to provide support for a broad range of computation architectures with minimal or no changes to existing NumPy applications. Furthermore, we insist on legacy support (at least back to version 1.6 of NumPy), thus we will not require any changes to the NumPy source code itself.

II. Related Work

Numerous projects strive to accelerate Python/NumPy applications through very different approaches. In order to utilize the performance of existing programming languages, projects such as Cython[13], IronPython[14], and Jython[15], introduce static source-to-source compilation to C, .NET, and Java, respectively. However, none of the projects are seamlessly compatible with Python – Cython extends Python with static type declarations whereas IronPython and Jython do not support third-party libraries such as NumPy.

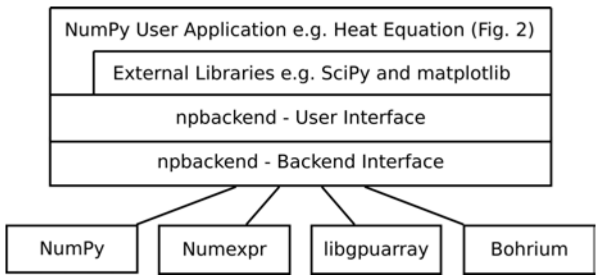
PyPy[16] is a Python interpreter that makes use of Just-in-Time (JIT) compilation in order to improve performance. PyPy is also almost Python compliant, but again PyPy does not support libraries such as NumPy fully and, similar to IronPython and Jython, it is not possible to fall back to the original Python interpreter CPython when encountering unsupported Python code.

Alternatively, projects such as Weave[17], Numexpr[18], and Numba[19] make use of JIT compilation to accelerate parts of the Python application. Common for all of them is the introduction of functions or decorators that allow the user to specify acceleratable code regions.

In order to utilize GPGPUs the PyOpenCL and PyCUDA projects enable the user to write GPU kernels directly in Python[20]. The user writes OpenCL[21] or CUDA[22] specific kernels as text strings in Python, which simplifies the utilization of OpenCL or CUDA compatible GPUs but still requires OpenCL or CUDA programming knowledge. Less intrusively, libgpuarray, which is part of the Theano[23] project, introduces GPU arrays on which all operations execute on the GPU. The GPU arrays are similar to NumPy arrays but are not a drop-in replacement.

III. The Interface

The interface of our unified NumPy backend (npbackend) consists of two parts: a user interface that facilitates the end NumPy user and a backend interface that facilitates the



NumPy User Application e.g. Heat Equation (Fig. 2)
External Libraries e.g. SciPy and matplotlib
npbackend - User Interface
npbackend - Backend Interface
NumPy
Numexpr
libgpuarray
Bohrium

Fig. 1: The Software Stack

.

backend writers (Fig. 1). The source code of both interfaces and all backend implementations is an available at the Bohrium project's website[1] for further inspection. In the following two subsections, we present the two interfaces.

A. *The User Interface*

The main design objective of the user interface is easy transition from regular NumPy code to code that utilizes a unified NumPy backend. Ideally, there should be no difference between NumPy code with or without a unified NumPy backend. Through modifications of the NumPy source code, the DistNumPy[24] and Bohrium[25] projects demonstrate that it is possible to implement an alternative computation backend that does not require any changes to the user's NumPy code. However, it is problematic to maintain a parallel version of NumPy that contains complex modifications to numerous parts of the project, particularly when we have to fit each modification to a specific version of NumPy (version 1.6 through 1.9).

As a consequence, instead of modifying NumPy, we introduce a new Python module *npbackend* that implements an array object that inherit from NumPy's ndarray. The idea is that this new npbackend-array can be a drop-in replacement of the numpy-array such that only the array object in NumPy applications needs to be changed. Similarly, the npbackend module is a drop-in replacement of the NumPy module.

The user can make use of npbackend through an explicit and an implicit approach. The user can explicitly import npbackend instead of NumPy in the source code e.g. "`import npbackend as numpy`" or the user can alias NumPy imports with npbackend imports globally through the `-m` interpreter argument e.g. "`python -m npbackend user_app.py`".

Even though the npbackend is a drop-in replacement, the backend might not implement all of the NumPy API, in which case npbackend will gracefully use the original NumPy implementation. Since npbackend-array inherits from numpy-array, the original NumPy implementation can access and apply operations on the npbackend-array seamlessly. The result is that a NumPy application can utilize an architecture-specific backend with minimal or no modification. However, npbackend does not guarantee that all operations in the application will utilize the backend — only the ones that the backend support.

[1]http://bh107.org

```
 1 import npbackend as np
 2 import matplotlib.pyplot as plt
 3
 4 def solve(height, width, epsilon=0.005):
 5   grid = np.zeros((height+2,width+2),dtype=np.float64)
 6   grid[:,0]   = -273.15
 7   grid[:,-1]  = -273.15
 8   grid[-1,:]  = -273.15
 9   grid[0,:]   = 40.0
10   center = grid[1:-1,1:-1]
11   north  = grid[:-2,1:-1]
12   south  = grid[2:,1:-1]
13   east   = grid[1:-1,:-2]
14   west   = grid[1:-1,2:]
15   delta  = epsilon+1
16   while delta > epsilon:
17     tmp = 0.2*(center+north+south+east+west)
18     delta = np.sum(np.abs(tmp-center))
19     center[:] = tmp
20   plt.matshow(center, cmap='hot')
21   plt.show()
```

Fig. 2: Python implementation of a heat equation solve that uses the finite-difference method to calculate the heat diffusion. Note that we could replace the first line of code with "`import numpy as np`" and still utilize npbackend through the command line argument "`-m`", e.g. "`python -m npbackend heat2d.py`"

Figure 2, is an implementation of a heat equation solver that imports the npbackend module explicitly at the first line and a popular visualization module, Matplotlib, at the second line. At line 5, the function `zeros()` creates a new npbackend-array that overloads the arithmetic operators, such as `*` and `+`. Thus, at line 17 the operators use npbackend rather than NumPy. However, in order to visualize (Fig. 3) the `center` array at line 20, Matplotlib accesses the memory of `center` directly.

Now, in order to explain what we mean by *directly*, we have to describe some implementation details of NumPy. A NumPy ndarray is a C implementation of a Python class that exposes a segment of main memory through both a C and a Python interface. The ndarray contains metadata that describes how the memory segment is to be interpreted as a multi-dimensional array. However, only the Python interface seamlessly interprets the ndarray as a multi-dimensional array. The C interface provides a C-pointer to the memory segment and lets the user handle the interpretation. Thus, with the word *directly* we mean that Matplotlib accesses the memory segment of `center` through the C-pointer. In which case, the only option for npbackend is to make sure that the computed values of `center` are located at the correct memory segment. Npbackend is oblivious to the actual operations Matplotlib performs on `center`.

Consequently, the result of the Matplotlib call is a Python warning explaining that npbackend will not accelerate the operation on `center` at line 20; instead the Matplotlib implementation will handle the operation exclusively.

B. *The Backend Interface*

The main design objective of the backend interface is to isolate the calculation-specific from the implementation-specific. In order to accomplish this, we translate a NumPy execution into a sequence of primitive function calls, which the backend must implement.

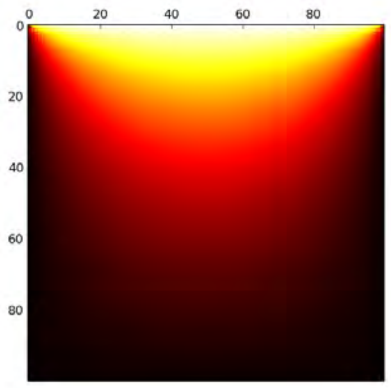


Fig. 3: The *matplotlib* result of executing the heat equation solver from figure 2: `solve(100,100)`

Figure 4 is the abstract Python module that a npbackend must implement. It consists of two Python classes, `base` and `view`, that represent a memory sequence and a multi-dimensional array-view thereof. Since this is the abstract Python module, the base class does not refer to any physical memory but only a size and a data type. In order to implement a backend, the base class could, for example, refer to the main memory or GPU memory. Besides the two classes, the backend must implement eight primitive functions. Seven of the functions are self-explanatory (Fig. 4), however the `extmethod()` function requires some explanation. In order to support arbitrary NumPy operations, npbackend introduces an Extension Method that passes any operations through to the backend. For example, it is not convenient to implement operations such as matrix multiplication or FFT only using ufuncs; thus we define an Extension Method called *matmul* that corresponds to a matrix multiplication. Now, if a backend knows the *matmul* operation it should perform a matrix multiplication. On the other hand, if the backend does not know *matmul* it must raise a `NotImplementedError` exception.

IV. The Implementation

The implementation of npbackend consists primarily of the new npbackend-array that inherits from NumPy's numpy-array. The npbackend-array is implemented in C and uses the Python-C interface to inherit from numpy-array. Thus, it is possible to replace npbackend-array with numpy-array both in C and in Python — a feature npbackend must support in order to support code such as the heat equation solver in figure 2.

As is typical in object-oriented programming, the npbackend-array exploits the functionality of numpy-array as much as possible. The original numpy-array implementation handles metadata manipulation, such as slicing and transposing; only the actual array calculations will be handled by the npbackend. The npbackend-array overloads arithmetic operators thus an operator on npbackend-arrays will call the backend function `ufunc` (Fig. 4 Line 26). Furthermore, since npbackend-arrays inherit from numpy-array, an operator on a mix of the two array classes will also use the backend function.

```
 1 """Abstract module for computation backends"""
 2
 3 class base(object):
 4   """Abstract base array handle (an array has only one ↩
        base array)"""
 5   def __init__(self, size, dtype):
 6     self.size = size #Total number of elements
 7     self.dtype = dtype #Data type
 8
 9 class view(object):
10   """Abstract array view handle"""
11   def __init__(self, ndim, start, shape, stride, base):
12     self.ndim = ndim #Number of dimensions
13     self.shape = shape #Tuple of dimension sizes
14     self.base = base #The base array this view refers to
15     self.start = start*base.dtype.itemsize #Offset from ↩
          base (in bytes)
16     self.stride = [x*base.dtype.itemsize for x in stride] ↩
          #Tuple of strides (in bytes)
17
18 def get_data_pointer(ary, allocate=False, nullify=False):
19   """Return a C-pointer to the array data (as a Python ↩
        integer)"""
20   raise NotImplementedError()
21
22 def set_data_from_ary(self, ary):
23   """Copy data from 'ary' into the array 'self'"""
24   raise NotImplementedError()
25
26 def ufunc(op, *args):
27   """Perform the ufunc 'op' on the 'args' arrays"""
28   raise NotImplementedError()
29
30 def reduce(op, out, a, axis):
31   """Reduce 'axis' dimension of 'a' and write the result ↩
        to out"""
32   raise NotImplementedError()
33
34 def accumulate(op, out, a, axis):
35   """Accumulate 'axis' dimension of 'a' and write the ↩
        result to out"""
36   raise NotImplementedError()
37
38 def extmethod(name, out, in1, in2):
39   """Apply the extended method 'name'"""
40   raise NotImplementedError()
41
42 def range(size, dtype):
43   """Create a new array containing the values [0:size["""
44   raise NotImplementedError()
45
46 def random(size, seed):
47   """Create a new random array"""
48   raise NotImplementedError()
```

Fig. 4: The backend interface of npbackend.

However, NumPy functions in general will not make use of the npbackend backend since many of them uses the C-interface to access the array memory directly. In order to address this problem, npbackend has to re-implement much of the NumPy API, which is a lot of work and is prone to error. However, we can leverage the work by the PyPy project; PyPy does not support the NumPy C-interface either but they have re-implemented much of the NumPy API already. Still, the problem goes beyond NumPy; any library that makes use of the NumPy C-interface will have to be rewritten.

The result is that the npbackend implements all array creation functions, matrix multiplication, random, FFT, and all ufuncs for now. All other functions that access array memory directly will simply get unrestricted access to the memory.

A. Unrestricted Direct Memory Access

In order to detect and handle direct memory access to arrays, npbackend uses two address spaces for each array memory: a user address space visible to the user interface and a backend address space visible to the backend interface. Initially, the user address space of a new array is memory protected with `mprotect` such that subsequent accesses to the memory will trigger a segmentation fault. In order to detect and handle direct memory access, npbackend can then handle this kernel signal by transferring array memory from the backend address space to the user address space. In order to get access to the backend address space memory, npbackend calls the `get_data_pointer()` function (Fig. 4, Line 18). Similarly, npbackend calls the `set_data_from_ary()` function (Fig. 4, Line 22) when the npbackend should handle the array again.

In order to make the transfer between the two address spaces, we use `mremap` rather than the more expensive `memcpy`. However, `mremap` requires that the source and destination are memory page aligned. That is not a problem at the backend since the backend implementer can simply use `mmap` when allocating memory; on the other hand, we cannot change how NumPy allocates its memory at the user address space. The solution is to re-allocate the array memory when the constructor of npbackend-array is called using `mmap`. This introduces extra overhead but will work in all cases with no modifications to the NumPy source code.

V. Backend Examples

In order to demonstrate the usability of npbackend, we implement four backends that use four different Python libraries: NumPy, Numexpr, libgpuarray, and Bohrium, all of whom are standalone Python libraries in their own right. In this section, we will describe how the four backends implement the eight functions that make up the backend interface (Fig. 4).

A. *NumPy Backend*

In order to explore the overhead of npbackend, we implement a backend that uses NumPy i.e. NumPy uses NumPy through npbackend. Figure 5 is a code snippet of the implementation that includes the `base` and `view` classes, which inherit from the abstract classes in figure 4, the three essential functions `get_data_pointer()`, `set_data_from_ary()`, and `ufunc()`, and the Extension Method function `extmethod()`.

The NumPy backend associates a NumPy view (`.ndarray`) with each instance of the `view` class and an `mmap` object for each `base` instance, which enables memory allocation reuse and guarantees memory-page-aligned allocations. In [26] the authors demonstrate performance improvement through memory allocation reuse in NumPy. The NumPy backend uses a similar technique[2] where it preserves a pool of memory allocations for recycling. The constructor of `base` will check this memory pool and, if the size matches, reuse the memory allocation (line 11-15).

The `get_data_pointer()` function simply returns a C-pointer to the ndarray data. The `set_data_from_ary()` function `memmove`s the data from the ndarray *ary* to the `view` *self*. The `ufunc()` function simply calls the NumPy library with the corresponding ufunc. Finally, the `extmethod()` recognizes the *matmul* method and calls NumPy's `dot()` function.

```
 1 import numpy
 2 import backend
 3 import os
 4
 5 VCACHE_SIZE = int(os.environ.get("VCACHE_SIZE", 10))
 6 vcache = []
 7 class base(backend.base):
 8   def __init__(self, size, dtype):
 9     super(base, self).__init__(size, dtype)
10     size *= dtype.itemsize
11     for i, (s,m) in enumerate(vcache):
12       if s == size:
13         self.mmap = m
14         vcache.pop(i)
15         return
16     self.mmap = mmap.mmap(-1, size)
17     def __str__(self):
18       return "<base memory at %s>"%self.mmap
19     def __del__(self):
20       if len(vcache) < VCACHE_SIZE:
21           vcache.append((self.size*self.dtype.itemsize, ↩
                self.mmap))
22
23 class view(backend.view):
24   def __init__(self, ndim, start, shape, stride, base):
25     super(view, self).__init__(ndim, start, shape, stride,↩
           base)
26     buf = np.frombuffer(self.base.mmap, dtype=self.dtype, ↩
          offset=self.start)
27     self.ndarray = np.lib.stride_tricks.as_strided(buf, ↩
          shape, self.stride)
28
29 def get_data_pointer(ary, allocate=False, nullify=False):
30   return ary.ndarray.ctypes.data
31
32 def set_data_from_ary(self, ary):
33   d = get_data_pointer(self, allocate=True, nullify=False)
34   ctypes.memmove(d, ary.ctypes.data, ary.dtype.itemsize * ↩
        ary.size)
35
36 def ufunc(op, *args):
37   args = [a.ndarray for a in args]
38   f = eval("numpy.%s"%op)
39   f(*args[1:], out=args[0])
40
41 def extmethod(name, out, in1, in2):
42   (out, in1, in2) = (out.ndarray, in1.ndarray, in2.ndarray↩
        )
43   if name == "matmul":
44     out[:] = np.dot(in1, in2)
45   else:
46     raise NotImplementedError()
```

Fig. 5: A code snippet of the NumPy backend. Note that the `backend` module refers to the implementation in figure 4.

B. *Numexpr Backend*

In order to utilize multi-core CPUs, we implement a backend that uses the Numexpr library, which in turn utilize Just-In-Time (JIT) compilation and shared-memory parallelization through OpenMP.

Since Numexpr is compatible with NumPy `ndarrays`, the Numexpr backend can inherit most functionality from the NumPy backend; only the `ufunc()` implementation differs. Figure 6 is a code snippet that includes the `ufunc()` implementation where it uses `numexpr.evaluate()` to evaluate a ufunc operation. Now, this is a very naïve implementation since we only evaluate one operation at a time. In order to maximize performance of Numexpr, we could collect as many ufunc operations as possible into one `evaluate()`

[2]Using a victim cache

```
ufunc_cmds = {'add'      : "i1+i2",
              'multiply' : "i1*i2",
              'sqrt'     : "sqrt(i1)",
              #...
              }

def ufunc(op, *args):
  args = [a.ndarray for a in args]
  i1=args[1];
  if len(args) > 2:
    i2=args[2]
  numexpr.evaluate(ufunc_cmds[op], \
                   out=args[0], casting='unsafe')
```

Fig. 6: A code snippet of the Numexpr backend.

```
 1 import pygpu
 2 import backend_numpy
 3 class base(backend_numpy.base):
 4   def __init__(self, size, dtype):
 5     self.clary = pygpu.empty((size,), dtype=dtype, cls=↩
         elemary)
 6     super(base, self).__init__(size, dtype)
 7
 8 class view(backend_numpy.view):
 9   def __init__(self, ndim, start, shape, stride, base):
10     super(view, self).__init__(ndim, start, shape, stride,↩
         base)
11     self.clary = pygpu.gpuarray.from_gpudata(base.clary.↩
          gpudata, offset=self.start, dtype=base.dtype, ↩
          shape=shape, strides=self.stride, writable=True, ↩
          base=base.clary, cls=elemary)
12
13 def get_data_pointer(ary, allocate=False, nullify=False):
14   ary.ndarray[:] = np.asarray(ary.clary)
15   return ary.ndarray.ctypes.data
16
17 def set_bhc_data_from_ary(self, ary):
18   self.clary[:] = pygpu.asarray(ary)
19
20 def ufunc(op, *args):
21   args = [a.ndarray for a in args]
22   out=args[0]
23   i1=args[1];
24   if len(args) > 2:
25     i2=args[2]
26   cmd = "out[:] = %s"%ufunc_cmds[op]
27   exec cmd
28
29 def extmethod(name, out, in1, in2):
30   (out, in1, in2) = (out.clary, in1.clary, in2.clary)
31   if name == "matmul":
32     pygpu.blas.gemm(1, in1, in2, 1, out, overwrite_c=True)
33   else:
34     raise NotImplementedError()
```

Fig. 7: A code snippet of the ligpuarray backend (the Python binding module is called `pygpu`). Note that the `backend_numpy` module refers to the implementation in figure 5 and note that `ufunc_cmds` is from figure 6.

call, which would enable Numexpr to fuse multiple ufunc operations together into one JIT compiled computation kernel. However, such work is beyond the focus of this paper – in this paper we map the libraries directly.

C. *Libgpuarray Backend*

In order to utilize GPUs, we implement a backend that makes use of libgpuarray, which introduces a GPU-array that is compatible with NumPy's ndarray. For the two classes, `base` and `view`, we associate a GPU-array that points to memory on the GPU; thus the user address space lies in main memory and the backend address space lies in GPU-memory. Consequently, the implementation of the two functions `get_data_pointer()` and `set_data_from_ary()` uses `asarray()` to copy between main memory and GPU-memory (Fig. 7 Line 14 and 15). The implementation of `ufunc()` is very similar to the Numexpr backend implementation since GPU-arrays supports ufunc directly. However, note that libgpuarray does not support the output argument, which means we have to copy the result of an ufunc operation into the output argument.

The `extmethod()` recognizes the *matmul* method and calls Libgpuarray's `blas.gemm()` function.

D. *Bohrium Backend*

Our last backend implementation uses the Bohrium runtime system to utilize both CPU and GPU architectures. Bohrium supports a range of frontend languages including C, C++, and CIL[3], and a range of backend architectures including multi-core CPUs through OpenMP and GPUs through OpenCL. The Bohrium runtime system utilizes the underlying architectures seamlessly. Thus, as a user we use the same interface whether we utilize a CPU or a GPU. The interface of Bohrium is very similar to NumPy – it consists of a multidimensional array and the same ufuncs as in NumPy.

The Bohrium backend implementation uses the C interface of Bohrium, which it calls directly from Python through SWIG[27]. The two `base` and `view` classes points to a Bohrium multidimensional array called `.bhc_obj` (Fig. 8). In order to use the Bohrium C interface through SWIG, we dynamically construct a Python string that matches a specific C function in the Bohrium C interface.

The `set_bhc_data_from_ary()` function is identical to the one in the NumPy backend. However, `get_data_pointer()` needs to synchronize the array data before returning a Python pointer to the data. This is because the Bohrium runtime system uses lazy evaluation in order to fuse multiple operations into single kernels. The synchronize function (Fig. 8 Line 34) makes sure that all pending operations on the array have been executed and that the array data is in main memory, e.g. copied from GPU-memory.

The implementations of `ufunc()` and `extmethod()` simply call the Bohrium C interface with the Bohrium arrays (`.bhc_obj`).

Processor:	Intel Xeon E5640
Clock:	2.66 GHz
L3 Cache:	12MB
Memory:	96GB DDR3
GPU:	Nvidia GeForce GTX 460
GPU-Memory:	1GB DDR5
Compiler:	GCC 4.8.2 & OpenCL 1.2
Software:	Linux 3.13, Python 2.7, & NumPy 1.8.1

TABLE I: The Machine Specification

VI. BENCHMARKS

In order to evaluate the performance of npbackend, we perform a number of performance comparisons between a

[3]Common Intermediate Language

```python
import backend
import backend_numpy
import numpy

def dtype_name(obj):
  """Return name of the dtype"""
  return numpy.dtype(obj).name

class base(backend.base):
  def __init__(self, size, dtype, bhc_obj=None):
    super(base, self).__init__(size, dtype)
    if bhc_obj is None:
      f = eval("bhc.bh_multi_array_%s_new_empty"%dtype_name(dtype))
      bhc_obj = f(1, (size,))
    self.bhc_obj = bhc_obj

  def __del__(self):
    exec "bhc.bh_multi_array_%s_destroy(self.bhc_obj)"%dtype_name(self.dtype)

class view(backend.view):
  def __init__(self, ndim, start, shape, stride, base):
    super(view, self).__init__(ndim, start, shape, stride, base)
    dtype = dtype_name(self.dtype)
    exec "base = bhc.bh_multi_array_%s_get_base(base.bhc_obj)"%dtype
    f = eval("bhc.bh_multi_array_%s_new_from_view"%dtype)
    self.bhc_obj = f(base, ndim, start, shape, stride)

  def __del__(self):
    exec "bhc.bh_multi_array_%s_destroy(self.bhc_obj)"%dtype_name(self.dtype)

def get_data_pointer(ary, allocate=False, nullify=False):
  dtype = dtype_name(ary)
  ary = ary.bhc_obj
  exec "bhc.bh_multi_array_%s_sync(ary)"%dtype
  exec "bhc.bh_multi_array_%s_discard(ary)"%dtype
  exec "bhc.bh_runtime_flush()"
  exec "base = bhc.bh_multi_array_%s_get_base(ary)"%dtype
  exec "data = bhc.bh_multi_array_%s_get_base_data(base)"%dtype
  if data is None:
    if not allocate:
      return 0
    exec "data = bhc.bh_multi_array_%s_get_base_data_and_force_alloc(base)"%dtype
    if data is None:
      raise MemoryError()
  if nullify:
    exec "bhc.bh_multi_array_%s_nullify_base_data(base)"%dtype
  return int(data)

def set_bhc_data_from_ary(self, ary):
  return backend_numpy.set_bhc_data_from_ary(self, ary)

def ufunc(op, *args):
    args = [a.bhc_obj for a in args]
    in_dtype = dtype_name(args[1])
    f = eval("bhc.bh_multi_array_%s_%s"%(dtype_name(in_dtype), op.info['name']))
    exec f(*args)

def extmethod(name, out, in1, in2):
  f = eval("bhc.bh_multi_array_extmethod_%s_%s_%s"%(dtype_name(out), dtype_name(in1), dtype_name(in2)))
  ret = f(name, out, in1, in2)
  if ret != 0:
    raise NotImplementedError()
```

Fig. 8: A code snippet of the Bohrium backend. Note that the `backend` module refers to the implementation in figure 4 and note that the `backend_numpy` module is figure 5.

	Hardware Utilization	Matrix Multiplication Software
Native	1 CPU-core	ATLAS v3.10
NumPy	1 CPU-core	ATLAS v3.10
Numexpr	8 CPU-cores	ATLAS v3.10
libgpuarray	1 GPU	clBLAS v2.2
BohriumCPU	8 CPU-cores	$O(n^3)$
BohriumGPU	1 GPU	$O(n^3)$

TABLE II: The benchmark execution setup. Note that *Native* refers to a regular NumPy execution whereas *NumPy* refers to the backend implementation that makes use of the NumPy library.

regular NumPy execution, referred to as Native, and the four backend implementations: NumPy, Numexpr, libgpuarray, and Bohrium, referred to by their name.

We run all benchmarks, on an Intel Xeon machine with a dedicated Nvidia graphics card (Table I). Not all benchmark executions utilize the whole machine; Table II shows the specific setup of each benchmark execution. For each benchmark, we report the mean of ten execution runs and the error margin of two standard deviations from the mean. We use 64-bit double floating-point precision for all calculations and the size of the memory allocation pool (vcache) is 10 entries when applicable.

We use three Python applications that use either the NumPy module or the npbackend module. The source codes of the benchmarks are available at the Bohrium project's website[4]:

Heat Equation simulates the heat transfer on a surface represented by a two-dimensional grid, implemented using jacobi-iteration with numerical convergence (Fig. 2).

Shallow Water simulates a system governed by the Shallow Water equations. The simulation commences by placing a drop of water in a still container. The simulation then proceeds, in discrete time-steps, simulating the water movement. The implementation is a port of the MATLAB application by Burkardt[5].

Snakes and Ladders is a simple children's board game that is completely determined by dice rolls with no player choices. In this benchmark, we calculate the probability of ending the game after k-th iterations through successive matrix multiplications. The implementation is by Natalino Busa[6].

Heat Equation

Figure 9 shows the result of the Heat Equation benchmark where the Native NumPy execution provides the baseline. Even though the npbackend invertible introduces an overhead, the NumPy backend outperforms the Native NumPy execution, which is the result of the memory allocation reuse (vcache). The Numexpr achieves a 2.2 speedup compared to Native NumPy, which is disappointing since Numexpr utilizes all eight CPU-cores. The problem is twofold: we only provide one ufunc for Numexpr to JIT compile at a time, which hinders loop fusion, and secondly, since the problem is memory bound, the utilization of eight CPU-cores through OpenMP is limited.

[4]http://www.bh107.org
[5]http://people.sc.fsu.edu/˜jburkardt/m_src/shallow_water_2d/
[6]https://gist.github.com/natalinobusa/4633275

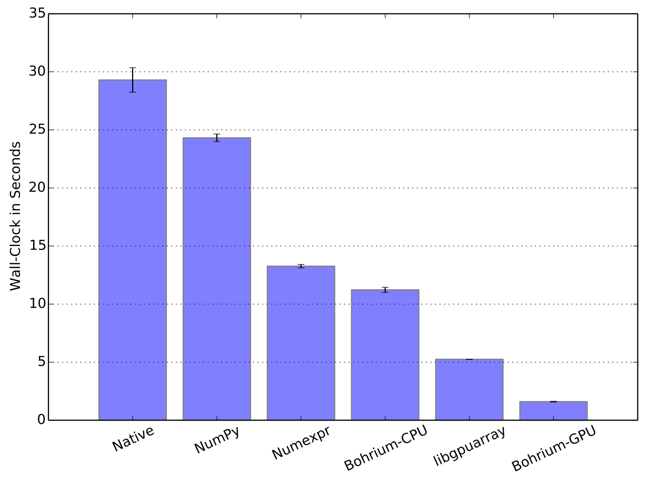


Fig. 9: The Heat Equation Benchmark where the domain size is 3000^2 and the number of iterations is 100.

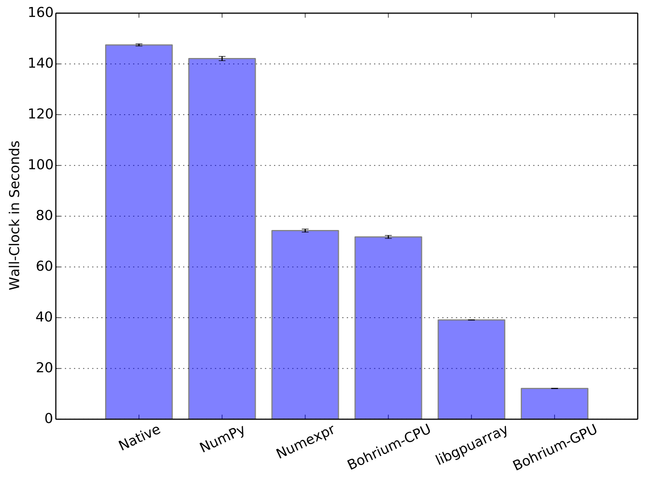


Fig. 10: The Shallow Water Benchmark where domain size is 2000^2 and the number of iterations is 100.

The Bohrium-CPU backend achieves a speedup of 2.6 while utilizing eight CPU-cores as well.

Finally, the two GPU backends, libgpuarray and Bohrium-GPU, achieve a speedup of 5.6 and 18 respectively. Bohrium-GPU performs better than libgpuarray primarily because of loop fusion and array contraction[28], which is possible since Bohrium-GPU uses lazy evaluation to fuse multiple ufunc operations into single kernels.

Shallow Water

Figure 10 shows the result of the Shallow Water benchmark. This time the Native Numpy execution and the NumPy backend perform the same, thus the vcache still hides the npbackend overhead. Again, Numexpr and Bohrium-CPU achieve a disappointing speedup of 2 compared to Native NumPy, which translate into a CPU utilization of 25%.

Finally, the two GPU backends, libgpuarray and Bohrium-GPU, achieve a speedup of 3.7 and 12 respectively. Again,

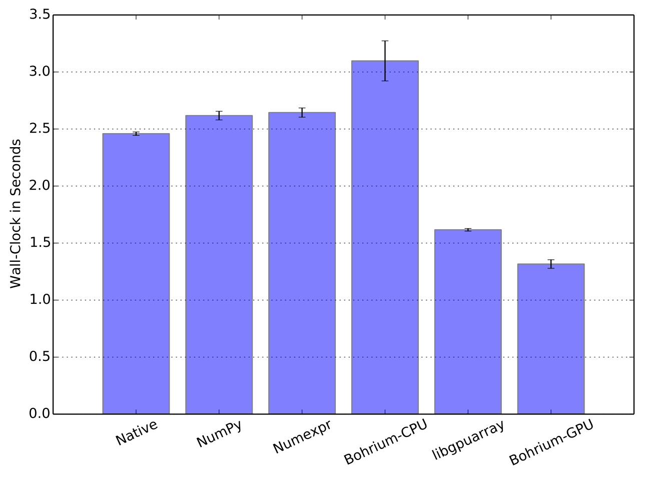


Fig. 11: The Snakes and Ladders Benchmark where the domain size is 1000^2 and the number of iterations is 10.

Bohrium-GPU outperforms libgpuarray because of loop fusion and array contraction.

Snakes and Ladders

Figure 11 shows the result of the Snakes and Ladders benchmark where the performance of matrix multiplication dominates the overall performance. This is apparent when examining the result of the three first executions, Native, NumPy, and Numexpr, that all make use of the matrix multiplication library ATLAS (Table II). The Native execution outperforms the NumPy and Numexpr executions with a speedup of 1.1, because of reduced overhead.

The performance of the Bohrium-CPU execution is significantly slower than the other CPU execution, which is due to the naïve $O(n^3)$ matrix multiplication algorithm and no clever cache optimizations.

Finally, the two GPU backends, libgpuarray and Bohrium-GPU, achieves a speedup of 1.5 and 1.9 respectively. It is a bit surprising that libgpuarray does not outperform Bohrium-GPU since it uses the clBLAS library but we conclude that the Bohrium-GPU with its loop fusion and array contraction matches clBLAS in this case.

Fallback Overhead: In order to explore the overhead of falling back to the native NumPy implementation, we execute the Snakes and Ladders benchmark where the backends do not support matrix multiplication. In order for the native NumPy to perform the matrix multiplication each time the application code uses matrix multiplication, npbackend will transfer the array data from the backend address space to the user address space and vice versa. However, since npbackend uses the `mremap()` function to transfer array data, the overhead is only around 14% (Fig. 12) for the CPU backends. The overhead of libgpuarray is 60% because of multiple memory copies when transferring to and from the GPU (Fig. 7 Line 13-18). Contrarily, the Bohrium-GPU backend only performs one copy when transferring to and from the GPU, which results in an overhead of 23%.

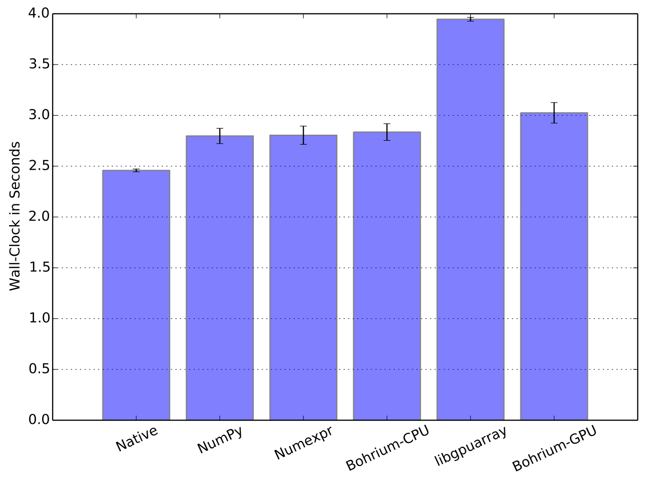


Fig. 12: The Snakes and Ladders Benchmark where the backends does not have matrix multiplication support. The domain size is 1000^2 and the number of iterations is 10.

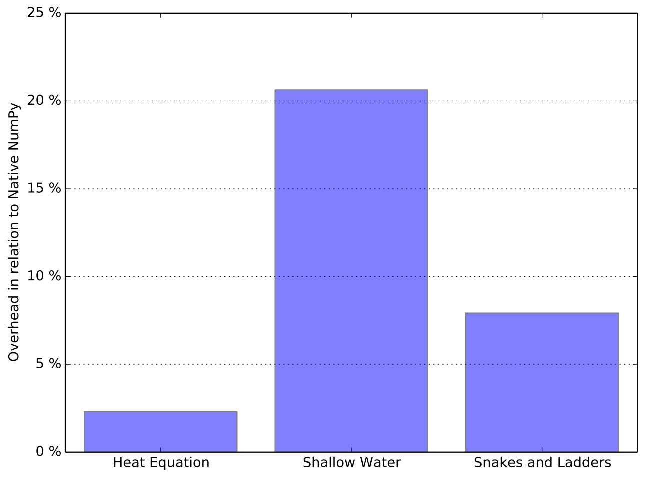


Fig. 13: Overhead of npbackend where we compare the NumPy backend with the native NumPy execution from the previous benchmarks.

Overhead

In the benchmarks above, the overhead of the npbackend is very modest and in the case of the Heat Equation and Shallow Water benchmarks, the overhead is completely hidden by the memory allocation pool (vcache). Thus, in order to measure the precise overhead, we deactivate the vcache and re-run the three benchmarks with the NumPy backend (Fig. 13). The ratio between the number of NumPy operations and the quantity of the operations dictates the npbackend overhead. Thus, the Heat Equation benchmark, which has a domain size of 3000^2, has a lower overhead than the Shallow Water benchmark, which has a domain size of 2000^2. The Snakes and Ladders benchmark has an even smaller domain size but since the matrix multiplication operation has a $O(n^3)$ time complexity, the overhead lies between the two other benchmarks.

VII. Future Work

An important improvement of the npbackend framework is to broaden the support of the NumPy API. Currently, npbackend supports array creation functions, matrix multiplication, random, FFT, and all ufuncs, thus many more functions remain unsupported. Even though we can leverage the work by the PyPy project, which re-implements a broad range of the NumPy API in Python[7], we still have to implement Extension Methods for the part of the API that is not expressed well using ufuncs.

Currently, npbackend supports CPython version 2.6 to 2.7; however there is no technical reason not to support version 3 and beyond thus we plan to support version 3 in the near future.

The implementation of the backend examples we present in this paper has a lot of optimization potential. The Numexpr and libgpuarray backends could use lazy evaluation in order to compile many ufunc operations into single execution kernels and gain similar performance results as the Bohrium CPU and GPU backends.

Current ongoing work explores the use of Chapel[29] as a backend for NumPy, providing transparent mapping (facilitated by npbackend), of NumPy array operations to Chapel array operations. Thereby, facilitating the parallel and distributed features of the Chapel language.

Finally, we want to explore other hardware accelerators, such as the Intel Xeon Phi Coprocessor, or distribute the calculations through MPI on a computation cluster.

VIII. Conclusion

In this paper, we have introduced a unified NumPy backend, npbackend, that unifies a broad range of Python code accelerators. Without any modifications to the original Python application, npbackend enables backend implementations to improve the Python execution performance. In order to assess this clam, we use three benchmarks and four different backend implementations along with a regular NumPy execution. The results show that the overhead of npbackend is between 2% and 21% but with a simple memory allocation reuse scheme it is possible to achieve overall performance improvements.

Further improvements are possible when using JIT compilation and utilizing multi-core CPUs, a Numexpr backend achieves 2.2 speedup and a Bohrium-CPU backend achieves 2.6 speedup. Even further improvement is possible when utilizing a dedicated GPU, a libgpuarray backend achieves 5.6 speedup and a Bohrium-GPU backend achieves 18 speedup. Thus, we conclude that it is possible to accelerate Python/NumPy application seamlessly using a range of different backend libraries.

References

[1] G. van Rossum, "Glue it all together with python," in *Workshop on Compositional Software Architectures, Workshop Report, Monterey, California*, 1998.

[2] T. E. Oliphant, *A Guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[7]http://buildbot.pypy.org/numpy-status/latest.html

[3] E. Jones, T. Oliphant, and P. Peterson, “Scipy: Open source scientific tools for python,” *http://www. scipy. org/*, 2001.

[4] J. D. Hunter, “Matplotlib: A 2d graphics environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[5] M. Sala, W. Spotz, and M. Heroux, “PyTrilinos: High-performance distributed-memory solvers for Python,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, March 2008.

[6] D. I. Ketcheson, K. T. Mandli, A. J. Ahmadia, A. Alghamdi, M. Quezada de Luna, M. Parsani, M. G. Knepley, and M. Emmett, “PyClaw: Accessible, Extensible, Scalable Tools for Wave Propagation Problems,” *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C210–C231, Nov. 2012.

[7] J. Enkovaaraa, M. Louhivuoria, P. Jovanovicb, V. Slavnicb, and M. Rännarc, “Optimizing gpaw,” *Partnership for Advanced Computing in Europe*, September 2012.

[8] D. Loveman, “High performance fortran,” *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 1, no. 1, pp. 25–42, 1993.

[9] B. Chamberlain, S.-E. Choi, C. Lewis, C. Lin, L. Snyder, and W. Weathersby, “Zpl: a machine independent programming language for parallel computers,” *Software Engineering, IEEE Transactions on*, vol. 26, no. 3, pp. 197–211, Mar 2000.

[10] W. Yang, W. Cao, T. Chung, and J. Morris, *Applied numerical methods using MATLAB*. Wiley-Interscience, 2005.

[11] C. Sanderson *et al.*, “Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments,” Technical report, NICTA, Tech. Rep., 2010.

[12] T. Veldhuizen, “Arrays in Blitz++,” in *Computing in Object-Oriented Parallel Environments*, ser. Lecture Notes in Computer Science, D. Caromel, R. Oldehoeft, and M. Tholburn, Eds. Springer Berlin Heidelberg, 1998, vol. 1505, pp. 223–230.

[13] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, “Cython: The best of both worlds,” *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[14] M. Foord and C. Muirhead, *IronPython in Action*. Greenwich, CT, USA: Manning Publications Co., 2009.

[15] S. Pedroni and N. Rappin, *Jython Essentials: Rapid Scripting in Java*, 1st ed. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2002.

[16] A. Rigo and S. Pedroni, “Pypy’s approach to virtual machine construction,” in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA ’06. New York, NY, USA: ACM, 2006, pp. 944–953. [Online]. Available: http://doi.acm.org/10.1145/1176617.1176753

[17] E. Jones and P. J. Miller, “Weaveinlining c/c++ in python.” OReilly Open Source Convention, 2002.

[18] D. Cooke and T. Hochberg, “Numexpr. fast evaluation of array expressions by using a vector-based virtual machine.”

[19] T. Oliphant, “Numba python bytecode to llvm translator,” in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2012.

[20] A. Klckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, “PyCUDA and PyOpenCL: A scripting-based approach to GPU runtime code generation,” *Parallel Computing*, vol. 38, no. 3, pp. 157 – 174, 2012.

[21] A. Munshi *et al.*, “The OpenCL Specification,” *Khronos OpenCL Working Group*, vol. 1, pp. 11–15, 2009.

[22] C. Nvidia, “Programming guide,” 2008.

[23] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, “Theano: a CPU and GPU math expression compiler,” in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Jun. 2010, oral Presentation.

[24] M. R. B. Kristensen and B. Vinter, “Numerical python for scalable architectures,” in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS ’10. New York, NY, USA: ACM, 2010, pp. 15:1–15:9.

[25] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter, “Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster,” in *4th Workshop on Python for High Performance and Scientific Computing (PyHPC’13)*, 2013.

[26] S. A. F. Lund, K. Skovhede, M. R. B. Kristensen, and B. Vinter, “Doubling the Performance of Python/NumPy with less than 100 SLOC,” in *4th Workshop on Python for High Performance and Scientific Computing (PyHPC’13)*, 2013.

[27] D. M. Beazley *et al.*, “Swig: An easy to use tool for integrating scripting languages with c and c++,” in *Proceedings of the 4th USENIX Tcl/Tk workshop*, 1996, pp. 129–139.

[28] V. Sarkar and G. R. Gao, “Optimization of array accesses by collective loop transformations,” in *Proceedings of the 5th International Conference on Supercomputing*, ser. ICS ’91. New York, NY, USA: ACM, 1991, pp. 194–205.

[29] D. Callahan, B. L. Chamberlain, and H. P. Zima, “The Cascade High Productivity Language,” in *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*. IEEE Computer Society, April 2004, pp. 52–60.

6.10 Prototyping for Exascale

Prototyping for Exascale

Brian Vinter
Niels Bohr Institute
University of Copenhagen
Denmark
vinter@nbi.ku.dk

Mads R. B. Kristensen
Niels Bohr Institute
University of Copenhagen
Denmark
madsbk@nbi.ku.dk

Simon A. F. Lund
Niels Bohr Institute
University of Copenhagen
Denmark
safl@nbi.ku.dk

Troels Blum
Niels Bohr Institute
University of Copenhagen
Denmark
blum@nbi.ku.dk

Kenneth Skovhede
Niels Bohr Institute
University of Copenhagen
Denmark
skovhede@nbi.ku.dk

ABSTRACT
Prototyping is a common approach to developing new scientific codes. Through prototypes a scientist builds confidence that an idea can in fact work for a scientific challenge, and the prototype also acts as the definitive design for a final implementation. As supercomputing approaches exaflops performance, the teraflops platforms that are available for prototyping becomes increasingly distant from the target performance, and new tools are needed to help close the performance gap between high productivity prototyping and high performance end-solutions. In this work we propose Bohrium, an open-source just-in-time compiler, as a possible solution to the prototyping problem. We will show how the same Python program can execute seamlessly on single-core, multi-cores, GPGPU and cluster architectures, and thus eliminating the need for parallel programming in the prototype stage. We will show how the same, unmodified, Python implementation of a Jacobi solver, a Black-Scholes pricing, an $O(n^2)$ complexity n-body simulation, and a Shallow-Water simulation scales to a 32-core machine with 50.1, 29.8, 17.3, and 44.4 speedups compared to the NumPy execution, while the same Python benchmarks run on a NVidia GTX 680, achieves speedups of 55.7, 43.0, 77.1, and 140.2, and a eight node cluster with gb-ethernet interconnect (256 cores in total), obtain speedups of 4.1, 7.9, 6.6 and 6.4, compared to a single 32 core node.

1. INTRODUCTION
While achieving exascale-computing in itself is a huge technical task, bringing scientific users to a competence level where they can utilize an exascale machine is likely to pose problems of the same scale. While large codes, maintained by a research community, is likely to make the transition from peta- to exascale as a natural evolution in the code, smaller teams will be hard pressed to make the move to exascale. One of the challenges that face researchers that write their own codes is that of prototyping. Today most teams will move from idea to code via a prototype, typically in Matlab, IDL, Python or a similar high productivity programming language.

Prototyping is an essential tool for testing the scientific hypothesis in small scale before spending more time on an actual implementation, since many scientific expressions do not easily translate into algorithms, and issues such as numerical stability, etc. are often not investigated formally. Thus the actual workflow for scientific codes is often iterative as shown in figure 1 below. Scientists will test their idea in a high productivity environment, using a small dataset, typically a ratio of one to a thousand, on a conventional, but large, computer, before moving on to an actual supercomputer for the real scale experiments.

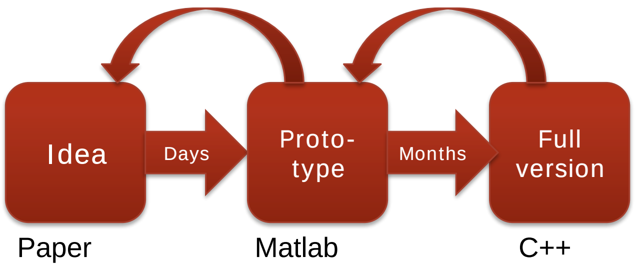


Figure 1: Prototyping workflow

With respect to exascale computing this approach poses a significant challenge, while exascale machines will be build by scaling supercomputers to a core count two orders of magnitude, in the order of 100 million, no such explosion in high-end servers is guaranteed, or even likely. Thus while researchers today, are expected to move three orders of magnitude, from teraflops to petaflops, when moving from prototype to final implementation, prototyping is likely to remain at teraflop when supercomputers move to exaflops, and researchers will have to move six orders of magnitude. Even scaling three orders of magnitude as is done today is non-trivial and often problems arise that were not detected by the prototype, when this challenge is increased another three orders of magnitude it is unlikely that the current approach will be sustainable.

Thus it makes sense to investigate new, scalable, approaches to prototyping. The successful prototyping tool must be highly productive and allow descriptive representations of an algorithm both of which are met by todays use of Matlab. A future prototyping tool must however be much faster than Matlab, it must have a better single core performance, and it must be able to run on multicores, accelerators, and cluster-computeres alike in order to satisfy the requirement of doing prototyping at the petaflops-scale in order to reduce the distance to state-of-the-art exaflops machines to three orders of magnitude, as it is today.

In the following, we introduce the Bohrium just-in-time compiler which, combined with the Numerical Python library NumPy[5], may provide a solution to next-generation-prototyping. Bohrium is not developed for prototyping, but rather for rapid solutions on parallel hardware, but non the less it matches the requirements that we believe are essential for prototyping in for the exascale.

2. THE BOHRIUM RUNTIME SYSTEM

The open-source project Bohrium[1] is a runtime system for high-performance high-productivity development[4, 3]. Bohrium provides the mechanics to couple an array-programming language or library with an architecture-specific implementation seamlessly.

Bohrium consists of a number of components that communicate by exchanging a hardware agnostic array bytecode. Components can be architecture-specific but they are all interchangeable since all uses the same bytecode and communication protocol. This design makes it possible to combine components in a setup that match a specific execution environment without changing the original user application. Bohrium consist of the following three component types (Fig. 2):

Bridge The role of the Bridge is to integrate Bohrium into existing languages and libraries. The Bridge generates array bytecode that corresponds to the user-code.

Vector Engine Manager (VEM) The role of the VEM is to manage data location and ownership of arrays. It also manages the distribution of computing jobs between potentially several Vector Engines and thus multiple processors.

Vector Engine (VE) The VE is the architecture-specific implementation that executes array bytecode.

When using the Bohrium framework, at least one implementation of each component type must be available. However, the exact component setup depends on the runtime system and what hardware to utilize, e.g. executing NumPy on a single machine using the CPU would require a Bridge implementation for NumPy, a VEM implementation for a machine node, and a VE implementation for a CPU. Now, in order to utilize a GPU instead, we can exchange the CPU-VE with a GPU-VE without having to change a single line of code in the NumPy application. This is a key feature of Bohrium: the ability to change the execution hardware without changing the user application.

2.1 Configuration

To make Bohrium as flexible a framework as possible, Bohrium manage the setup of all the components at runtime through a configuration file. The idea is that the user or system administrator can specify the hardware setup of the system through an configuration file (Fig. 3). Thus, it is just a matter of editing the configuration file when changing or moving to a new hardware setup and there is no need to change the user applications.

[1]Available at http://www.bh107.org.

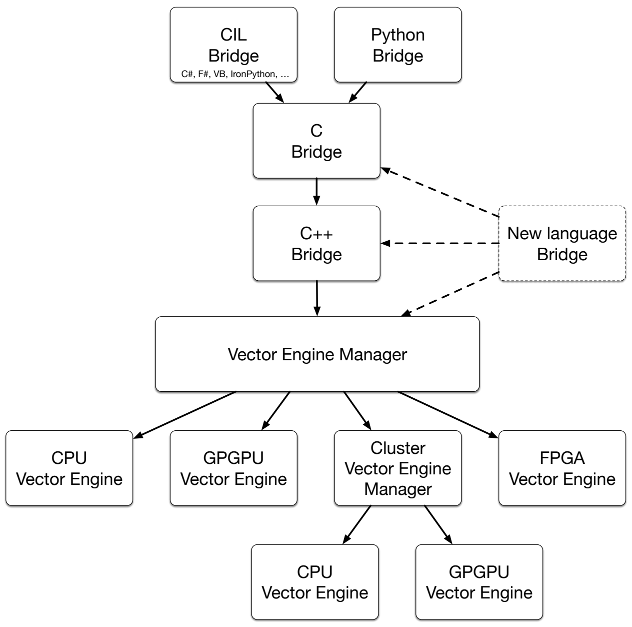


Figure 2: Component Overview

2.2 Vector Bytecode

A vital part of Bohrium is the array bytecode that constitutes the link between the high-level user language and the low-level execution engine. The bytecode is designed with the declarative array-programming model in mind where the bytecode instructions operate on input and output arrays. To avoid excessive memory copying, the arrays can also be shaped into multi-dimensional arrays. These reshaped array views are then not necessarily comprised of elements that are contiguous in memory. Each dimension comprises a stride and size, such that any regularly shaped subset of the underlying data can be accessed. We have chosen to focus on a simple, yet flexible, data structure that allows us to express any regularly distributed arrays. Figure 4 shows how the shape is implemented and how the data is projected.

The aim is to have an array bytecode that support data parallelism implicitly and thus makes it easy for the bridge to translate the user language into the bytecode efficiently. Additionally, the design enables the VE to exploit data parallelism through SIMD[2] and the VEM through SPMD[3].

2.3 Bridge

The Bridge component is the *bridge* between the programming interface. In order to interface with the frontend language, the language-specific bridge component translates array operations into Bohrium array bytecode lazily. That is, the bridge collects array operations until it encounter a language condition, in which case it sends the collected

[2]Single Instruction, Multiple Data
[3]Single Program, Multiple Data

```
# Bridge for NumPy
[numpy]
type = bridge
children = node

# Vector Engine Manager for a single machine
[node]
type = vem
impl = libbh_vem_node.so
children = gpu

# Vector Engine for a GPU
[gpu]
type = ve
impl = lbbh_ve_gpu.so
```

Figure 3: This example configuration provides a setup for utilizing a GPU on one machine by instructing the Vector Engine Manager to use the GPU Vector Engine implemented in the shared library `lbhvb_ve_gpu.so`.

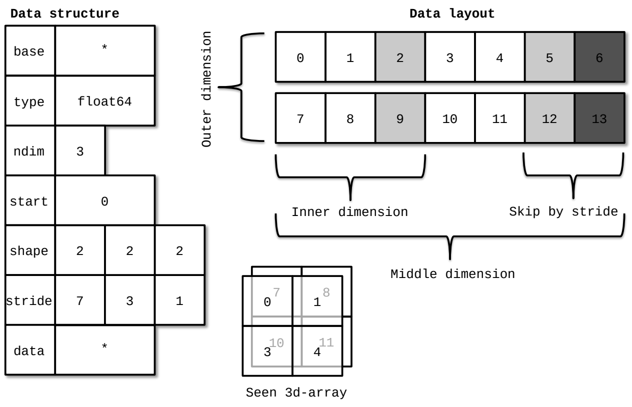


Figure 4: Descriptor for n-dimensional array and corresponding interpretation

```
import numpy as np

def solve(height, width, epsilon=0.005):
  grid = np.zeros((height+2,width+2),np.float64)
  grid [:,0]    = -273.15
  grid [:,-1]   = -273.15
  grid [-1,:]   = -273.15
  grid [0,:]    = 40.0
  center = grid[1:-1,1:-1]
  north  = grid [:-2,1:-1]
  south  = grid [2:,1:-1]
  east   = grid [1:-1,:-2]
  west   = grid [1:-1,2:]
  delta  = epsilon+1
  while delta > epsilon:
    tmp = 0.2*(center+north+south+east+west)
    delta = np.sum(np.abs(tmp-center))
    center[:] = tmp
```

Figure 5: Python implementation of a heat equation solve that uses the finite-difference method to calculate the heat diffusion. Note that in order to utilize Bohrium, we use the command line argument "-m", e.g. "`python -m npbackend heat2d.py`"

operations to the underlaying Bohrium components. Consequently, Bohrium only handles a subset of the frontend language – namely the array operations. The frontend language handles all non-deterministic aspect of program, such as conditional branches and loops, by itself.

An example of a Bohrium bridge is the Python/NumPy-bridge that seamlessly integrates with NumPy. The bridge is a drop-in replacement of NumPy thus without changing a single line of code, it is possible to utilize Bohrium (Fig. 5).

2.4 Vector Engine Manager

In order to utilize scalable architectures fully, distributed memory parallelism is mandatory. The Cluster component in Bohrium is currently quite naïve; it uses the bulk-synchronous parallel model[6] with static data decomposition and no communication latency hiding. We know from previous work than such optimizations are possible[2].

Bohrium implements all communication through the MPI-2 library and use a process hierarchy that consists of one master-process and multiple worker-processes. The master-process executes a regular Bohrium setup with the Bridge, Cluster-VEM, Node-VEM, and VE. The worker-processes, on the other hand, execute the same setup but without the Bridge and thus without the user applications. Instead, the master-process will broadcast array bytecode and array meta-data to the worker-processes throughout the execution of the user application.

Bohrium use a data-centric approach where a static decomposition dictates the data distribution between the MPI-processes. Because of this static data decomposition, all processes have full knowledge of the data distribution and need not exchange data location meta-data. Furthermore, the task of computing array operations is also statically distributed which means that any process can calculate locally what needs to be sent, received, and computed. Meta-data communication is only needed when broadcasting array bytecode and creating new arrays – a task that has an asymptotic complexity of $O(\log_2 n)$, where n is the number of nodes.

2.5 Vector Engine

The Vector Engine (VE) is the only component type that actually performs array operations. Bohrium implements two VEs, the CPU-VE and GPU-VE, that utilizes multi-core CPUs and GPGPUs respectively. Through the use of Just-In-Time (JIT) compilation, both VEs compiles the array bytecode received from the Node-VEM into architecture specific binary kernels. In order to utilize multi-core CPUs, the CPU-VE feeds the JIT-compiler with OpenMP annotated ANSI C source code whereas the GPU-VE generates OpenCL source code in order to utilize GPGPUs from both Nvidia and AMD.

3. BENCHMARKS

In order to evaluate the performance of Bohrium, we will perform a series of benchmarks that compares Bohrium against Python/NumPy. For each benchmark, we report the mean of five execution runs all within 10% deviation from the mean. We use 64-bit double floating-point precision for all calculations and all speedup results are strong scaling where the data size is fixed. The benchmarks consist of the following four applications:

Black Scholes The Black-Scholes pricing model is a partial differential equation, which is used in finance for calculating price variations over time[1]. This implementation uses a Monte Carlo simulation to calculate the Black-Scholes pricing model.

Heat Equation simulates the heat transfer on a surface represented by a two-dimensional grid, implemented using jacobi-iteration with numerical convergence (Fig. 5).

N-Body Nice The Nice variation of the newtonian n-body simulation is used to model larger galaxies with a large number of asteroids. The mass of the asteroids is small enough that their gravitational pull is insignificant. Thus, only the force of the planets are applied to the asteroids. The planets exchange forces similar to a regular n-body simulation.

Shallow Water simulates a system governed by the Shallow Water equations. The simulation commences by placing a drop of water in a still container. The simulation then proceeds, in discrete time-steps, simulating the water movement. The implementation is a port of the MATLAB application by Burkardt[4].

Multi-Core Processor

Figure 6 shows that results of running the four applications on 32 CPU-cores (Table 1). Beside comparing Bohrium versus Python/NumPy, the results of the Heat Equation includes two handwritten parallel implementations – one in ANSI-C and one in C++11 – both using OpenMP. The results clearly shows that the CPU-VE of Bohrium achieve a significant performance boost compared to Python/NumPy and is even competitive to handwritten compiled C/C++ code.

[4]http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_2d/

Processor:	AMD Opteron 6272
Clock:	2.1 GHz
Cores:	32
L3 Cache:	16MB
Memory:	128GB DDR3
Compiler:	GCC 4.6.3
Network:	Gigabit Ethernet
Software:	Linux 3.13, Python 2.7, NumPy 1.8.2

Table 1: Multi-Core Processor Specification

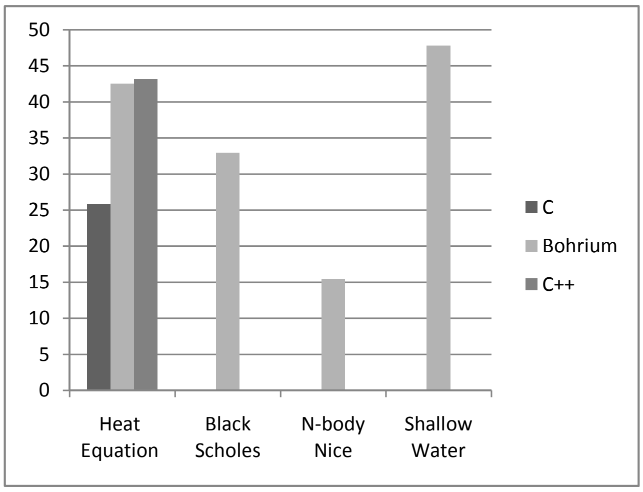


Figure 6: Relative speedup utilizing 32-cores compared to a sequential Python/NumPy execution.

GPGPU

Figure 7 shows that results of running the four applications on a GPGPU (Table 2). Compared to the multi-processor, the GPGPU takes the performance boost even further. Noticeable is the Black Scholes results with more than 300 times speedup compared to Python/NumPy.

Cluster

Figure 8 shows that results of running the four applications on an eight-node cluster where each node is the multi-code processor from Table 1 connected through Gigabit Ether-

Processor:	Intel Core i7-3770
Clock:	3.4 GHz
Cores:	4
L3 Cache:	16MB
Memory:	128GB DDR3
Compiler:	GCC 4.6.3
Network:	Gigabit Ethernet
GPGPU:	AMD HD 7970
Clock:	1000 MHz
Memory:	3GB GDDR5
-bandwidth:	288 GB/s
Software:	Linux 3.13, Python 2.7, NumPy 1.8.2, OpenCL 2.1

Table 2: GPGPU Specification

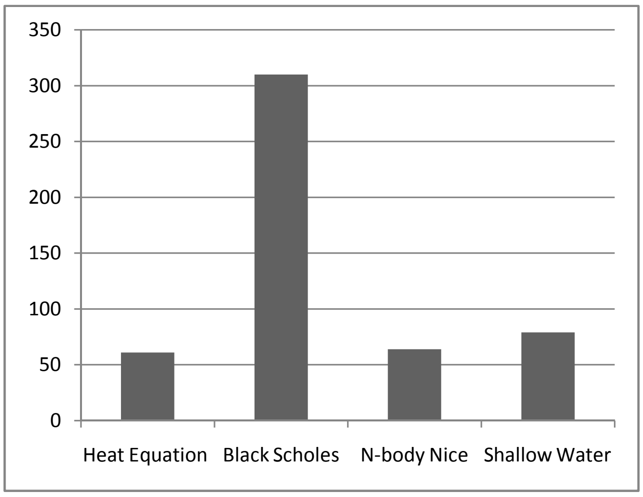


Figure 7: Relative speedup utilizing GPGPU compared to a sequential Python/NumPy execution.

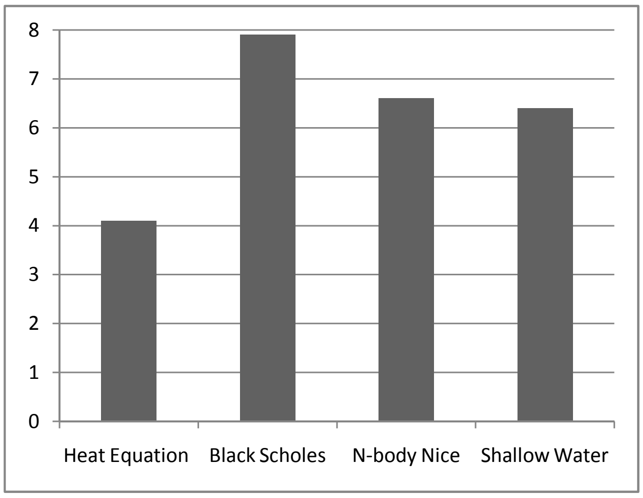


Figure 8: Relative scalability on an eight-node cluster. We compared the utilization of one node (32-cores) against the utilization of eight node (256-cores).

net. We run a MPI-process per cluster node and use the CPU-VE benchmark (here, we use an older version of the CPU-VE implementation than the one previously used) in Bohrium to utilize the 32-cores on each node. In order to show scalability, we compare 32-cores executions with 256-cores executions. The scalability goes from 50% to 95% speedup utilization.

4. CONCLUSIONS

In this work we have shown how Python/Numpy in combination with the Bohrium just-in-time compiler, offers an attractive work-to-performance ratio. While better performance can be had from expert implementations, a Python/Numpy implementation is fully on-par with other high-producitivity languages with respect to the the effort the scientist has to put in, and the performance is on-par, or close to, that of an straight forward C++ implementation. The end result is that a scientist may move seamlessly from a laptop version of a code to a large, heterogeneous, parallel machine, without any changes to the code. In fact, we will show that the scientist can continue to work from a laptop, with interactive graphics if needed, while the contracted array operations are all executed on a remote machine, including supercomputers, granted that the scientist is willing to wait online for the job to be scheduled at the SC site. The descriptive approach not only makes scientists more productive, but also reduces the number of errors as no explicit parallelism is expressed, and synchronization requirements are fully derived from the descriptive implementation of the algorithm.

5. REFERENCES

[1] F. Black and M. Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, pages 637–654, 1973.

[2] M. Kristensen and B. Vinter. Managing communication latency-hiding at runtime for parallel programming languages and libraries. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 546–555, 2012.

[3] M. R. Kristensen, S. A. Lund, T. Blum, K. Skovhede, and B. Vinter. Bohrium: a virtual machine approach to portable parallelism. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 312–321. IEEE, 2014.

[4] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter. Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster. In *4th Workshop on Python for High Performance and Scientific Computing (PyHPC'13)*, 2013.

[5] T. E. Oliphant. Python for Scientific Computing. *Computing in Science and Engg.*, 9(3):10–20, May 2007.

[6] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

6.11 Scripting Language Performance Through Interoperability

Scripting Language Performance Through Interoperability

Simon A. F. Lund
Niels Bohr Institute, University of Copenhagen
safl@nbi.dk

Bradford L. Chamberlain
Cray, Inc.
bradc@cray.com

Brian Vinter
Niels Bohr Institute, University of Copenhagen
vinter@nbi.dk

1. Introduction

Attractiveness of programming in scripting languages can arguably be attributed to language features, removal of responsibilities from the programmer, and a rich execution environment. Attractive language features include dynamic typing and type inference supporting generic and less verbose code. Dynamic, managed memory, and garbage collection removes the error-prone tasks of allocating, reallocating and freeing memory from the programmer. Interactive interpreters facilitates experimentation and rapid application development. The lack of features such as parallel language constructs gives the programmer straightforward sequential semantics without concern to the hazards related with parallel execution. Scripting languages are often labeled as high-level since they remove these responsibilities from the programmer, allowing for code to evolve around manipulating abstractions closer to the application domain.

The price for these conveniences is often paid with lowered hardware utilization since the programmer only expresses *what* is to be computed not *how* to compute it. With a lack of control and no means of obtaining it, it becomes the task of the interpreter to map high-level application code to hardware efficiently.

One approach is to rely on language interoperability to increase application performance. Using Python as an example, the CPython interpreter allows for interoperability with C/C++, either via language extensions or by providing access to libraries.

Python has seen widespread use and popularity using this approach, specifically the NumPy/SciPy/iPython software stack gives the programmer a rich interactive environment for scientific computing. This is achieved by maintaining high-level abstractions and enabling significant performance improvements by implementing the computationally demanding portions in C and providing access to them via high-level data structures and operations upon them.

This becomes more challenging (or important) as efficient utilization of hardware becomes increasingly complex with continuing developments in hardware architectures such as increasing core counts on CPUs with NUMA architectures, distinct address spaces in accelerators such as GPUs, MICs, FPGAs. Interpreters are rarely able to keep up with the developments in hardware, and the abstractions provided by scripting languages will fail to deliver the potential use of available hardware. At this point, the need arises to enable the programmer to peel off layers of the language abstractions and take control.

Current approaches for doing so in Python include writing Python extensions, using SWIG or ctypes. In addition low-level APIs such as pyCUDA, pyOpenCL, pyMIC, and MPI4PY can be used.

These approaches successfully provide the means of recovering the majority of the lost performance and have contributed to Python's popularity in HPC environments serving as a steering-language for scientific computing. Language interoperability is the driver for the realization of these tools and libraries.

However, they come at the price of sacrificing all of the conveniences described earlier. The following sections introduce a different approach to foreign-function interfaces and interface generation which leverages the advantages of interoperability without sacrificing the attractive qualities of the scripting language.

2. Approach

The general idea is to provide a less verbose foreign function interface (FFI) by using introspection on the function definition and function decoration to construct a function prototype which can either be implemented inline, from file, or mapped to a library.

In order to maintain the high-level abstractions, the foreign language must support them. C does not meet this requirement, however, Chapel provides an ideal target. It supports high-level operations on arrays and also supports peeling off layers of the abstraction allowing the programmer to take control if they have sufficient motivation for doing so.

```
def simulation(tsteps, spin, decay, velocity):
    """Simulation of some physical phenomenon on a dataset."""

    delta = velocity
    for _ in xrange(0, tsteps):
        delta = (spin * decay + velocity) / delta

    return np.add.reduce(delta)
```

Figure 1. Computationally expensive Python function.

```
@Chapel()
def simulation(tsteps=int, spin=np.ndarray, decay=np.ndarray, velocity=np.ndarray):
    """
    var delta = velocity;
    for timestep in 1..tsteps {
        delta = (spin * decay + velocity) / delta;
    }
    return +reduce(delta);
    """
    return float
```

Figure 2. Python function rewritten in Chapel using an inlined foreign function body.

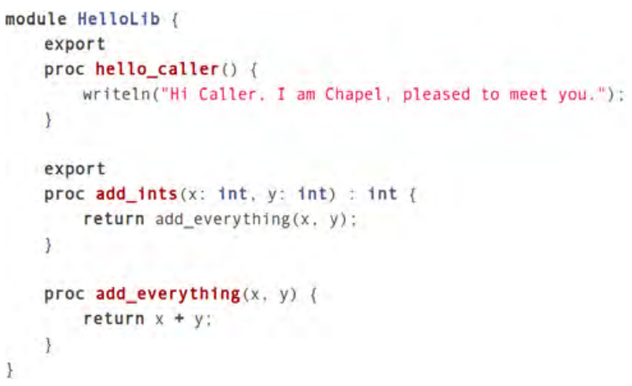
PPOPP '14, Month 0–0, 2014, San Francisco, California, United States of America.
Copyright © 2014 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

A prototype implementation named *pyChapel*[1] serves to illustrate the approach. pyChapel consists of a foreign function interface and a module compiler / interface generator.

The idea is that a computationally expensive function such as the one in figure 1 can be rewritten in a foreign language or mapped to a library.

The raw foreign function body can be provided inline, from file or by mapping it to an existing library. When using pyChapel in this manner, Chapel code will be dynamically generated and compiled at runtime. Machinery within pyChapel lowers compilation overhead by re-using previously compiled code.

PyChapel also provides a module-compiler, compiling Chapel modules into Python modules as illustrated in figure 3.

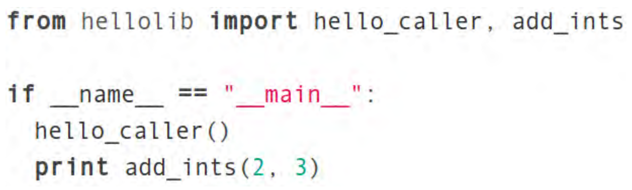


```
module HelloLib {
    export
    proc hello_caller() {
        writeln("Hi Caller, I am Chapel, pleased to meet you.");
    }

    export
    proc add_ints(x: int, y: int) : int {
        return add_everything(x, y);
    }

    proc add_everything(x, y) {
        return x + y;
    }
}
```

```
# pych --compile hellolib.chpl
```

Figure 3. Source code of the Chapel *HelloLib* module and command to compile it into a Python module.

The module can then be accessed from Python as illustrated in figure 4.



```
from hellolib import hello_caller, add_ints

if __name__ == "__main__":
  hello_caller()
  print add_ints(2, 3)
```

Figure 4. Accessing Chapel module *HelloLib* from Python.

When using the pyChapel module compiler, code will be generated and compiled prior to runtime.

3. Results

Two synthetic Python applications, hereafter referred to as *finance* and *scicomp*, were used to get an initial impression of the performance gained by mapping the computationally expensive portion of the application code to Chapel. The code can inspected in the pyChapel online documentation[2]. Two versions of each application were implemented, a reference implementation and a modification mapping the expensive functions (`simulation` and `quant`) to Chapel using techniques illustrated in figure 1 and figure 2. The rewrites were implemented without any explicit parallelism in order to maintain the abstraction-level of the code.

[1] http://pychapel.rtfd.org

[2] http://pychapel.readthedocs.org/usage_examples.html#accelerate-your-numpy-code

The machine executing was a laptop with 6GB of memory with an Intel i5-2410M CPU @ 2.3Ghz CPU with two physical cores. *finance* ran in $26.7s$ without pyChapel and $4.2s$ with. *scicomp* ran in $28.3s$ without pyChapel and $8.8s$ with. Resulting in speedups of 6.3× and 3.2×, respectfully.

A third implementation of *scicomp* was written, replacing the `@Chapel` decorator with `@FromC`. Mapping the Python function `simulation` to a C implementation. Running the C-targeted implementation showed only a marginal decrease in wall-clock time compared to the Chapel-targeted implementation.

4. Future Work

The pyChapel module was created as a means of providing interoperability between Python and Chapel. However, it can with little effort be expanded to support any language capable of interoperating with C including but not limited to Fortran and Haskell and thereby providing a generic and simplified approach to FFIs in Python.

The dynamic compilation machinery in pyChapel opens up exploration within staged computation[1]. This can be done by manipulating inlined foreign function bodies at runtime as a means of generating optimized code based on run-time values of inputs, such as specializing input-sensitive BLAS and FFT routines.

Current work focuses on maintaining the attractiveness of Python by implementing pyChapel as a target for the npbackend[2] module. The array operations of Python/NumPy can thereby be mapped transparently to Chapel. Thereby, effectively increasing the performance of the Python/NumPy program without changing a single line of code.

5. Conclusion

The experimental prototype pyChapel proposes a new approach to foreign function interfaces and interface generation for Python. The primary target for the prototype is Chapel, however, C is currently also supported, and further work can with little effort expand support for any language capable of interoperating with C. Chapel is the primary target as it maintains the attractive qualities of the scripting languages such as high-level array operations for scientific computing. Initial results indicate that the performance improvement of targeting Chapel is equivalent to that of targeting C.

Acknowledgments

The Chapel Team at Cray, Inc. provided valuable insights during the development of PyChapel especially Elliot Ronaghan, Ben Harshbarger, Thomas Van Doren, and Lydia Duncan.

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (hiperfit.dk) under contract number 10-092299.

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the 'High Performance High Productivity' project under contract number 09-067060.

References

[1] O. Kiselyov, C.-c. Shan, and Y. Kameyama, "Bridging the theory of staged programming languages and the practice of highperformance computing," Tech. Rep., 2012.

[2] M. R. B. Kristensen, S. A. F. Lund, T. Blum, and K. Skovhede, *Separating NumPy API from Implementation*, 2014.

6.12 Fusion of Array Operations at Runtime

Fusion of Array Operations at Runtime

Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, and James Avery
Niels Bohr Institute, University of Copenhagen, Denmark
{madsbk/safl/blum/avery}@nbi.dk

***Abstract*—In this paper, we address the problem of fusing array operations based on criteria, such as compatibility, data communication, and/or reusability. We formulate the problem as a graph partition problem that is general enough to handle loop fusion, combinator fusion, or other fusion of subroutines.**

I. Introduction

Array operation fusion is a program transformation which combines, or fuses, multiple array operations into a single *kernel* of operations. When it is applicable, the technique can drastically improve cache utilization through temporal data locality and enables other program transformations such as streaming and array contraction[1]. In scalar programming languages, such as C, array operation fusion typically corresponds to loop fusion where individual computation loops are combined into single loops. The effect is a reduction of arrays traversals (Fig. 1). Similarly, in functional programming languages it typically corresponds to the fusion of individual combinators. In array programming languages, such as HPF[2] and ZPL[3], the fusion of array operations is mandatory since a user application in these languages will consist of array operations almost exclusively.

However, the fusion of two array operations is not always applicable. Consider the two for-loops in Fig. 2; since the second loop traverse the result from the first loop reversely, we have to compute the complete result of the first loop before continuing to the second loop thus fusion is not directly applicable. With clever analytics, it might be possible to transform the program to a form where fusion is applicable however in this paper we presume that such optimizations have already been done.

We generalize this problem to: *Given a mixed graph, find a legal partition of vertices that cuts all non-directed edges and minimizes the cost of the partition.*[1]. Hereafter, we refer to this problem as the *Weighted Subroutine Partition Problem.*

We develop different algorithms to solve this problem and evaluate both their theoretical and practical performance compared to an optimal solution. In order to maximize data locality, we use them to fuse array operations within the Bohrium project[4] thus evaluating the algorithms in practice. All of Bohrium including the work of this paper is open source and available at www.bh107.org.

II. The Weighted Subroutine Partition Problem

The Weighted Subroutine Partition (WSP) problem is an extension of the *The Weighted Loop Fusion Problem*[5] where we include the weight function in the problem formulation. In this section, we will formally define the WSP problem and show that it is NP-hard.

[1]See Sec. II for the definition of a legal partition and its cost.

```
#define N 1000
double A[N], B[N], T[N];
// Array expression: A += B*A
for(int i=0; i<N; ++i)
  T[i] = B[i] * A[i];
for(int i=0; i<N; ++i)
  A[i] += T[i];
```

(a) Two individaul for-loops.

```
for(int i=0; i<N; ++i){
  T[i] = B[i] * A[i];
  A[i] += T[i];
}
```

(b) Loop fusion: the two for-loops fused into one.

```
for(int i=0; i<N; ++i){
  double t = B[i] * A[i];
  A[i] += t;
}
```

(c) Array contraction: the temporary array T is contracted into the scalar t.

Fig. 1. Loop fusion and array contraction in C.

```
#define N 1000
double A[N], B[N], T[N];
int j = N;
// Array expression: A += reverse(B * A)
for(int i=0; i<N; ++i)
  T[i] = B[i] * A[i];
for(int i=0; i<N; ++i)
  A[i] += T[--j];
```

Fig. 2. Two for-loops that cannot be fused easily because of inter-iteration dependencies.

Definition 1. *A WSP graph, $G = (V, E_d, E_f)$, is a mixed graph where (V, E_d) forms a directed graph and E_f is undirected edges between vertices in V.*

Definition 2. *The vertices in a WSP graph, $G = (V, E_d, E_f)$, have a strict partial order imposed by the directed edges in E_d such that if there exist a path from $v_1 \in V$ to $v_2 \in V$ then $v_1 < v_2$. Since the order is strict, the directed part of the graph, (V, E_d), is also acyclic.*

Definition 3. *Let a partition, P, of a WSP graph, $G = (V, E_d, E_f)$, denote a partitioning of the vertices, V, into k blocks, $P = \{B_1, B_2, ..., B_k\}$. Let Π_V denotes the set of possible partitions of V and let $P, P' \in \Pi_V$ have a partial order, $P \leq P'$, defined as $\forall B \in P, \exists B' \in P' : B \subseteq B'$.*

Definition 4. *Given a WSP graph, $G = (V, E_d, E_f)$, a partition, $P \in \Pi_V$, is said to be legal when for each block, $B \in P$, the following holds:*

1) $\nexists v_1, v_2 \in B : (v_1, v_2) \in E_f$. *(I.e. no block contains both endpoints of a fuse-preventing edge)*
2) *If* $v_1 < v_2 < v_3$ *and* $v_1, v_3 \in B$ *then* $v_2 \in B$. *(I.e. the directed edges between blocks must not form cycles)*

Definition 5. *Given a partition, P, of vertices in a WSP graph, a cost function cost(P) returns the cost of the partition and respects the following conditions:*

1) $cost(P) \geq 0$
2) $P \leq P' \Rightarrow cost(P) \geq cost(P')$

Definition 6. *Given a WSP graph,* $G = (V, E_d, E_f)$*, and a cost function, cost(P), the WSP problem is the problem of finding a legal partition, P, of V with minimal cost:*

$$P \in \operatorname*{argmin}_{P' \in \hat{\Pi}_V} cost(P') \tag{1}$$

where $\hat{\Pi}_V$ *denotes the set of legal partitions of V.*

A. Complexity

In order to proof that the WSP problem is NP-hard, we perform a reduction from the *Multiway Cut Problem*[6], which Dahlhaus et al. shows is NP-hard for all fixed $k \geq 3$.

Definition 7. *The Multiway Cut (MWC) problem can be defined as follows: An instance of the MWC problem,* $\mu = (V, E, S, w)$*, consist of a graph (V, E), a set* $S = \{s_1, s_2, ..., s_k\}$ *of k specified vertices or terminals, and a non-negative weight* $w(u, v)$ *for each edge* $(u, v) \in E$*. A partition,* $P = \{B_1, B_2, ..., B_k\}$*, of V associate a cost:*

$$cost_{MWC}(\mu, P) := \sum_{\substack{B, B' \in P' \\ B \neq B'}} \sum_{\substack{u \in B \\ v \in B' \\ (u,v) \in E}} w(u, v) \tag{2}$$

and is said to be legal when each $B \in P$ *contains no more than one terminal.*

Given an instance of the MWC problem, $\mu = (V, E, S, w)$*, the set of solutions is given by:*

$$\operatorname*{argmin}_{P \in \hat{\Pi}_V} cost_{MWC}(\mu, P) \tag{3}$$

where $\hat{\Pi}_V$ *denote the set of possible legal partitions of the vertices in V.*

Theorem 1. *The WSP problem is NP-hard for a graph,* $G = (V, E_d, E_f)$*, with a chain of* $k \geq 3$ *edges in* E_f*.*

Proof. We prove NP-hardness through a reduction from the MWC problem.

Given an instance of the MWC problem, $\mu = (V, E, S, w)$, we build an instance of the WSP problem as follows. We form the graph $G = (V, E_d, E_f)$ where $V = V'$, $E_d = \emptyset$, and $E_f = \{(s_i, s_j) : 1 \leq i < j \leq k\}$. We set the cost function to: $cost(P) = cost_{\mathrm{MWC}}(\mu, P)$.

We then have that $\hat{\Pi}_V = \hat{\Pi}_{V'}$, where $\hat{\Pi}_V$ and $\hat{\Pi}_{V'}$ is the set of legal partitions of the vertices V and V' respectively, because:

- The fuse-preventing edges in E_f is exactly between each terminal in S thus in both partition sets, multiple terminals are never in the same block (required by Def. 7 and the first condition of Def. 4).
- The set of directed edges in E_d is empty, which makes Def. 2 and the second condition of Def. 4 always true.

The cost function, $cost(P)$, is a legal WSP cost function because it respects the conditions of Def. 5:

- Since $w(u, v)$ is non-negative for all $(u, v) \in E'$, $cost(P)$ is non-negative for all $P \in \hat{\Pi}_V$.
- When $P, P' \in \hat{\Pi}_V$ and $P < P'$, it means that some blocks in P have been merged into shared blocks in P', which reduces the cost, but they are otherwise identical thus $cost(P) > cost(P')$.

Finally, since $\hat{\Pi}_V = \hat{\Pi}_{V'}$ and the set of solutions to the MWC and WSP instance is identical (Eq. 3), we hereby conclude the proof. □

III. Concretization of the WSP problem

Since the WSP problem formulation is very general, we will express a concrete optimization problem as a WSP problem thus demonstrates its real world use. The concrete problem is an optimization phase within the Bohrium runtime system where Bohrium partitions a set of array operations for fusion – the *Fusion of Array Operations* (FAO) problem:

Definition 8. *Given set of array operations equipped with a strict partial order imposed by the data dependencies between them,* $(A, <)$*, find a partition, P, of A where:*

1) *All array operations within a block in P are fusible (Def. 9)*
2) *For all blocks,* $B \in P$*, if* $v_1 < v_2 < v_3$ *and* $v_1, v_3 \in B$ *then* $v_2 \in B$*. (I.e. the directed edges between blocks must not form cycles).*
3) *The cost of the partition (Def. 10) is minimized.*

In the following, we will provide a description of Bohrium and show that a solution to the WSP problem is a solution to the FAO problem (Theorem 2).

A. Fusion of Array Operations in Bohrium

Bohrium is a computation backend for array programming languages and libraries that supports a range of languages, such as Python, C++, and .NET, and a range of computer architectures, such as CPU, GPU, and clusters thereof. The idea is to decouple the domain specific frontend implementation with the computation specific backend implementation in order to provide a high-productivity and high-performance framework.

Similar to NumPy, a Bohrium array operation operates on a set of inputs and produces a set of outputs[4]. Both input and output operands are *views* of arrays. An array view is a structured way to observe the whole or parts of an underlying *base* array. A base array is always a contiguous one-dimensional array whereas views can have any shape, stride, and dimensionality[4]. *Hereafter when we refer to an array, we mean an array view; when we refer to identical arrays, we mean identical array views that points to the same*

```
 1 import bohrium as bh
 2
 3 def synthetic():
 4   A = bh.zeros(4)
 5   B = bh.zeros(4)
 6   D = bh.zeros(5)
 7   E = bh.zeros(5)
 8   A += D[:-1]
 9   A[:] = D[:-1]
10   B += E[:-1]
11   B[:] = E[:-1]
12   T = A * B
13   bh.maximum(T, E[1:], out=D[1:])
14   bh.minimum(T, D[1:], out=E[1:])
15   return D
16 print synthetic()
```

(a)

```
 1 COPY A, 0
 2 COPY B, 0
 3 COPY D, 0
 4 COPY E, 0
 5 ADD A, A, D[:-1]
 6 COPY A, D[:-1]
 7 ADD B, B, E[:-1]
 8 COPY B, E[:-1]
 9 MUL T, A, B
10 MAX D[1:], T, E[1:]
11 MIN E[1:], T, D[1:]
12 DEL A
13 DEL B
14 DEL E
15 DEL T
16 SYNC D
17 DEL D
```

(b)

Fig. 3. A Python application that utilizes the Bohrium runtime system. In order to demonstrate various challenges and trade-offs, the application is very synthetic. Fig. (a) shows the Python code and Fig. (b) shows the corresponding Bohrium array bytecode.

base array; and when we refer to overlapping arrays, we mean array views that points to some of the same elements in a common base array.

Fig. 3a is a Python application that imports and uses Bohrium as a drop-in replacement of NumPy. The application allocates and initiates four arrays (line 4-7), manipulates those array through array operations (line 8-14), and prints the content of one of the arrays (line 16).

In order to be language agnostic, Bohrium translates the Python array operations into array bytecode (Fig. 3b) that the Bohrium backend can execute[2]. In the case of Python, the Python array operations and the Bohrium array bytecode is almost a one-to-one mapping where the first bytecode operand is the output array and the following operands are either input arrays or input literals. Since there is no scope in the bytecode, Bohrium uses `DEL` to destroy arrays and `SYNC` to move array data into the address space of the frontend language – in this case triggered by the Python `print` statement (Fig. 3a, line 16). There is no explicit bytecode for constructing arrays; on first encounter, Bohrium constructs them implicitly. Hereafter, we use the term *array bytecode* and *array operation* interchangeable.

In the next phase, Bohrium partitions the list of array operations into blocks that consists of fusible array operations – the FAO problem. As long as the preceding constraints between the array operations are preserved, Bohrium is free to reorder them as it sees fit thus optimizations based on data locality, array contraction, and streaming are possible.

In the final phase, the hardware specific backend implementation JIT-compiles each block of array operations and execute them.

1) Fusibility: In order to utilize data-parallelism, Bohrium and most other array programming languages and libraries impose a *data-parallelism* property on some or all array operations. The property ensures that the runtime system can calculate each output element independently without any communication between threads or processors. In Bohrium, all array operation must have this property.

[2]For a detailed description of this Python-to-bytecode translation we refer to previous work [7], [8].

Definition 9. *An array operation, f, in Bohrium has the data-parallelism property where each output element can be calculated independently, which imposes the following restrictions to its input f_{in} and output f_{out}:*

$$\forall i \in f_{in}, \forall o, o' \in f_{out} : i \cap o = \emptyset \vee o = o' \wedge o \cap o' = \emptyset \vee o = o' \tag{4}$$

In other words, if an input and an output or two output arrays overlaps, they must be identical. This does not apply to `DEL` and `SYNC` since they do not do any actual computation.

Consequently, array operation fusion must preserve the data-parallelism property:

Corollary 1. *In Bohrium, two array operations, f and f', are said to be fusible when the following holds:*

$$\begin{gathered}\forall i' \in f'_{in}, \forall o \in f_{out} : i' \cap o = \emptyset \vee i' = o \\ \wedge \\ \forall o' \in f'_{out}, \forall o \in f_{out} : o' \cap o = \emptyset \vee o' = o \\ \wedge \\ \forall o' \in f'_{out}, \forall i \in f_{in} : o' \cap i = \emptyset \vee o' = i\end{gathered}$$

Proof. It follows directly from Definition 9. □

2) Cost Model: In order to have an optimization object, Bohrium uses a generic cost function that quantify unique memory accesses and thus rewards optimizations such as array contraction, data reuse, and operation streaming. For simplicity, Bohrium will not differentiate between reads and writes and will not count access to literals and register variables, such accesses adds no cost:

Definition 10. *In bohrium, the cost of a partition, $P = \{B_1, B_2, ..., B_k\}$, of array operations is given by:*

$$0 \leq cost(P) = \sum_{B \in P} length\left(\bigcup_{f \in B} (f_{in} \cup f_{out})\right) \tag{6}$$

where $length(A)$ returns the total number of bytes accessed by the unique arrays in A. Note that the cost of `DEL` and `SYNC` is always zero.

Bohrium implements two techniques to improve data locality through array operation fusion:

Array Contraction When an array is created and destroyed within a single partition block, Bohrium will *contract* the array into temporary register variables, which typically corresponds to a scalar variable per parallel computing thread. Consider the program transformation between Fig. 1b and 1c where the temporary array T is array contracted into the scalar variable t. In this case, the transformation reduces the accessed data and memory requirement with `(N-1) * sizeof(double)` bytes.

Data Access Reuse When a partition block accesses an array multiple times, Bohrium will only read and/or write to that array once thus saving access to main memory. Consider the two for-loops in Fig. 1a that includes two traversals of `A` and `T`. In this case, it is possible to save one traversal of `A` and one traversal of `T` through fusion (Fig. 1b). Furthermore, the compiler can reduce the access to the main memory with $(N-1)^2$ since it can keep the current element of `A` and `T` in register.

Corollary 2. *In Bohrium, the cost saving of fusing two array operations, f and f', (in that order) with inputs, f_{in} and f'_{in}, outputs, f_{out} and f'_{out}, and the arrays to be destroyed, f_{del} and f'_{del}, is given by:*

$$0 \leq saving(f, f') = length(f_{out} \cap f'_{del}) + length(f_{out} \cap f'_{in})$$

Proof. The cost saving of fusing f and f' follows directly from the definition of the two optimizations Bohrium will apply when able: *Array Contraction* and *Data Access Reuse*. Array Contraction is applicable when an output of f is destroyed in f', which saves us $length(f_{out} \cap f'_{del})$ bytes. Meanwhile, Data Access Reuse is applicable when an output of f is also an input of f', which saves us $length(f_{out} \cap f'_{in})$ bytes. Thus, we have a total saving of $length(f_{out} \cap f'_{del}) + length(f_{out} \cap f'_{in})$. □

3) Fusion of Array Operations: Finally, we will show that algorithms that find solutions to the WSP problem also find solutions to the FAO problem.

Lemma 1. *In Bohrium, the cost of a partition of array operations is non-negative and monotonically decreasing on fusion thus satisfies the WSP requirement (Def. 5).*

Proof. The cost is clearly non-negative since the amount of bytes accessed by an array is non-negative. The cost is also monotonically decreasing when fusing two array operations since no new arrays are introduced; rather, the cost is based on the union of the arrays the two array operations accesses (Eq. 6). □

Theorem 2. *A solution to the WSP problem is a solution to the FAO problem.*

Proof. Given an instance of the FAO problem, $(A, <)$, we build an instance of the WSP problem as follows. We form the graph $G = (V, E_d, E_f)$ where:

1) For each array operation $a \in A$, we have a unique vertex $v \in V$ such that v represents a.
2) For each pair of array operations $a, a' \in A$, if $a < a'$ then there is an edge $(a', a) \in E_d$.
3) For each pair of array operations $a, a' \in A$, if a and a' is non-fusible then there is an edge $(a, a') \in E_f$.

We set the WSP cost function to the partition cost in Bohrium (Def. 2).

Through the same logical steps as in Theorem 1, it is easy to see that the set of legal partition of A equals the set of legal partitions of V. Additionally, Lemma 1 shows that the partition cost in Bohrium is a legal WSP cost function (Def. 5), which hereby concludes the proof. □

IV. Finding a Solution

In this section, we will present a range of WSP partition algorithms – from a greedy to an optimal solution algorithm. We use the Python application in Fig. 3 to demonstrate the results of each partition algorithm.

In order to unify the WSP problem into one data structure, Bohrium represents the WSP problem as a partition graph:

Definition 11. *Given an instance of the WSP problem – a graph, $G = (V, E_d, E_f)$, a partition, $P = \{B_1, B_2, ..., B_k\}$ of V, and a partition cost, $cost(P)$ – the partition graph representation is given as $\hat{G} = (\hat{V}, \hat{E}_d, \hat{E}_f, \hat{E}_w)$ where:*

- *Vertices, $\hat{V}$, represents partition blocks thus $\hat{V} = P$.*
- *Directed edges, $\hat{E}_d$, represents dependencies between blocks thus if $(B_1, B_2) \in \hat{E}_d$ then there exist an edge $(u, v) \in E_d$ where $u \in B_1$ and $v \in B_2$.*
- *Fuse-preventing edges, $\hat{E}_f$, represents non-fusibility between blocks thus if $(B_1, B_2) \in \hat{E}_f$ then there exist an edge $(u, v) \in E_f$ where $u \in B_1$ and $v \in B_2$.*
- *Weighted cost-saving edges, $\hat{E}_w$, represent the reduction in the partition cost if blocks are fused thus there is an edge between all fusible blocks and the weight of an edge $(B_1, B_2) \in E_w$ is $cost(P_1) - cost(P_2)$ where $B_1, B_2 \in P_1$ and $(B_1 \cup B_2) \in P_2$, which in Bohrium corresponds to $saving(B_1, B_2)$ (Corollary 2).*

Additionally, we need to define some notation:

- *Given a partition graph, $\hat{G}$, the notation $V[\hat{G}]$ denotes the set of vertices in $\hat{G}$, $E_d[\hat{G}]$ denotes the set of dependency edges, $E_f[\hat{G}]$ denotes the set of fuse-preventing edges, and $E_w[\hat{G}]$ denotes the set of weight edges.*
- *Given a partition graph, $\hat{G}$, let each vertex $v \in V[\hat{G}]$ associate a set of vertices, $\theta[v]$, where each vertex $u \in \theta[v]$ cannot fuse with v either directly because u, v are connected with a fuse-preventing edge or indirectly because there exist a path from u to v in $E_d[\hat{E}]$ that contains vertices connected with a fuse-preventing edge.*

Thus we have that a vertex in a partition graph $\hat{v} \in V[\hat{G}]$ is a set of vertices in the WSP problem. In order to remember this relationship, we mark the vertices in a partition graph with the $\hat{}$ notation.

A. Initially: No Fusion

Initially, Bohrium transforms the list of array operations into a WSP instance as described in Theorem 2 and then represents the WSP instance as a partition graph (Def. 11) where each block in the partition is assigned exactly one array operation and the weight of the cost-saving edges is derived by Corollary 2. We call this the *non-fused partition graph.*

The complexity of this transformation is $O(V^2)$ since we might have to check all pairs of array operations for dependecies, fusibility, and cost-saving, all of which is $O(1)$. Fig. 5 shows a partition graph of the Python example where all blocks have one array operation. The cost of the partition is 86.

B. *Sequences of Vertex Fusions*

We will now show that it is possible to build an optimal partition graph through a sequences of weighted edge fusions (i.e. edge contractions). For this, we need to define vertex fusion in the context of a partition graph:

Definition 12. *Given a partition graph, $\hat{G} = (\hat{V}, \hat{E}_d, \hat{E}_f, \hat{E}_w)$, and two vertices, $\hat{u}, \hat{v} \in \hat{V}$, the function* FUSE$(\hat{G}, \hat{u}, \hat{v})$ *returns a new partition graph where $\hat{u}, \hat{v}$ has been replaced with a single vertex $\hat{x} \in \hat{V}$. All three sets of edges, $\hat{E}_d$, $\hat{E}_f$, and $\hat{E}_w$, are updated such that the adjacency of $\hat{x}$ is the union of the adjacency of $\hat{u}, \hat{v}$. The vertices within $\hat{x}$ becomes the vertices within $\hat{u} \cup \hat{v}$. Finally, the weights of the edges in $\hat{E}_w$ that connects to $\hat{x}$ is re-calculated.*

The asymptotic complexity of FUSE is $O(\hat{E}_d + \hat{E}_f + \hat{E}_w \varpi)$ where ϖ is the complexity of calculating a weight edge. In Bohrium ϖ equals the set of vertices within the WSP problem V thus we get $O(\hat{E}_d + \hat{E}_f + \hat{E}_w V)$.

In order to simplify, hereafter when reporting complexity we use $O(V)$ to denote $O(V + \hat{V})$ and $O(E)$ to denote $O(\hat{E}_d + \hat{E}_f + \hat{E}_w + E_d + E_f)$ where E_d, E_f are the set of edges in the WSP problem. Therefore, the complexity of FUSE in Bohrium is simply $O(VE)$.

Furthermore, the FUSE function is commutaive:

Corollary 3. *Given a partition graph $\hat{G}$ and two vertices $\hat{u}, \hat{v} \in \hat{G}$, the function* FUSE$(\hat{G}, \hat{u}, \hat{v})$ *is commutaive.*

Proof. This is because FUSE is basically a vertex contraction and an union of the vertices within $\hat{u}, \hat{v}$ both of which are commutative operations[9]. □

However, it is not always legal to fuse over a weighted edge because of the preservation of the partial order of the vertices. That is, given three vertices, a, b, c, and two dependency edges, (a, b) and (b, c); it is illegal to fuse a, c without also fusing b. We call such an edge between a, c a *transitive weighted edge* and we must ignore them. Now we have:

Lemma 2. *Given a basic non-fused partition graph, $\hat{G}_1$, there exist a sequences of weighted edge fusions,* FUSE$(\hat{G}_1, \hat{u}_1, \hat{v}_1)$, FUSE$(\hat{G}_2, \hat{u}_2, \hat{v}_2)$, ..., FUSE$(\hat{G}_n, \hat{u}_n, \hat{v}_n)$, *where $(\hat{u}_i, \hat{v}_i) \in E_w[\hat{G}_i]$ and $(\hat{u}_i, \hat{v}_i)$ is non-transitive for $i = 1, 2, ..., n$, for any legal partition graph $\hat{G}_n$.*

Proof. This follows directly from Corollary 3 and the build of the basic non-fused partition graph, which has weight-edges between all pairs of fusible vertices. The fact that we ignore transitive weighted edges does not preclude any legal partition. □

Bohrium implements, IGNORE$(\hat{G}, e)$, which given a weight edge, $e \in E_w[\hat{G}]$, determines whether the edge should be ignored or not (Fig. 4). The search of the longest path (line 3) dominates the complexity of this function thus the overall complexity is $O(V + E)$.

```
1: function IGNORE($G, e$)
2:     $(u, v) \leftarrow e$
3:     $l \leftarrow$ length of longest path between $u$ and $v$ in $E_d[G]$
4:     if $l = 1$ then
5:         return $true$
6:     else
7:         return $false$
8:     end if
9: end function
```

Fig. 4. A help function thet determines whether the weight edge, $e \in E_w[G]$, should be ignored when searching for vertices to fuse.

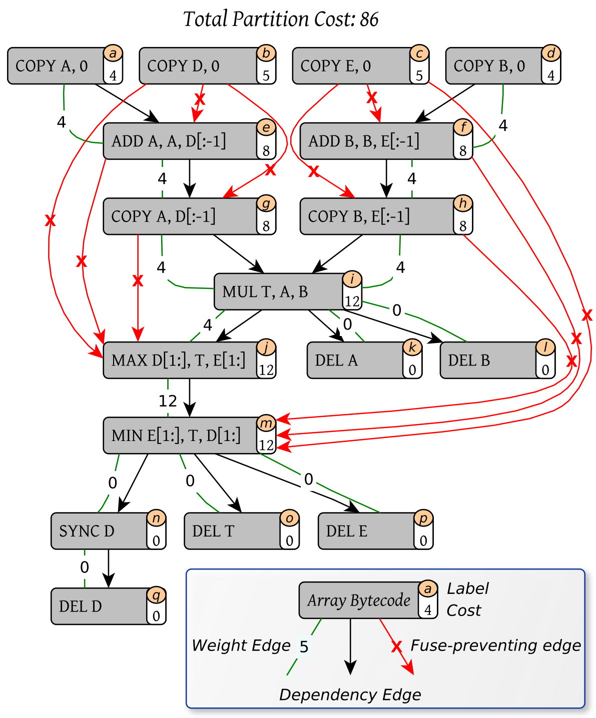


Fig. 5. A partition graph of the Python application in Fig. 3. For illustrative proposes, the graph does not include ignored weight edges (cf. Fig. 4).

C. *Greedy Fusion*

Fig. 6 shows a greedy fuse algorithm. It uses the function FIND-HEAVIEST to find the edge in E_w with the greatest weight and either remove it or fuse over it. Note that FIND-HEAVIEST must search through E_w in each iteration since FUSE might change the weights.

The number of iterations in the while loop (line 2) is $O(E)$ since minimum one weight edge is removed in each iteration either explicitly (line 5) or implicitly by FUSE (line 7). The complexity of finding the heaviest (line 3) is $O(E)$, calling IGNORE is $O(E + V)$, and calling FUSE is $O(VE)$ thus the overall complexity is $O(VE^2)$.

Fig. 7 shows a greedy partition of the Python example. The partition cost is 46, which is a significant improvement over no fusion. However, it is not the optimal partitioning, as we

```
function GREEDY($G$)
    while $E_w[G] \neq \emptyset$ do
        $(u, v) \leftarrow$ FIND-HEAVIEST($E_w[G]$)
        if IGNORE($G, (u, v)$) then
            Remove edge $(u, v)$ from $E_w$
        else
            $G \leftarrow$ FUSE($G, u, v$)
        end if
    end while
    return $G$
end function
```

Fig. 6. The greedy fusion algorithm that greedily fuses the vertices connected with the heaviest weight edge in G.

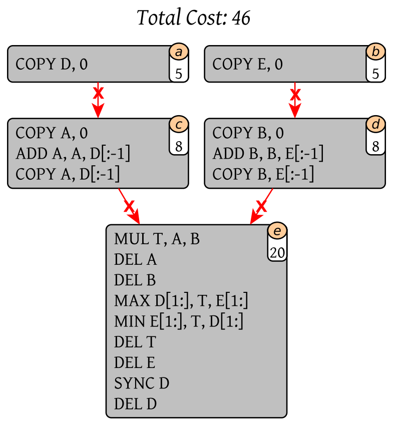


Fig. 7. A partition graph of the greedy fusion of the graph in Fig. 5.

shall see later.

D. Unintrusive Fusion

In order to reduce the size of the partition graph, we apply an unintrusive strategy where we fuse vertices that are guaranteed to be part of an optimal solution. Consider the two vertices, a, e, in Fig. 5. The only beneficial fusion possibility a has is with e thus if a is fused in the optimal solution, it is with e. Now, since fusing a, e will not impose any restriction to future possible vertex fusions in the graph. The two vertices are said to be *unintrusively fusible*:

Theorem 3. *Given a partition graph, $\hat{G}$, that, through the fusion of vertices $u, v \in V[\hat{G}]$ into $z \in V[\hat{G}']$, transforms into the partition graph $\hat{G}'$; the vertices u, v is said to be unintrusively fusible when the following conditions holds:*

1) *$\theta[z] = \theta[v] = \theta[u]$, i.e. the set of non-fusibles must not changes after the fusion.*
2) *Either u or v is a pendant vertex in graph $(V[\hat{G}], \{e \in E_w[\hat{G}] | \neg$IGNORE$(\hat{G}, e)\})$, i.e. the degree of either u or v must be 1 in respect to the weigh edges in $\hat{G}$ that are not ignored.*

Proof. The sequences of fusions that obtain an optimal partition solution cannot include the fusion of u, v into z when

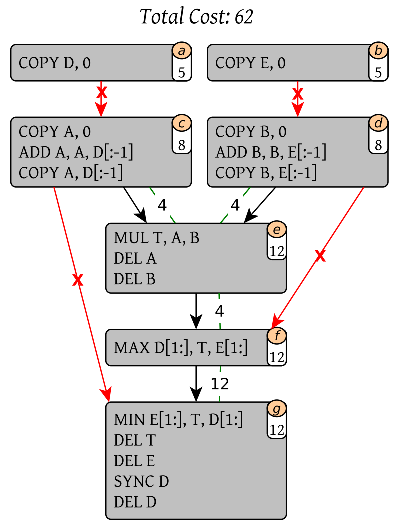


Fig. 8. A partition graph of the unintrusive fusion of the graph in Fig. 5.

two conditions exists:

1) There exist a vertex x both in G and G' that are fusible with u or v but not z and the cost saving of fusing z, u or z, v is greater than the cost saving of fusing u, v.
2) There exist two non-fusible vertices x, y both in $\hat{G}$ and $\hat{G}'$ in which the cost saving of fusing x, u and fusing v, y is not greater than the cost saving of fusing u, v.

Since we have that $\theta[z] = \theta[v] = \theta[u]$, condition (1) cannot exist and since either u or v is a pendant vertex condition (2) cannot exist. Thus, we have that the fusion of u, v is always beneficial and is part of an optimal solution, which concludes the proof. □

Fig. 9 shows the unintrusive fusion algorithm. It uses a help function, FINDCANDIDATE, to find two vertices that are *unintrusively fusible*. The complexity of FINDCANDIDATE is $O(E(E+V))$, which dominates the while-loop in UNINTRUSIVE thus the overall complexity of the unintrusive fusion algorithm is $O(E^2(E+V))$. Note that there is no need to further optimize UNINTRUSIVE since we only use it as a preconditioner for the optimal solution, which will dominate the computation time anyway.

Fig. 8 shows an unintrusive partition of the Python example with a partition cost of 62. However, the significant improvement is the reduction of the number of weight edges in the graph. As we shall see next, in order to find an optimal graph partition in practical time, the number of weight edges in the graph must be very modest.

E. Optimal Fusion

Generally, we cannot hope to solve the WSP problem in polynomial time because of the NP-hard nature of the problem. In worse case, we have to search through all possible fuse combinations of which there are 2^E. However, in some cases we may be able to solve the problems within reasonable time through a carefully chosen search strategy. For this purpose,

```
function FINDCANDIDATE($G$)                              ▷ Help function
    for $(v, u) \leftarrow E_w[G]$ do
        if IGNORE($G, (u, v)$) then
            Remove edge $(u, v)$ from $E_w$
        end if
    end for
    for $(v, u) \leftarrow E_w[G]$ do
        if the degree is less than 2 for either $u$ or $v$
         when only counting edges in $E_w[G]$ then
            if $\theta[u] = \theta[v]$ then
                return $(u, v)$
            end if
        end if
    end for
    return (NIL, NIL)
end function

function UNINTRUSIVE($G$)
    while $(u, v) \leftarrow$ FINDCANDIDATE($G$) $\neq$ (NIL, NIL) do
        $G \leftarrow$ FUSE($G, u, v$)
    end while
    return $G$
end function
```

Fig. 9. The unintrusive fusion algorithm that only fuse *unintrusively fusible* vertices.

we implement a branch-and-bound algorithm that explores the monotonic decreasing property of the partition cost (Lemma 1).

Consider the result of the unintrusive fusion algorithm (Fig. 8). In order to find the optimal solution, we start a search down through a tree of possible partitions. At the root level of the search tree, we check the legality of a partition that fuses over all weigh edges. If the partition graph is legal, i.e. it did not fuse vertices connected with fuse-preventing edges, than it follows from Lemma 1 that the partition is optimal. If the partition is not legal, we descend a level down the tree and try to fuse over all but one weight edge. We continue this process such that for each level in the search tree, we fuse over one less weight edge. We do this until we find a legal partition (Fig. 10).

Furthermore, because of Lemma 1, we can bound the search using the cheapest legal partition already found. Thus, we ignore sub-trees that have a cost greather than the cheapest already found.

Fig. 11 shows the implementation and Fig. 12 shows an optimal partition of the Python example with a partition cost of 34.

F. Naïve Fusion

For completeness, we also implement a partition algorithm that does not use a graph representation. In our naïve approach, we simply go through the array operation list and add each array operation to the *current* partition block unless the array operations makes the current block illegal, in which case we add the array operation to a new partition block, which then becomes the current one. The asymptotic complexity of this algorithm is $O(n^2)$ where n is the number of array operations.

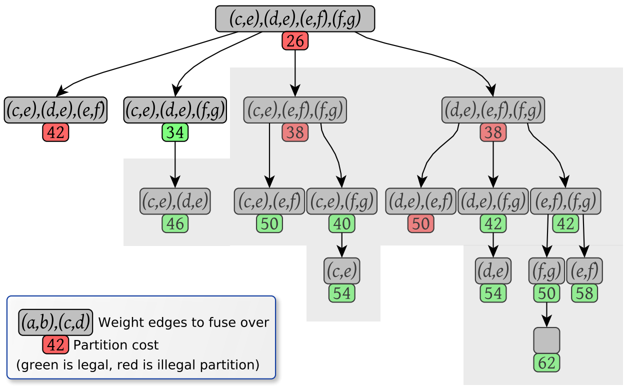


Fig. 10. A branch-and-bound search tree of the unintrusively fused partition graph (Fig. 8). Each vertex lists a sequences of vertex fusions that build a specific graph partition. The grayed out area indicates the part of the search tree that a depth-first-search can skip because of the cost bound.

Fig. 13 show that result of partitioning the Python example with a cost of 50.

G. Fuse Cache

In order to amortize the runtime of applying the fuse algorithms, Bohrium implements a fuse cache of previously found partitions of array operation lists. It is often the case that scientific applications use large calculation loops such that an iteration in the loop corresponds to a list of array operations. Since the loop contains many iterations, the cache can amortize the overall runtime time.

V. EVALUATION

In this section, we will evaluate the different partition algorithm both theoretically and practically. We execute a range of scientific Python benchmarks, which are part of an open source benchmark tool and suite named Benchpress[3]. Table I shows the specific benchmarks that we uses and Table II specifies the host machine. When reporting runtime results, we use the results of the mean of eight identical executions as well as error bars that shows two standard deviations from the mean.

We would like to point out that even though we are using benchmarks implemented in pure Python/NumPy, the performance is comparable to traditional high-performance languages such as C and Fortran. This is because Bohrium overloads NumPy array operations[8] in order to JIT compile and execute them in parallel seamlessly[cite simon].

1) Theoretical Partition Cost: Fig. 14 shows that theoretical partition cost (Def. 10) of the four different partition algorithms previously presented. Please note that the last five benchmarks do not show an optimal solution. This is because the associated search trees are too large for our branch-and-bound algorithm to solve. For example, the search tree of the Lattice Boltzmann is 2^{664}, which is simply too large

[3]Available at http://benchpress.bh107.org. For reproducibility, the exact version can be obtained from the source code repository at https://github.com/bh107/benchpress.git revision `01e84bd995`.

```
function FUSEBYMASK($G, M$)                              ▷ Help function
    $f \leftarrow$ true                          ▷ Flag that indicates fusibility
    for $i \leftarrow 0$ to $|E_w[G]| - 1$ do
        if $M_i = 1$ then
            $(u, v) \leftarrow$ the $i$'th edge in $E_w[G]$
            if not FUSIBLE($G, u, v$) then
                $f \leftarrow$ false
            end if
            $G \leftarrow$ FUSE($G, u, v$)
        end if
    end for
    return $(G, f)$
end function

function OPTIMAL($G$)
    $G \leftarrow$ UNINTRUSIVE($G$)
    for $(v, u) \leftarrow |E_w[G]|$ do
        if IGNORE($G, (u, v)$) then
            Remove edge $(u, v)$ from $E_w$
        end if
    end for
    $B \leftarrow$ GREEDY($G$)                      ▷ Initially best partitioning
    $M_{0..|E_w[G]|} \leftarrow 1$                     ▷ Fill array $M$ with ones
    $o \leftarrow 0$                                          ▷ The mask offset
    $Q \leftarrow \emptyset$
    ENQUEUE($Q, (M, o)$)
    while $Q \neq \emptyset$ do
        $(M, o) \leftarrow$ DEQUEUE($Q$)
        $(G', f) \leftarrow$ FUSEBYMASK($G, M$)
        if $cost(G') < cost(B)$ then
            if $f$ and $G'$ is acyclic then
                $B \leftarrow G'$                       ▷ New best partitioning
            end if
        end if
        for $i \leftarrow o$ to $|M| - 1$ do
            $M' \leftarrow M$
            $M'_i \leftarrow 0$
            ENQUEUE($Q, (M', i + 1)$)
        end for
    end while
    return $B$
end function
```

Fig. 11. The optimal fusion algorithm that optimally fuses the vertices in G. The function, $cost(G)$, returns the partition cost of the partition graph G.

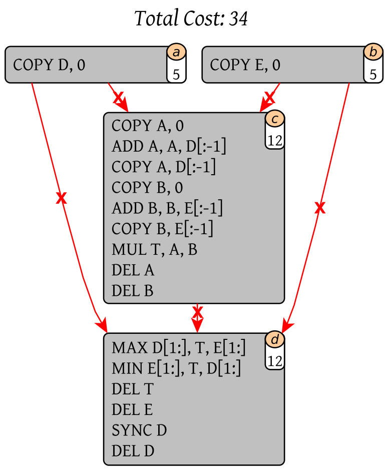


Fig. 12. A partition graph of the optimal fusion of the graph in Fig. 5.

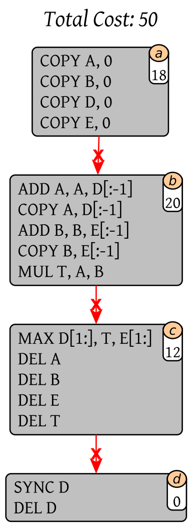


Fig. 13. A partition graph of a Naïve partition of the Python example (Fig. 3).

Benchmark	Input size (in 64bit floats)	Iterations
Black Scholes	$1.5{\times}10^6$	20
Game of Life	10^8	20
Heat Equation	$1.44{\times}10^8$	20
Leibnitz PI	10^8	20
Gauss Elimination	2800	2799
LU Factorization	2800	2799
Monte Carlo PI	10^8	20
27 Point Stencil	$4.2875{\times}10^7$	20
Shallow Water	$1.024{\times}10^7$	20
Rosenbrock	$2{\times}10^8$	20
Successive over-relaxation	$1.44{\times}10^8$	20
NBody	6000	20
NBody Nice	40 plantes, $2{\times}10^6$asteroids	20
Lattice Boltzmann D3Q19	$3.375{\times}10^6$	20
Water-Ice Simulation	$6.4{\times}10^5$	20

TABLE I
BENCHMARK APPLICATIONS

Processor:	Intel Core i7-3770
Clock:	3.4 GHz
#Cores:	4
Peak performance:	108.8 GFLOPS
L3 Cache:	16MB
Memory:	128GB DDR3
Operating system:	Ubuntu Linux 14.04.2 LTS
Software:	GCC v4.8.4, Python v2.7.6, NumPy 1.8.2

TABLE II
SYSTEM SPECIFICATIONS

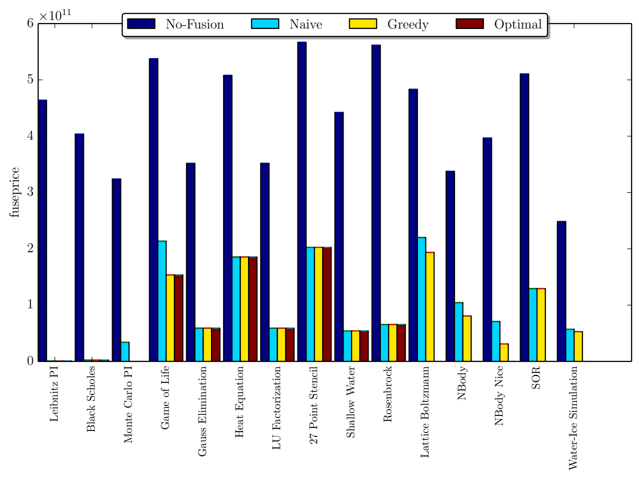


Fig. 14. Theoretical cost of the different partition algorithms. NB: the last five benchmarks, Lattice Boltzmann, NBody, NBody Nice, SOR, Water-Ice Simulation, does not show an optimal solution.

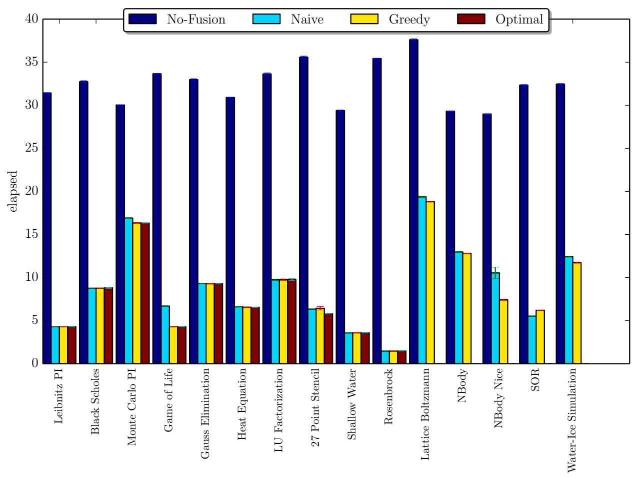


Fig. 15. Runtime of the different partition algorithms using a **warm cache**.

even when the bound can cut much of the search tree away. As expected, we observe that the three algorithms that do fusion, Naïve, Greedy, and Optimal, have a significant smaller cost than the non-fusing algorithm Singleton. The difference between Naïve and Greedy is significant in some of the benchmarks but the difference between greedy and optimal does almost not exist.

2) Practical Runtime Cost: In order to evaluate the full picture, we do three runtime measurements: one with a warm fuse cache, one with a cold fuse cache, and one with no fuse cache. Fig. 15 shows the runtime when using a warm fuse cache thus we can compare the theoretical partition cost with the practical runtime without the overhead of running the partition algorithm. Looking at Fig. 14 and Fig. 15, it is evident that our cost model, which is a measurement of unique array accesses (Def. 10), compares well to the practical runtime result in this specific benchmark setup. However, there are some outliners – the Monte Carlo Pi benchmark has a theoretical partition cost of 1 when using the Greedy and Optimal algorithm but has a significantly greater practical runtime. This is because the execution becomes computation bound rather than memory bound thus a further reduction in memory accesses does not improve performance. Similarly, in the 27 Point Stencil benchmark the theoretical partition cost is identical for Naïve, Greedy, and Optimal but in practices the Optimal is marginal better. This is an artifact of our cost model, which define the cost of reads and writes identically.

Fig. 16 shows the runtime when using a cold fuse cache such that the partition algorithm runs once in the first iteration of the computation. The results show that 20 iterations, which most of the benchmarks uses, is enough to amortize the partition overhead. Whereas, when running the partition algorithm in each iteration, which is the case when running with no fuse cache (Fig. 17), the Naïve partition algorithm outperforms both the Greedy and Optimal algorithm because of its smaller time complexity.

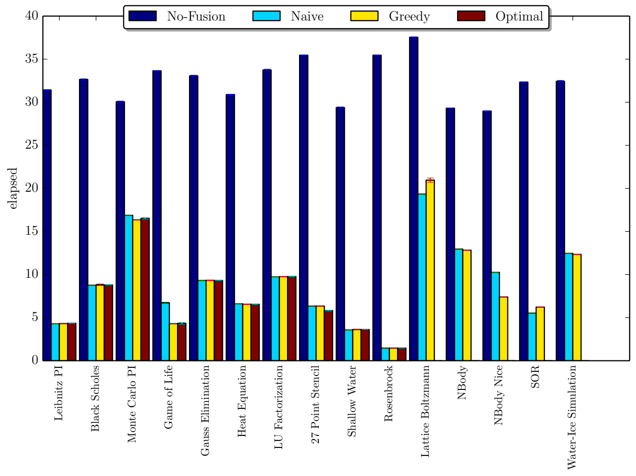


Fig. 16. Runtime of the different partition algorithms using a **cold cache**.

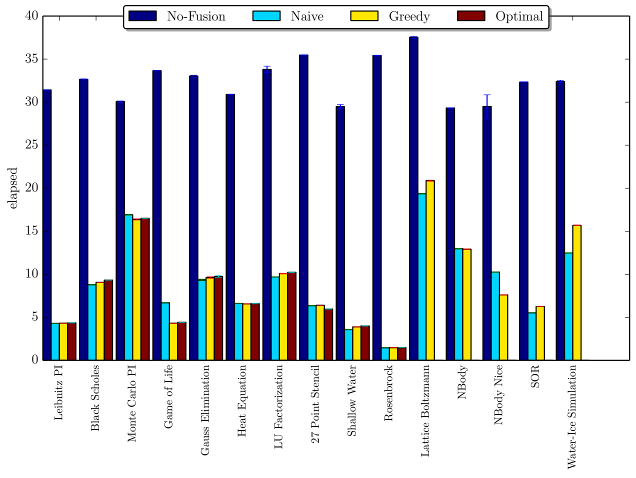


Fig. 17. Runtime of the different partition algorithms using **no cache**.

References

[1] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath, “Collective loop fusion for array contraction,” in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Springer Berlin Heidelberg, 1993, vol. 757, pp. 281–295.

[2] D. Loveman, “High performance fortran,” *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 1, no. 1, pp. 25–42, 1993.

[3] B. Chamberlain, S.-E. Choi, C. Lewis, C. Lin, L. Snyder, and W. Weathersby, “Zpl: a machine independent programming language for parallel computers,” *Software Engineering, IEEE Transactions on*, vol. 26, no. 3, pp. 197–211, Mar 2000.

[4] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter, “Bohrium: a Virtual Machine Approach to Portable Parallelism,” in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 2014, pp. 312–321.

[5] K. Kennedy and K. McKinley, “Maximizing loop parallelism and improving data locality via loop fusion and distribution,” in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1994, vol. 768, pp. 301–320.

[6] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis, “The complexity of multiway cuts,” in *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*. ACM, 1992, pp. 241–251.

[7] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter, “Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster,” in *4th Workshop on Python for High Performance and Scientific Computing (PyHPC’13)*, 2013.

[8] M. R. B. Kristensen, S. A. F. Lund, T. Blum, and K. Skovhede, “Separating NumPy API from Implementation,” in *5th Workshop on Python for High Performance and Scientific Computing (PyHPC’14)*, 2014.

[9] T. Wolle, H. L. Bodlaender *et al.*, “A note on edge contraction,” Technical Report UU-CS-2004, Tech. Rep., 2004.