Automatic Parallelization of Scientific Application

Troels Blum
blum@nbi.ku.dk

Supervisor:
Brian Vinter
vinter@nbi.ku.dk

October 2015

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Contributions . . . 4
- 1.3 Publications . . . 6

2 Background **8**
- 2.1 Computational Science . . . 8
  - 2.1.1 Productivity . . . 8
- 2.2 Parallel Programming Models . . . 10
  - 2.2.1 Shared Memory . . . 10
  - 2.2.2 Message Passing . . . 13
  - 2.2.3 Vector Based Programming . . . 14
- 2.3 Programming the GPU . . . 15
  - 2.3.1 Architecture . . . 16
  - 2.3.2 Programming model . . . 18

3 Bohrium **21**
- 3.1 Design . . . 22
  - 3.1.1 Vector Byte-code . . . 24
  - 3.1.2 Bridge . . . 29
  - 3.1.3 Vector Engine . . . 30
  - 3.1.4 Example . . . 32
  - 3.1.5 Vector Engine Manager . . . 32
  - 3.1.6 Configuration . . . 34
- 3.2 The Bohrium NumPy Bridge . . . 34

4 The Bohrium GPU Vector Engine **36**
- 4.1 JIT Compilation . . . 37
- 4.2 Data Management . . . 39
- 4.3 Code Specialization . . . 42
  - 4.3.1 Limitations . . . 42

4.3.2 Strategy 43

5 Ongoing Work **47**

6 Conclusion **48**

7 Publications **49**
7.1 cphVB: A Scalable Virtual Machine for Vectorized Applications 49
7.2 Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster 58
7.3 Bohrium: a Virtual Machine Approach to Portable Parallelism 67
7.4 Transparent GPU Execution of NumPy Applications 78
7.5 Separating NumPy API from Implementation 88
7.6 Code Specialization of Auto Generated GPU Kernels 98
7.7 Fusion of Array Operations at Runtime 113

Chapter 1

Introduction

Computer simulations, which are widely used in both academia and the industry, often consists of large compute intensive tasks. At the same time there is a continuing, and growing, interest in constructing mathematical models and quantitative analysis techniques, i.e. computational science. These are good candidates for harvesting the computing power of modern, highly parallel computing systems, such as Graphics processing units (GPU) and other massively parallel accelerator cards like the Xeon Phi. These technologies promise to deliver more "bang for the buck" over conventional CPUs. The challenge lies in the fact, that these systems must be programmed using specialized programming models, which, even for skilled programming professionals, make the development cycle very long. This is a big problem in an environment which relies on an iterative method of developing new models. Alternatively programs that are written by domain experts, but they do not have the knowledge to program the highly parallel systems. In the best case the program ends up not utilizing the hardware properly, more likely the program will simply be sequential in nature and to slow. Parallelization of existing sequential programs is both a bug prone and time consuming task. All in all this is a big and costly problem for both academic and industrial communities.

1.1 Motivation

Much research time is spent on transforming codes from prototypes in high productivity array-based languages, such as Matlab or Python/NumPy, into compilable languages, because the scientist expect performance improvements of an order of magnitude.

Most scientists within physics, chemistry, geology, etc., are naturally con-

fortable with array-based expression of their problems, largely because their first approach to any programming problem is usually Matlab or Python/Numpy, where array-based notation is both natural and essential to achieve even a reasonable performance. Once a correctly functioning prototype has evolved, the code is often ported to C/C++ or Fortran for performance improvement. Little work has been done on formally verifying the speed difference between Matlab/Numpy and compiled languages, but a blog [1] does some structured experiments and end up with the conclusion that on a Jacobi solver in Numpy is approximately 8-9 times slower than Intel Fortran depending on the dimensionality of the problem. This is inline with common assumption that Numpy is approximately 10 times slower than compiled code; Matlab is slightly slower than that.

Thus, it is reasonable to conclude that much researcher time is spent transforming solutions from Matlab/Numpy versions into compileable code, in order to reach an order of magnitude in performance. In addition to using researcher resources, and delaying progress, it also stands to reason that this conversion process is a source of errors in the final version.

1.2 Contributions

The majority of the research work I have done as part of my PhD has been implemented in the Bohrium project. The Bohrium project is an open source project which is being developed in collaboration with Mads R. B. Kristensen, Simon A. F. Lund and Kenneth Skovhede. All of whom have been valuable research partners.

Bohrium Architecture

The idea behind Bohrium is to separate the programming language, or front-end, from the execution engine. Allowing the programmer to be oblivious of the hardware and specific programming model for the given hardware. To my knowledge this is a novel approach to closing the gap between high productivity languages and highly parallel architectures.

Bohrium consists of a number of components which communicate by the defined *Vector Bytecode* language. Some components, like the execution engine are architecture specific; others are language specific, like the language bridges; fusers, filters and managers are neither. Bohrium relies on lazy evaluation and Just In Time (JIT) compilation for performance benefits.

A short description of the different component types:

Bridge This is the users programming interface to Bohrium. The Bridge may integrate Bohrium into an existing programming language as a library, like the C++ and CIL bridge does. It cal also integrate into an existing library, as is the case with the NumPy bridge. The bridge could also define a new domain specific language (DSL), or implement an existing language, the MiniMatlab bridge is somewhere between these two. The Bridge generates the Bohrium bytecode that corresponds to the users program at runtime.

Vector Engine *Manager* The Vector Engine Manager (VEM) is named so for historical reasons. As the original design contained a bridge, a VEM, and one or more Vector engines. The current design is more flexible, and as a result a VEM may manage any component below the language bridge, even other VEMs. The VEM's role is to manage the data location and ownership of arrays. It also manages the distribution of computing jobs between potentially several Vector Engines.

Vector Engine The Vector Engines (VE) are the components in Bohrium which execute the calculations described by the vector bytecodes it receives. The vector engines are architecture-specific.

Fuser Originally it was up to the vector engines, specifically the GPU vector engine, to combine multiple Bohrium vector bytecodes into suitable kernels. However we realized that it would be possible to devise a strategy for combining the vector bytecodes into kernels that are beneficial and well suited for a wide variety of vector engines and hardware.

The fuser combines, or fuses, multiple array operations into a single *kernel* of operations. The main benefit of which is temp array elimination, and ensuring that input and output arrays are read and written as few times as possible. Limiting the stress in the memory bandwidth of the system. Since the fuser is now a component it is possible to test and compare different strategies for kernel fusion.

Filter The main idea behind introducing filters into Bohrium is a means to bytecode transformation, but other uses have proven useful. The reason to introduce a means to bytecode transformation is twofold. One, to keep the vector engines simpler. The vector engines are already very complex, but at least they can be kept conceptually simple: execute the calculations described by the vector bytecode. With filters some optimizations and restrictions can be implemented here. The second benefit is that many optimizations are applicable across vector engines.

This type of optimization can generally be expressed ad bytecode transformations.

I played a central role in the design and implementation of the Bohrium system.

Bohrium Bridge for Numerical Python

There are several different language bridges included in the Bohrium project. There is a C and a C++ bridge, there is a .NET bridge for the Microsoft .NET languages, and there is a small Matlab interpreter. The first bridge, and the one that has received the most attention is the NumPy bridge. NumPy is widely used in the scientific computing community, and it is open source. I have also played a major role in the development of the NumPy bridge, while Mads R. B. Kristensen has been the main contributor due to his vast knowledge about Numerical Python.

GPU Execution Engine

My largest contribution to the Bohrium project has been the GPU vector engine. Which facilitates the execution of Bohrium byte code on any GPU with OpenCL support. The development of the GPU vector engine has inspired some major changes and improvements to the rest of the Bohrium system. Most prominently the architecture independent kernel construction which is being documented in the paper "Fusion of Array Operations at Runtime" is a generalization of how kernels were built in the GPU vector engine, see Section 7.7. This in turn has improved the GPU vector engine, as the kernels have become more general and include more instructions and concepts. The GPU vector engine now streaming by including generators and reductions in the same kernels with element wise instructions this work is unfortunately at this time undocumented.

1.3 Publications

cphVB: A Scalable Virtual Machine for Vectorized Applications

Mads Ruben Burgdorff Kristensen, Simon Andreas Frimann Lund, Troels Blum, Brian Vinter.
Proceedings of The 11th Python In Science Conference (SciPy'12). Austin, Texas, USA.

Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster

Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter.
4th Workshop on Python for High Performance and Scientific Computing (PyHPC 2013)

Bohrium: a Virtual Machine Approach to Portable Parallelism

Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter.
28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014)

Transparent GPU Execution of NumPy Applications

Troels Blum, Mads R. B. Kristensen, and Brian Vinter.
28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014)

Separating NumPy API from Implementation

Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, and Kenneth Skovhede.
5th Workshop on Python for High Performance and Scientific Computing (PyHPC 2014)

Code Specialization of Auto Generated GPU Kernels

Troels Blum and Brian Vinter.
Communicating Process Architectures 2015

Fusion of Array Operations at Runtime

Mads R. B. Kristensen, Troels Blum, Simon A. F. Lund, and James Avery.
To be submitted.

Chapter 2

Background

2.1 Computational Science

Computer simulations, which are widely used in both academia and the industry, often consists of large compute intensive tasks. This makes them good candidates for harvesting the computing power of modern, highly parallel computing systems, such as GPUs. The challenge lies in the fact, that these systems must be programmed using specialized programming models, which, even for skilled programming professionals, make the development cycle very long. This is a big and costly problem for both academic and industrial communities, which rely on an iterative method of developing new models.

2.1.1 Productivity

There is a continuing, and growing, interest in constructing mathematical models and quantitative analysis techniques, i.e. computational science, in both academia and in industry. This results in an increasing number of programs that are written by domain experts, as opposed to trained programmers. At the same time the availability and power of accelerator cards, e.g. Graphics processing units (GPU), and coprocessors, e.g. Xeon Phi, is also increasing. These technologies promise to deliver more bang for the buck over conventional CPUs. However, these technologies require programming experts, i.e. engineers and computer scientists, to program.

The gap between these fields of expertise can be overcome by employing both domain experts and programming experts. However, this solution is often both expensive and time consuming, since coordination between scientists and programmers constitutes an overhead. Other possible solutions include developing domain specific languages (DSLs). Such DSLs may either be

compiled or interpreted, in both cases JIT compilation may be involved, for example Fortress[6] and X10[17] running on the Java Virtual Machine. One may port or develop accelerator enabled libraries like QDP-JIT/PTX[69], which is a lattice quantum chromodynamics (QCD) calculation library, a port of the QDP++[23] library, which may help domain experts to leverage the power of accelerators without learning accelerator based programming, naturally the performance imprpovements is then limited to the portion of the code that uses accelerated libraries.

For general purpose programming languages such as C/C++ or Fortran annotation in the shape of pragmas as in OpenACC[24] is often used to annotate parts of the code that is well suited for execution on the GPU. This may not in fact bridge the gap between domain and programming expertise, as both the languages and correct use of pragmas require a high level of programming and architecture knowledge. Template libraries like Thrust[10] from NVIDIA or Bolt[7] from AMD are also available. They save the programmer the trouble of writing some boiler plate code, and contain implementations of standard algorithms. Again some architecture knowledge is required, and most problems can not simply be solved by gluing standard algorithms together. SyCL[63], a specification from the Khronos group, and C++ AMP[20], from Microsoft, are both single source solution compilers for parallel, heterogeneous hardware. This allows for GPU kernel code to be templated, and methods can be implemented on vector data types for seamless parallelization. All of these technologies, while useful for skilled programmers, are of little help to domain experts as they still require advanced programming skills and hardware knowledge.

An increasingly popular choice for domain experts is to turn to interpreted languages[53] like MATLAB[3] or Python with the scientific computing package NumPy[51]. These languages allow the scientist to express their problems at higher level of abstraction, and thus improves their productivity as well as their confidence in the correctness of their code. The function decorators of SEJITS[16] is one way to utilize accelerators (GPUs) from Python, another is project Copperhead[14] which rely on decorators to execute parts of the Python code on GPUs through CUDA[50]. It is also possible is to use a more low level framework such as pyOpenCL/pyCUDA[33], which provides tools for writing GPU kernels directly in Python. The user writes OpenCL[48] or CUDA specific kernels as text strings in Python. This allows for the control structure of the program to be written in Python while also avoiding writing boilerplate OpenCL- or CUDA code. This still required programming knowledge of OpenCL or CUDA.

Systems such as pyOpenCL/pyCUDA[33] provides tools for interfacing a high abstraction front-end language with kernels written for specific po-

tentially 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.

unPython[27] is a compilation framework 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. Which requires the user to modify the source code manually, by applying hints in a manner similar to that of OpenMP.

2.2 Parallel Programming Models

Modern computing hardware is inherently parallel, CPUs are multicore, and accelerator cards like GPGPUs and Xeon Phi are many core. However, the idea of performing computational tasks in parallel is almost as old as the computers themselves[28]. Parallelism exists on many levels, from independent tasks that running on different computing nodes, to Streaming SIMD Extensions (SSE) capabilities in modern CPUs.

Through the years, several programming models and methods have been developed to reduce the complexity of parallel programming. This chapter is not intended be a complete survey of parallel programming methods, but an overview of the most common methods and their benefits and drawbacks.

2.2.1 Shared Memory

In the shared memory programming model every processing unit, logical of physical, share access to the same memory. The model does not impose any restrictions on the program on how to access the memory. In the most basic case, all threads in a process share all resources, including memory, and cooperate in solving a computational task. Processes, however, do not, in general, share any memory or resources. The shared memory programming model is widely used in symmetric multiprocessing (SMP) architectures like multicore CPUs.

Since threads work independently on common data structures, and communicate through shared variables some means of synchronization is needed. We need locks, semaphores, and mutexes to control the execution flow of threads and access to data to ensure correct program execution. Many programming languages support threads, either as a native feature like C++11 [60] or through libraries like Java's thread class[2]. Higher level languages, such as Python[4], dictate a thread model, and abstracts the actual thread

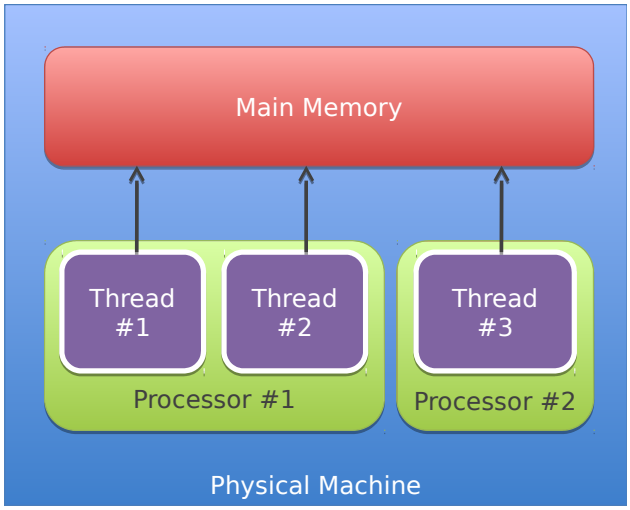


Figure 2.1: Multiple processes on a shared memory machine

implementation. Languages like C[1] and Fortran do not provide native support for threads. They provide access to threads through an API to the operating system's thread library. The POSIX standard is the most ubiquitous thread library standard, and defines an API for creating and manipulating threads including access control.

Most hardware systems support shared memory. The x86 architecture originally did not have any hardware features to restrict access to memory and as such had only shared memory[19]. With the 80286 Intel started supporting *protected mode*[19] in their processors. This feature enabled the operating system (OS) to prevent one process from accessing the data of another. The primary motivation was not to ensure security and OS stability, but rather to allow memory segmentation. Enabling the system to utilize more than 1MB of memory, which was originally the limit[19]. The support, and switching between *real mode* and *protected mode* improved with the 80386 processor[18]. Figure 2.1 shows an example of a shared memory machine.

Open Multi-Processing

Open Multi-Processing (OpenMP)[22] is a multi-platform shared-memory programming extension for writing parallel programs for C/C++ and Fortran. The specification states how memory consistency must work which makes OpenMP based code more portable across different systems[54]. OpenMP

[1]C++ only started having native support for threads after the C++11 standard.

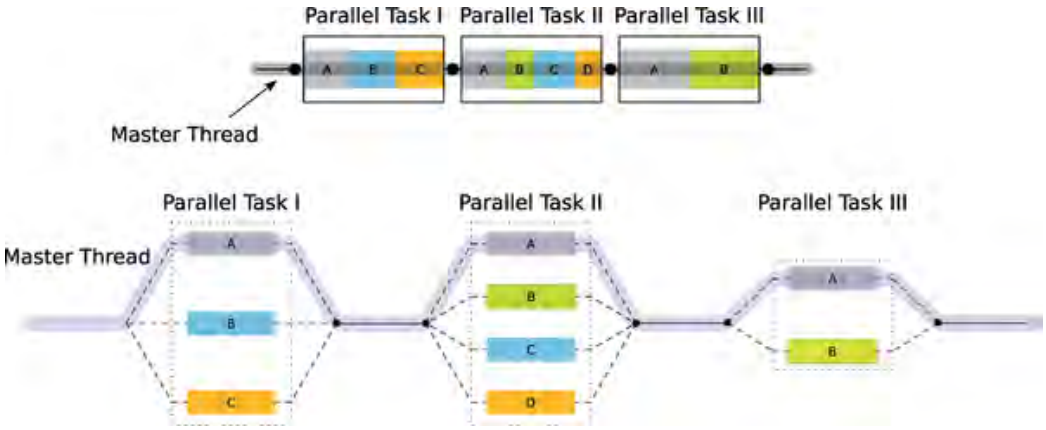


Figure 2.2: The Fork/Join parallel paradigm. The master thread forks off a number of threads which execute blocks of code in parallel.

also makes it easier to create parallel application by using the fork/join programming paradigm, see Figure2.2. This encapsulates much of the tedious work involved in multi-threading programming, such as creating, joining and destroying threads. OpenMP also provides a parallelization strategies for implementing common code structures, such as automatic parallelization of for-loops that which no dependent iterations.

The fork/join programming paradigm (Figure 2.2) implemented by OpenMP begins execution as a single process, called the master thread of execution. When the master threads enters a parallel section in the program, it forks a team of threads (one of them being the master thread), and work is continued in parallel among these threads. Upon exiting the parallel section, all the threads in the team synchronize (join the master), and only the master continues execution. The fundamental directive for expressing parallelism on OpenMP is the `parallel directive`.

All statements in the parallel section of the program, including function calls, are executed in parallel by each thread in the team. It is up to the programmer to classify all variables as either shared or private, to the thread. The compiler uses this information to ensure thread safe access to the variables. In other words OpenMP does not hide the parallelism from the user, but provides convenient language constructs to write parallel programs.

Numerical Libraries

Libraries, and especially numerical libraries is another way of parallelizing certain sections of a program. The libraries implement commonly used functions and subroutines which the user can the use to create his program. From

the view of the programmer certain functionality is encapsulated, and it becomes possible to write seemingly sequential code where the computational intensive parts executes in parallel. From a portability perspective, the libraries can also support multiple platforms and expose the same interface, making it easier to write portable code. The libraries place some burden on the programmer who is responsible for dividing the program into tasks that are suitable for the library. The two most commonly used numerical libraries for linear algebra are BLAS[41] and LAPACK[9].

2.2.2 Message Passing

Message passing is very broad parallel programming paradigm, witch is implemented by very different libraries, systems, and languages. The one thing they have in common is that data messages are being sent over some communication channel.

Parallel Virtual Machine (PVM)[61] is a software system that enables a collection of heterogeneous computers to be used as one Distributed Memory Machine. PVM is built around the concept of a *virtual machine* which is a dynamic collection of (potentially heterogeneous) computational resources managed as a single parallel computer. One aspect of the virtual machine is how parallel tasks exchange data. This is accomplished using simple message passing constructs. The virtual machine transparently handles message routing and data conversion for incompatible architectures.

Message Parsing Interface (MPI)[59] focuses on a more tightly bound communication paradigm, in which a cluster of homogeneous nodes is preferred. To limit the amount of memory which needs to be copied when transmitting data, the MPI standard supports user-defined data types. This makes it possible to send and receive non-contiguous data blocks: A typical use case is when working with data in a matrix structure, and column of data needs to be communicated.

Communicating Sequential Processes (CSP)[30] is very different from both PVM and MPI. It is a formal language for describing interaction between concurrent processes. The basic components of CSP are processes and channels. By using the abstraction of a channel, it is possible to formally verify that the program is correct. The channel also serves the purpose of hiding the implementation details. Using the CPS programming paradigm it is simpler to describe highly irregular problems, as each process is isolated.

2.2.3 Vector Based Programming

The Python programming language and its de-facto scientific library NumPy[51] targets the academic and the industrial community as a high-productivity framework with a very short development cycle. Python/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 frequently used in programming languages and libraries that target the scientific community and the high-technology industry, e.g. HPF[42], MATLAB[70], Armadillo[56], and Blitz++[68].

Microsoft Accelerator [62] 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) [49] 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.

The concept of *views* is essential to NumPy programming. A view is a reference to a subpart of an array that appears as a regular array to the user. Views make it possible to implement a broad range of applications through element-wise vector (or array) operations. In Figure 3.3, we implement a heat equation solver that uses views to implement a 5-point stencil computation of the domain.

On multiple points, Bohrium is closely related in functionality and goals to the SEJITS [15] 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 . This approach is

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

Figure 2.3: Python/NumPy implementation of a 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 Bohrium and our GPU backend.

shown to provide performance that at times out-performs even hand-written specialized code towards a given architecture[46]. Being able to construct computational kernels is a core issue in data-parallel programming. 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 [8] was based on a complete virtual machine for generic execution whereas Bohrium provides an optimized subset.

2.3 Programming the GPU

The Graphics card industry, driven by the gaming marker, has found the need for specialized processors used to perform shading. When rendering a 3D object onto a 2D screen, each pixel can to be calculated individually to create a realistic representation. This has lead to Graphics Processing Units (GPU) with a set of specialized floating-point processors that are specifically targeted at performing shading. The shaders became programmable which made it possible to running programs on them. Although the input had to be represented as textures and polygons, and the output would be pixel values[57]. Even with these constraints the need for compute power in scien-

tific computing led to research in utilizing the GPU shaders for non graphical use[29].

In 2007 NVIDIA announced CUDA[5], which made it possible to use the GPU for General Purpose processing (GPGPU)

2.3.1 Architecture

A GPGPU device is a computing device, separate from the host system. It has its own memory hierarchy with a separate address space. Data has to be explicitly copied to and from the device for manipulation and viewing. This is typically done over the PCI-express bus, which makes it an expensive operation, even for simple memory copy.

The GPGPU consists of a number of multiprocessing units, each containing several cores. The cores are the hardware units doing the actual computations.

Memorywise the device contains a global memory with random access. Each multiprocessor contains some memory called shared memory. This is shared by the cores belonging to the multiprocessor, and only accessible to those. The shared memory can be thought of, as a user managed cache, although in newer generations of GPGPUs the default is to have part of the shared memory used as a traditional cache, controlled by the driver. Each compute core has access to a private register file. A schematic model a generic GPGPU can be seen in figure 2.4.

The GPUs global memory has very high bandwidth (tens to hundreds of GB/s). This is achieved using a wide data path,up to 512 bits. Resulting in reads of 128 bytes at a time. This is important to have in mind when accessing global memory. The access time, on the other hand, is high compared to the instruction cycle time. It takes in the order of 100 instructions to access global memory. The register file and shared memory are both accessed directly in one instruction.

The GPU architecture supports light weight hardware threads. The threads are light weight in the sense that threads can be scheduled for execution within a single clock cycle. Threads running on the same multiprocessor execute the same instruction at the same time. NVIDIA calls this a Single Instruction Multiple Threads (SIMT) architecture.

SIMT is akin to Single Instruction, Multiple Data (SIMD), with the key difference that SIMD is data-centric, where SIMT is execution-centric. It is however, very simple to map SIMT to SIMD by mapping every data point to a thread. This is in fact a recommended abstraction, and is very good for latency hiding, as long as the problem at hand allows for it.

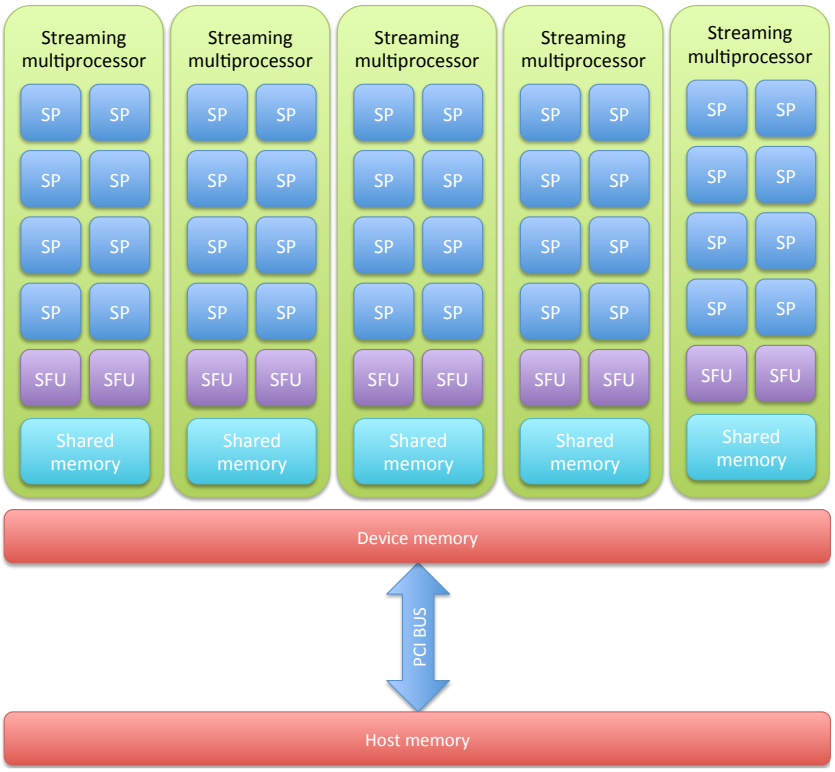


Figure 2.4: Generic GPGPU hardware setup

2.3.2 Programming model

The fundamental concept in programming GPGPU's is the *kernel*. A kernel is basically a function, whose body is executed N time by N different concurrent threads. Take for example the pseudo c-style function in listing 2.1, for increasing the brightness of a RGB image. It does this by running through all the pixels, and increasing the RGB-value by delta.

```
void increase_brightness(rgb* img, int height, int width, int delta)
{
    for (int h = 0; i < height; ++h)
    {
        for (int w = 0; w < width; ++w)
        {
            img[h][w].red   += delta;
            img[h][w].green += delta;
            img[h][w].blue  += delta;
        }
    }
}
```

Code Listing 2.1: Classic C style

The same function can easily be rewrite into a kernel for execution on a GPU. That would look something like the pseudo code in listing 2.2. Here the width and the height is coded into the *kernel shape*. In general a kernel has a shape which represents the layout of the threads in one, two or three dimensions. I will not go further into details with this, but instead refer the reader to the CUDA C programming guide [21].

```
__global__
void increase_brightness(rgb* img, int delta)
{
    int h = threadIdx.x;
    int w = threadIdx.y;

    img[h][w].red   += delta;
    img[h][w].green += delta;
    img[h][w].blue  += delta;
}
```

Code Listing 2.2: CUDA C style

Cooperative Threads

To help manage resources and enable some synchronization, Threads are divided into *blocks*. A thread block can consist of up to 512 threads. The threads that make up a block are able to communicate via shared memory, and can synchronize execution and communication via special function calls. This means that all threads belonging to a block will be running on the same multi-processor, although not necessarily at the same time. Threads running in parallel in a single multiprocessor are called a *warp*. Threads in a warp always belong to the same thread block.

Threads that belong to the same warp execute the same operation at the same time. therefore it comes at great cost, if threads within a warp diverge in their execution path. As all paths will have to be followed by all threads. Only memory reads and writes are the selected appropriately.

Threads that belong to the same block share all resources. That includes register space. So even though threads cannot access each other registers, the amount of threads in a thread block, limits the number of registers available to each thread.

The threads in a tread block can be laid out in a one, two or three dimensional pattern. Each thread can retrieve its position within the block via special registers.

When invoking a kernel, it is done with an array of thread blocks, called a Cooperative Thread Array (CTA). The thread blocks can be laid out in one or two dimensions within the CTA. For this reason it is often referred to as a grid (of thread blocks). Threads belonging to the same CTA, but different blocks can only communicate via global memory. In that sense they have no special relation, other than belonging to the same kernel. A thread can retrieve information about which thread block it belongs to, in the same way as its position within a block. That way it can calculate its global position. This information is often used to calculate which data elements to access from global memory.

Kernel Wrapping Libraries

A framework such as pyOpenCL/pyCUDA[33] provides tools for writing GPU kernels directly in Python. The user writes OpenCL[47] or CUDA[50] specific kernels as text strings in Python, which simplifies the utilization of OpenCL or CUDA compatible GPUs. This strategy saves the programmer the trouble of writing some boilerplate code, while sacrificing some control and flexibility.

Code annotation

Some projects enable the use of GPGPU's via code annotations. The Copperhead[14] project relies on Python decorators, when compiling and executing a restricted subset of Python through CUDA. Because of the Bohrium runtime system, our GPU backend does not require any modifications to the Python code.

Array API

Python libraries such as CUDAMat[45] and Gnumpy[64] provide an API similar to NumPy for utilizing GPUs. The API of Gnumpy is almost identical with the API of NumPy. However, Gnumpy does not support arbitrary slicing when aliasing arrays.

Chapter 3

Bohrium

Bohrium is a runtime-system for mapping vector operations onto a number of different hardware platforms, from simple multi-core CPU systems to clusters and GPU enabled systems. In order to make efficient choices Bohrium is implemented as a virtual machine which makes runtime decisions.

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. A high-productivity language that allows rapid prototyping and still enables efficient utilization of a broad range of architectures is would be preferable. There exist high-productivity language and libraries which automatically utilize parallel architectures [37, 62, 49]. 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.

To provide a high productivity environment for the end user, the Bohrium system provides a number of language bridges. The language bridges attempt to integrate naturally into the host language. As the name implies, they provide a bridge from the programming language to the Bohrium runtime system. The overall programming method is known as array (or vector) programming, and is similar to the programming model found in Matlab [3] and NumPy [52].

In the current implementation of Bohrium, we provide language bridges

for C++, Python, and the Common Intermediate Language (CIL), also known as .Net. While these are the only three bridges implemented, they show that the array programming approach can be applied to different classes of programming languages, including: static and dynamic typed, compiled, interpreted, procedural, and functional. All implementations are provided as libraries, so that they may be used without customized tool-chains, compilers, or special runtime environments. Both the Python and CIL implementations include fallback implementations, which allows successful execution, even in the case where Bohrium is not present on the system. This enables the programmer the possibility to develop and verify applications in an environment the user is comfortable with, and then later add the option of efficient execution.

The Bohrium Python bridge [35] mimics the NumPy libraries and uses NumPy for fallback execution, so any existing NumPy program can be executed with Bohrium simply by changing the import statement to refer to Bohrium. The C++ bridge uses templates and operator overloads to provide an elegant interface to vectors and matrices. The CIL implementation uses the NumCIL [58] library as a front-end and provides simple access to vector and matrix operations from any of the CIL languages, such as C#, F#, IronPython and Visual Basic.

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}.

3.1 Design

Bohrium performs data-centric optimizations on vector operations, which can be viewed as akin to selective optimizations, in the respect that it does *not* optimize the program as a whole. This ensures a less intrusive user experience when using the Bohrium for interpreted languages like Python/NumPy. Where all arrays are by default handled by Bohrium. 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. We envision that this approach could also be used for R and other languages.

The Bohrium system consists of a number of runtime components, see Figure 3.1

Bridge The bridge translates the high level host language into Bohrium byte code. The bridge can either map an existing language or library into Bohrium bytecode as is the case with the NumPy bridge. In this

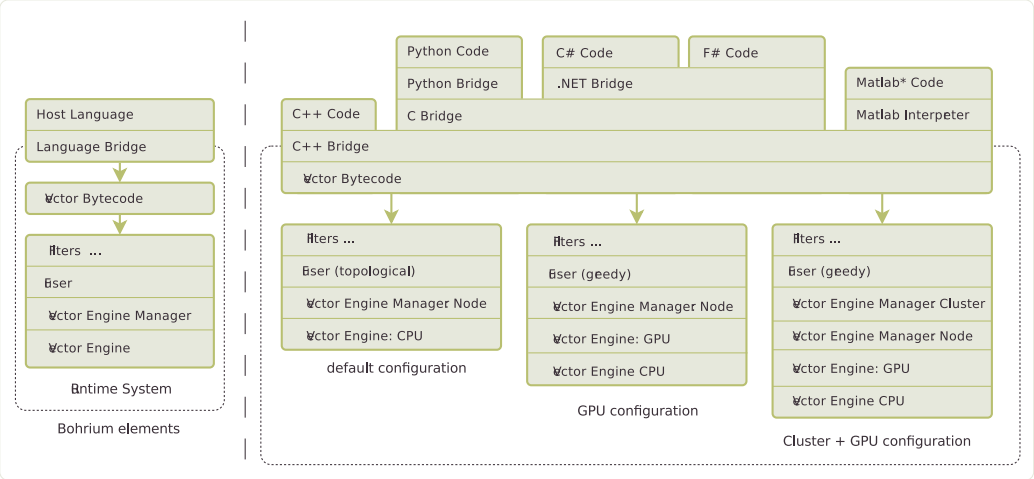


Figure 3.1: Bohrium Component Overview

case the use of Bohrium is completely transparent to the user. Another possibility is to implement a library which exposes the multidimensional array constructs and operations in a natural way to the programmer. The C, C++ and CIL bridges are examples of this approach. The main difference being that it is obvious to the programmer that he/she is using the Bohrium system directly via the implemented library.

In the MiniMatlab interpreter a complete, though small, language is implemented. This is an example of how Bohrium may be used by implementing a Domain Specific Language (DSL). A subset of the Matlab language was chosen so a new language did not have to be invented, but a new language for a specific set of tasks is a likely scenario for a DSL implementation of a Bohrium bridge.

Even with the compiled languages the Bohrium bytecode is generated at runtime. This allows for lazy evaluation of the generated code, which enables the rest of the Bohrium system to implement latency hiding, streaming, and other performance optimizing strategies.

Vector Engine *Manager* The Vector Engine Manager (VEM) is named so for historical reasons. As the original design contained a bridge, a VEM, and one or more Vector engines. The current design is more flexible, and as a result a VEM may manage any component below the language bridge, even other VEMs. The VEM's role is to manage the data location and ownership of arrays. It also manages the distribution of computing jobs between potentially several Vector Engines.

Vector Engine The Vector Engines (VE) are the components in Bohrium which execute the calculations described by the vector bytecodes it receives. The vector engines are architecture-specific.

Fuser Originally it was up to the vector engines, specifically the GPU vector engine, to combine multiple Bohrium vector bytecodes into suitable kernels. However we realized that it would be possible to devise a strategy for combining the vector bytecodes into kernels that are beneficial and well suited for a wide variety of vector engines and hardware.

The fuser combines, or fuses, multiple array operations into a single *kernel* of operations. The main benefit of which is temp array elimination, and ensuring that input and output arrays are read and written as few times as possible. Limiting the stress in the memory bandwidth of the system. Since the fuser is now a component it is possible to test and compare different strategies for kernel fusion.

Filter The main idea behind introducing filters into Bohrium is a means to bytecode transformation, but other uses have proven useful. The reason to introduce a means to bytecode transformation is twofold. One, to keep the vector engines simpler. The vector engines are already very complex, but at least they can be kept conceptually simple: execute the calculations described by the vector bytecode. With filters some optimizations and restrictions can be implemented here. The second benefit is that many optimizations are applicable across vector engines. This type of optimization can generally be expressed ad bytecode transformations.

3.1.1 Vector Byte-code

The Bohrium project builds upon a common, vector based, byte-code language. The Bohrium byte-code supports operations on multidimensional arrays, which can be sliced into sub-arrays, broadcast into higher dimensions, or viewed in other exotic ways. High level languages construct and operations are translated into this byte-code, that in turn will be executed by the different Bohrium vector engines. Every bytecode contains information on how the data of each array operand should be accessed (*viewed*). Each *view* consists of a data type, number of dimensions, an offset, and number of elements and a stride[1] for each dimension. See Figure 3.2 for a graphical representation of a Bohrium view.

[1]Number of base elements to skip ahead, to access the next data element in the given view, may be negative.

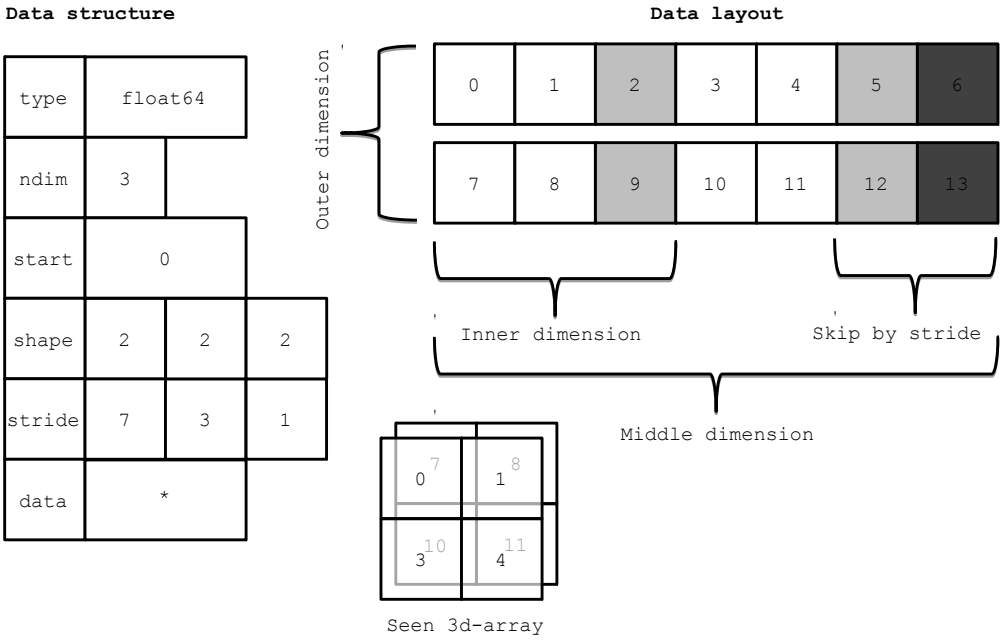


Figure 3.2: Descriptor for n-dimensional array and corresponding interpretation

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. The arrays can also be shaped into multi-dimensional arrays, to avoid excessive memory copying. 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 3.2 shows how the shape is implemented and how the data is projected.

Figure 3.4 illustrates a list of vector bytecode that the NumPy Bridge will generate when executing one of the iterations in the Python/NumPy implementation of the heat equation solver (Fig. 3.3). The example demonstrates the nearly one-to-one mapping from the NumPy vector operations to the Bohrium vector bytecode. 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.

The aim is to have a vector bytecode that support data parallelism im-

```
1  import bohrium as numpy
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*(center+north+south+east+west)
11     delta = numpy.sum(numpy.abs(tmp-center))
12     center[:] = tmp
```

Figure 3.3: 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.

plicitly 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 the "Single Instruction, Multiple Data" (SIMD) paradigm and the VEM through the "Single Program, Multiple Data" (SPMD) paradigm.

In the following section, we will go through the four types of vector bytecodes in Bohrium.

Element-wise

Element-wise bytecodes perform 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, Bohrium only allows 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.

Reduction

Reduction bytecodes reduce an input dimension using a binary operator. Again, Bohrium does 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

```
1  ADD t1, grid[center], grid[north]
2  ADD t2, t1, grid[south]
3  FREE t1
4  DISCARD t1
5  ADD t3, t2, grid[east]
6  FREE t2
7  DISCARD t2
8  ADD t4, t3, grid[west]
9  FREE t3
10 DISCARD t3
11 MUL work, const(0.2), t4
12 FREE t4
13 DISCARD t4
14 MINUS t5, work, grid[center]
15 ABS t6, t5
16 FREE t5
17 DISCARD t5
18 ADD_REDUCE t7, t6
19 FREE t6
20 DISCARD t6
21 ADD_REDUCE delta, t7
22 FREE t7
23 DISCARD t7
24 IDENTITY grid[center], work
25 FREE work
26 DISCARD work
27 SYNC delta
```

Figure 3.4: Bytecode generated in each iteration of the Python/NumPy implementation of the heat equation solver (Fig. 3.3). For convenience, the arrays and views are given readable names. Views are annotate in "[]". Note that the `SYNC` instruction at line 27 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. 3.3, line `9`).

none of them is stateless, the reductions are all straightforward to execute in parallel because of the non-overlap and associative properties.

Generators

Generators are byte codes which generate data in some structured way. This can simply be the assignment of a single value to every element of an array. The `BH_RANGE` byte code enables the assignment of an equally spaced range of values to an array. The values can the be manipulated into other structured distributions of values like for example values evenly spaced on a logarithmic scale.

Bohrium also implements a pseudo random number generator via the Random123[55] library. This enables Bohrium to generate the same sequence of pseudo-random numbers on different hardware. Separate portions of the same sequence can even be generated by different engines with out the need for communication.

Data Management

Data Management bytecodes determine the data ownership of arrays and consist 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 memory allocations and copies.

Extension methods

The bulk of a Bohrium execution consists mainly of the above three types of bytecode. 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 a fourth bytecode type: extension methods. Bohrium imposes no restrictions to this generic operation; the extension writer has total freedom. However, Bohrium does not require 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[11].

3.1.2 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.

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.

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 [42, 70, 44, 66]. 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. 3.3) 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 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.

Due to the nature of Bohrium, being a runtime system, and Python being an interpreted language, the Python bridge will need to synchronize with Bohrium every time the `while` statement in figure 3.3 line 9 is evaluated i.e. for each loop iteration. This means that the body of the loop (Fig. 3.3 l. 10 – 12) resulting in the bytecode in figure 3.4 will be sent repeatedly by the bridge. Every time as a separate batch. In fact, Bohrium does not provide any control instructions at all.

3.1.3 Vector Engine

The Vector Engine (VE) is the only component that does computations, specified by the user application. It has to execute the instructions it receives in a valid order; that comply with the dependencies between instructions, i.e.

program order. Furthermore, it has to ensure that its parent VEM has access to the results as governed by the Data Management bytecodes.

The VE is the focus of this paper – in order to utilize the GPU, we implement a GPU-VE that execute Bohrium bytecode on the GPU. In the following section, we will describe our GPU-VE in detail.

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[31]. 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[43].

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.

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.

3.1.4 Example

Figure 3.4 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. 3.3). 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.

3.1.5 Vector Engine Manager

In order to couple the Bridge and the hardware specific Vector Engine, Bohrium uses a third component: the Vector Engine Manager (VEM). The

VEM is responsible for one memory address space in the hardware configuration. In our configuration, we only use a single machine (Node-VEM) thus the role of the VEM component is insignificant. The Node-VEM will simply forward all instruction from its parent to its child components.

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.

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[65] with static data decomposition and no communication latency hiding. We know from previous work than such optimizations are possible[39].

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.

```
# 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 3.5: 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`.

3.1.6 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. 3.5). 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.

3.2 The Bohrium NumPy Bridge

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[67].

Numerical Python (NumPy[51]) is the de-facto standard for scientific applications written in Python. 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[42], MATLAB[70], Armadillo[56], and Blitz++[68].

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`".

The Python/NumPy[51] support in Bohrium consists of an extension of NumPy version 1.6, which seamlessly implements a new array back-end that inherits the manipulation features, such as *view*, *slice*, *reshape*, *offset*, and *stride*. As a result, the user only needs to modify the import statement of NumPy in order to utilize the GPU back-end.

Chapter 4

The Bohrium GPU Vector Engine

The Bohrium vector bytecode with its SIMD properties is a good match for the GPU. In this section, we will explain how we have chosen to convert the vector bytecode into code that executes on the GPU.

In order to implement the GPU-VE, we use the OpenCL[47] framework. There are many possible choices for an implementation framework. One of the main goals of the Bohrium project is to deliver high performance. For this reason, we want a framework that allows for fairly low-level control of the hardware and relatively close mapping between the code we generate and the operations the hardware executes. The other candidates for this level of control are CUDA[50] and LLVM[40]. While LLVM could be a good choice for the code generated for the GPU kernels, LLVM depends on CUDA or OpenCL, to drive the GPU i.e. compiling and loading kernels and moving data to and from GPU memory.

The obvious alternative is the CUDA framework though it has a couple of drawbacks compared to OpenCL. Firstly, CUDA imposes a vendor lock in since it only supports devices by NVIDIA. We would prefer our solution to be vendor independent. Secondly, CUDA uses the pseudo-assembly language, PTX, which is more complex, requiring address calculations, and explicit load/store operations and register management. Alternatively, it is possible to compile C/C++ code to PTX using the external compiler, nvcc, and pay the relatively expensive cost of a system call. CUDA allegedly has a performance advantage (on NVIDIA devices) but studies[25] have shown that OpenCL achieves comparable performance under fair comparison. On top of this, we expect that the simplicity of the GPU kernels we generate makes the effect of advanced optimizations negligible.

4.1 JIT Compilation

The element-wise operations (Sec. 3.1.1) are considered the basic operations of Bohrium, as these are in effect vector operations. They are also the most common bytecodes received by the vector engine. A notable property of these vector operations is that all input and output operands have the same size even though the underlying base array might have different sizes. The Bridge will enforce this property through dimension duplications and/or vector slicing. Because of this property, mapping the element-wise operations to the GPU is straightforward through the SIMD execution model. Since the SIMD execution model is often recommended as an implementation model for the SIMT architecture of the GPU[50], it seems like a simple and logical choice. We map one thread to each output data element where each thread is responsible for fetching the required input data, doing the required calculations, and saving the result to the correct index in the output data.

Detecting kernel boundaries

A Bohrium bytecode represents a relatively simple operation, addition, subtraction, square root, less than, etc. The execution time for such an operation is very small compared to the time required to fetch the data elements from global memory, and writing the result back to global memory. Even when spawning millions of threads, and thus implementing latency hiding on the GPU, fetching and writing data is going to be the dominant time factor. The result is that single operation *microkernels* is not a viable solution for the GPU vector engine.

We need a scheme for collecting multiple operations into compound kernels for the GPU to run. Finding the optimal execution order for all the instructions in a batch is an NP-hard problem[26]. However, there exist different heuristic methods for finding *good* execution orders, while obeying the inter-instruction dependencies[32]. It is, however, out of the scope of this thesis to do a performance analysis of these methods for our given setup. We have chosen to implement a simple scheme that guarantees a legal ordering of instruction since it does not reorder the instructions.

This scheme keeps adding instructions, in order, from the batch to the current kernel for as long as the instructions are compatible. When the scheme encounters a non-compatible instruction, it schedules the current kernel under construction for execution. Subsequently, the scheme initiates a new kernel with the non-compatible instruction as the first instruction. The scheme repeats this process until all instruction has been scheduled and executed. The criteria for a compatible instruction are:

1. The instruction has to be one of the element-wise operations. If it is a *reduction* or an *extension method* the current kernel will be executed before executing the reduction- or extension method instruction.

2. The shape and size of the data elements that are accessed must be the same as that of the kernel.

3. Adding the instruction can not create a data hazard. A data hazard is created if the new instruction reads a data object, that is also written in the same kernel, and the access patterns are not completely aligned. This is similar to loop fusion strategies implemented by compilers.

When the rules, outlined above, are applied to the bytecode example shown in figure 3.4, four kernel boundaries will be found. Resulting in four kernels. Of these four kernels only three of them will be executed on the GPU. The bytecode shown in figure 3.4 is produced by the loop body in figure 3.3, line 9 – 12, and is repeated for as long as `delta > epsilon`, although every iteration will produce a separate bytecode batch as explained in sec. 3.1.2.

At the beginning of the batch, the current kernel is empty thus the first operation (the `ADD` in line 1) is added. The first operation that requires the insertion of a boundary and triggers kernel compilation and execution is the `ADD_REDUCE`-instruction in line 18, because this is not an element-wise operation. Reduction is handled separately, and it is compiled into its own kernel. The second reduction will also be handled as a separate kernel, except, it is not executed on the GPU. Rather the result of the previous reduction is transferred to the host memory, and the second reduction is executed on the host CPU. This is done for efficiency, as the second reduction is too small to utilize the GPU. Finally, the `IDENTITY`-operation in line 24, to copy the intermediate result back to the main grid, is compiled into its own kernel due the fact that the batch of instructions ends after the `SYNC`-instruction in line 27.

It is worth noticing that if the while loop from figure 3.3 was a for-loop instead, or in another way did not depend on the calculations within the loop, the batches would be concatenated into one, i.e. the bytecode in figure 3.4 would repeat with line 1 again after line 27. If this were the case, the `IDENTITY`-operation in line, 24 would still end up in its own kernel, because trying to add the following `ADD`-operation would break the third rule, listed above. As the input `grid[north]` is unaligned with the output `grid[center]` of the `IDENTITY`-operation.

Of the *data management* instructions, only the `DISCARD` instruction potentially effects the kernel. If it is one of the output arrays of one of the instructions of the current kernel, that means that the result is not used

outside the kernel, and; therefore it does not need to be saved for later use. This saves the memory write of one data element per thread.

Source code generation

Before a kernel can be scheduled for execution, it needs to be translated into an OpenCL C kernel function. The bytecode sequence shown in figure 3.4 will create three GPU kernels. Since the bytecode sequence is repeated for each iteration of the while loop of the Python/NumPy code from figure 3.3, the three kernels could be generated for each iteration. While generation of the OpenCL source code is relatively inexpensive, calling the OpenCL compiler and translating the source code into a hardware specific kernel comes at a cost that has a significant impact on the total execution time of the program. To minimize the time spent on compilation, the GPU-VE will cache all compiled kernels for the lifetime of the program. The recurrence of a byte-code sequence is registered, and the compiled kernel is reused by simply calling it again, possibly with new parameters.

Every single bytecode is trivially translated into a single line of OpenCL C code, since every unique array-view just needs to be given a unique name. This way line 1 – 15 from figure 3.4 is translated into the calculation body an OpenCL function kernel: line 18 – 31 of figure 4.1, which is then prepended with the code for loading the needed data (line 13 – 17), and appended with code for saving the results (line 32 – 33). Notice that t1 ... t5 are not saved, as they are not needed outside the kernel. Code to make sure surplus threads do not write to unintentional addresses is inserted (line 8 – 9 & 11 – 12). Finally, the function header, with call arguments consisting of input and output parameters (line 1 – 5) are added to the code block. The kernel is then compiled and executed. The kernel code for the reduction is included in figure 4.2. Figure 4.3 shows the code for the copy-back-function generated by the `IDENTITY`-instruction in line 24 of figure 3.4.

4.2 Data Management

The remaining *data management* instructions, `FREE` and `SYNC`, do not effect the content of the kernel. A `SYNC`-instruction may force the execution of a kernel, if the array in question is being written by the current kernel, and the data is the copied from the GPU to the main memory for availability to the rest of the system.

A `FREE`-instruction only concerns the main memory of the system thus; it is just executed when encountered.

```
1  __kernel void kernelb981208bc41e203a(
2                          __global float* grid
3                        , __global float* work
4                        , const float s0
5                        , __global float* t6)
6  {
7      const size_t gidy = get_global_id(1);
8      if (gidy >= 1000)
9          return;
10     const size_t gidx = get_global_id(0);
11     if (gidx >= 1000)
12         return;
13     float center = grid[gidy*1002 + gidx*1 + 1003];
14     float north = grid[gidy*1002 + gidx*1 + 1];
15     float south = grid[gidy*1002 + gidx*1 + 1004];
16     float east = grid[gidy*1002 + gidx*1 + 1002];
17     float west = grid[gidy*1002 + gidx*1 + 2005];
18     float t1;
19     t1 = center + north;
20     float t2;
21     t2 = t1 + south;
22     float t3;
23     t3 = t2 + east;
24     float t4;
25     t4 = t3 + west;
26     float work_;
27     work_ = s0 * t4;
28     float t5;
29     t5 = work_ - center;
30     float t6_;
31     t6_ = fabs(t5);
32     work[gidy*1000 + gidx*1 + 0] = work_;
33     t6[gidy*1000 + gidx*1 + 0] = t6_;
34 }
```

Figure 4.1: First of three kernels generated by the GPU-VE from the bytecode shown in Fig. 3.4. The kernel inplements line 1 – 17 of the bytecodes.

```
1  __kernel void reduce6020b3d120d0ec9a(
2                          __global float* t7
3                        , __global float* t6)
4  {
5          const size_t gidx = get_global_id(0);
6          if (gidx >= 1000)
7                  return;
8          size_t element = gidx*1 + 0;
9          float accu = t6[element];
10         for (int i = 1; i < 1000; ++i)
11         {
12                 element += 1000;
13                 accu = accu + t6[element];
14         }
15         t7[gidx*1 + 0] = accu;
16 }
```

Figure 4.2: Source code for the reduction kernel produced by line 18 of the bytecode in Fig. 3.4

```
1  __kernel void kernela016730fd6e00085(
2                          __global float* grid
3                        , __global float* work)
4  {
5          const size_t gidy = get_global_id(1);
6          if (gidy >= 1000)
7                  return;
8          const size_t gidx = get_global_id(0);
9          if (gidx >= 1000)
10                 return;
11         float work_ = work[gidy*1000 + gidx*1 + 0];
12         float center;
13         center = work_;
14         grid[gidy*1002 + gidx*1 + 1003] = center;
15 }
```

Figure 4.3: Source code for the copy back kernel produced by line 24 of the bytecode in Fig. 3.4

There is no instruction for signaling that data needs to be copied to the GPU from main memory. The data will simply be copied to the GPU as it is needed.

4.3 Code Specialization

The Bohrium operator access patterns are prime candidates for code specialization. The GPU vector engine of Bohrium translates the vector byte code into OpenCL bases GPU kernels at run-time. We can choose to make the generated kernels more general by including the access patterns as parameters, or specialize the kernels by including the access patterns as literals. It is our expectation that specialized kernels will perform better than non-specialized due to the fact the compiler will have more information to work with.

4.3.1 Limitations

Bohrium byte-code does not support loops or control structures. Bohrium relies on the host language for these programming constructs. When iterating through a loop the same byte-code sequence will simply be repeated. Though operands and access patterns may change with each iteration. It is important for the vector engine to detect these repetitions since JIT compiling OpenCL source code at run-time is a time consuming task, compared to the execution time of the generated kernel. That way JIT compiling cost is amortized with repeated calls to same OpenCL kernel. We need to ensure that our specialization of the generated GPU kernels do not prohibit them from being reused. If we consider the loop body of LU decomposition, as it may implemented in Python/NumPy, as shown in Figure 4.4. The *views* on both `l` and `u` change with each iteration. When translated into Bohrium bytecode the specific view attributes are concretized, i.e. no longer symbolic.

Bohrium is designed to support interpreted languages. As a result of this design choice we may need to execute calculations based on very limited knowledge of the general structure of the host program — since we have no knowledge of future calculations. This needs to be taken into account when devising a strategy for code specialization. Consider the Jacobi stencil code in Figure 4.5: The Python interpreter evaluates the boolean clause of the while statement in line 3, thus the value of `delta` is returned to the host-language bridge together with the control for each iteration of the loop. So from Bohriums point of view we do not know that the views do not change. At least not for the first calculation of `delta`. With each iteration we could

```
def lu(a):
    u = a.copy()
    l = numpy.zeros_like(a)
    numpy.diagonal(l)[:] = 1.0
    for c in xrange(1,u.shape[0]):
        l[c:,c-1]  = u[c:,c-1] / u[c-1,c-1:c]
        u[c:,c-1:] = u[c:,c-1:] -
                     l[c:,c-1][:,None] * u[c-1,c-1:]
    return (l,u)
```

Figure 4.4: Python/NumPy implementation of LU decomposition.

assume static views with higher and higher confidence.

```
def jacobi_2d(grid, epsilon=0.005):
    delta  = epsilon + 1
    while delta > epsilon:
        work = (grid[1:-1,1:-1] + grid[0:-2,1:-1] +
                grid[1:-1,2:] + grid[1:-1,0:-2] +
                grid[2:,1:-1]) * 0.2
        delta = numpy.sum(numpy.absolute(work -
                                         grid[1:-1,1:-1]))
        grid[1:-1,1:-1] = work
    return grid
```

Figure 4.5: Python/NumPy implementation of 2D-Jacobi stencil.

4.3.2 Strategy

There are two extremes of code specialization, focusing on the views or access patterns of Bohrium. The first is of course to specialize everything found in the Bohrium view descriptor, i.e. use only literals. The other extreme is to parametrisize the generated kernels with the Bohrium view descriptor, i.e. use only function variables. Choosing no specialization gives us the best chance of code reusability, resulting in less time spent on the compiler compiling code if the indexes change, but *perhaps* more time calculatng data indexes during execution. Symmetrically, the effect of full specialization is the reverse: Giving the compiler as much information as possible to work with, hopefully enabling better optimized code. The optimal solution would be to include exactly those values that do not change over the lifetime of the program as literals. In our scenario, however, we do not have the global knowledge required for implementing this optimal solution. We may, depend-

ing on the application, actually have very limited knowledge — it would be desirable with a strategy that is both simple, and works well in most situations.

The strategy we have chosen for this work, testing the benefits of code specialization, if any, is as follows:

> The first time a unique set of byte codes, which constitute a GPU kernel, are encountered: We compile both a completely specialized kernel and a generalized kernel, i.e. the two extremes. The compilations are done separately and asynchronously. We enqueue the fully specialized kernel for execution. As we expected this to be the most efficient version of the kernel. We also expected the compilation time for both kernels to be largely the same.
>
> Upon receiving the same pattern of byte codes, constituting a kernel, again, we know that we have a matching kernel i.e. the generalized, parametrizied, kernel. The specialized version of the kernel is tested for fitness, i.e. the access pattern is the same as when the code was generated. If the specialized kernel matches it is scheduled for execution, otherwise the generalized kernel is scheduled.

Most of the code for the generated kernels are the same, independent of specialization, since the core functionality is the same. So we found it most convinient to generate one source code, and use the c preprocessors `#define` directives to generate the different GPU kernels. The Python code shown in Figure 4.5 generates two GPU kernels. The main body of the generated OpenCL source code is shown in Figure 4.6, and the define and size parameter parts are shown in Figure 4.7.

```
1  #pragma OPENCL EXTENSION cl_khr_fp64 : enable
2  #ifdef FIXED_SIZE
3  // defines go here
4  #endif
5  __kernel __attribute__((work_group_size_hint(64, 4, 1))) void
6  #ifndef FIXED_SIZE
7  kernel429d67e832590722
8  #else
9  kernel429d67e832590722_
10 #endif
11 (
12           __global double* a1
13         , __global double* a5
14         , __global double* a7
15         , const double s0
16 #ifndef FIXED_SIZE
17 // size parameters go here
18 #endif
19 )
20
21 {
22         const size_t gidx = get_global_id(0);
23         if (gidx >= ds0)
24                 return;
25         const size_t gidy = get_global_id(1);
26         if (gidy >= ds1)
27                 return;
28         double v1 = a1[gidy*v1s2 + gidx*v1s1 + v1s0];
29         double v2 = a1[gidy*v2s2 + gidx*v2s1 + v2s0];
30         double v4 = a1[gidy*v4s2 + gidx*v4s1 + v4s0];
31         double v6 = a1[gidy*v6s2 + gidx*v6s1 + v6s0];
32         double v8 = a1[gidy*v8s2 + gidx*v8s1 + v8s0];
33         double v0;
34         v0 = v1 + v2;
35         double v3;
36         v3 = v0 + v4;
37         double v5;
38         v5 = v3 + v6;
39         double v7;
40         v7 = v5 + v8;
41         double v9;
42         v9 = v7 * s0;
43         double v10;
44         v10 = v9 - v1;
45         double v11;
46         v11 = fabs(v10);
47         a5[gidy*v9s2 + gidx*v9s1 + v9s0] = v9;
48         a7[gidy*v11s2 + gidx*v11s1 + v11s0] = v11;
49 }
```

Figure 4.6: Generated OpenCL kernel created by 2D-Jacobi stencil.

```
#define ds1 1998
#define ds0 1998
#define v1s0 2001
#define v1s2 2000
#define v1s1 1
#define v2s0 1
#define v2s2 2000
#define v2s1 1
#define v4s0 2002
#define v4s2 2000
#define v4s1 1
#define v6s0 2000
#define v6s2 2000
#define v6s1 1
#define v8s0 4001
#define v8s2 2000
#define v8s1 1
#define v9s0 0
#define v9s2 1998
#define v9s1 1
#define v11s0 0
#define v11s2 1998
#define v11s1 1
```

```
, const int ds1
, const int ds0
, const int v1s0
, const int v1s2
, const int v1s1
, const int v2s0
, const int v2s2
, const int v2s1
, const int v4s0
, const int v4s2
, const int v4s1
, const int v6s0
, const int v6s2
, const int v6s1
, const int v8s0
, const int v8s2
, const int v8s1
, const int v9s0
, const int v9s2
, const int v9s1
, const int v11s0
, const int v11s2
, const int v11s1
```

Figure 4.7: The define and parameter declaration part of the generated code in figure 4.6.

Chapter 5

Ongoing Work

At the time of writing this theses the work described in the "Fusion of Array Operations at Runtime" (see Section 7.7) has been implementer in the Bohrium system as the fuser component. The paper has, however not been published yet.

The current implementation of the GPU vector engine supports these architecture independent kernels. That means that the GPU vector engine is able to generate more complex GPU-kernels, containing more instructions per kernel. This also means that the GPU vector engine now supports combining generators, element-wise operations, and reductions in the same GPU-kernel. All of this leads to better performance for the GPU vector engine. How much of a performance gain is dependent on the benchmark. This work is unfortunately undocumented at this time.

Chapter 6

Conclusion

In my PhD work I show that it is possible to run unmodified Python/NumPy code on modern GPUs. This is done by using the Bohrium runtime system to translate the NumPy array operations into an array based bytecode sequence. Executing these byte-codes on two GPUs from different vendors shows great performance gains.

Scientist working with computer simulations should be allowed to focus on their field of research and not spend excessive amounts of time learning exotic programming models and languages. We have with the Bohrium achieved very promising results by starting out with a relatively simple approach. This has the leas to more specialized methods as I have shown with the work done with both specialized-, and parametrizied- kernels. Both have their benefits and recognizable use cases. We achieved clear performance benefits without any significant negative impact on overall application performance. Even in the cases where we were not able to gain any performance boost by specialization the added cost, for kernel generation and extra bookkeeping, is minimal.

Many of the lessons learned developing and optimizing the Bohrium GPU vector engine has proven to be valuable in a broader perspective. Which has made it possible to generalize to benefit the complete Bohrium project.

Chapter 7

Publications

7.1 cphVB: A Scalable Virtual Machine for Vectorized Applications

Mads Ruben Burgdorff Kristensen, Simon Andreas Frimann Lund, Troels Blum, Brian Vinter.
Proceedings of The 11th Python In Science Conference (SciPy'12).

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 outperforms 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

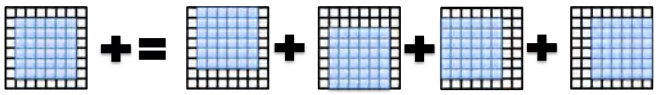


Fig. 1: *Matrix expression of a simple 5-point stencil computation example. See line eight in the code example, for the Python expression.*

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.

```
1 center = full[1:-1, 1:-1]
2 up     = full[0:-2, 1:-1]
3 down   = full[2:  , 1:-1]
4 left   = full[1:-1, 0:-2]
5 right  = full[1:-1, 2:  ]
6 while epsilon < delta:
7     work[:] = center
8     work += 0.2 * (up+down+left+right)
9     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:

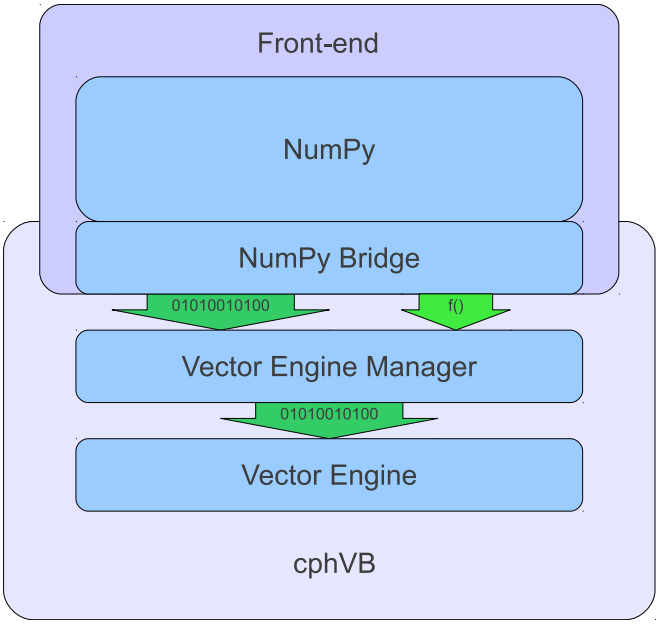


Fig. 2: *cphVB design idea.*

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).
- **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

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

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

§. NumPy version 1.6.1.

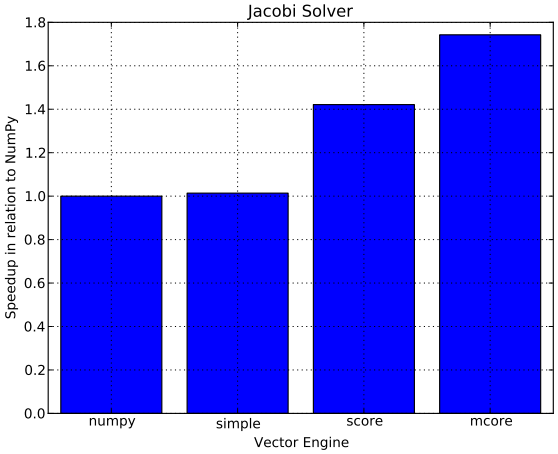


Fig. 3: *Relative speedup of the Jacobi Method. The job consists of a vector with* 7168*x*7168 *elements using four iterations.*

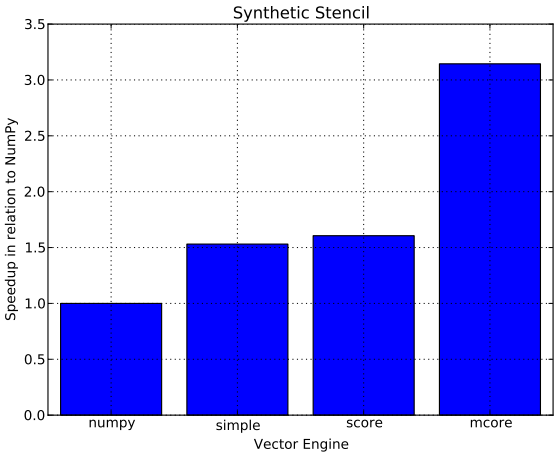


Fig. 6: *Relative speedup of the synthetic stencil code. The job consists of vector with* 10240*x*1024 *elements that simulate 10 time steps.*

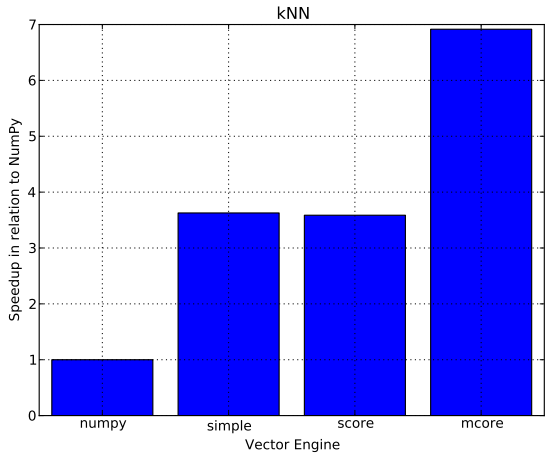


Fig. 4: *Relative speedup of the k Nearest Neighbor search. The job consists of 10.000 elements and the query set also consists of 1K elements.*

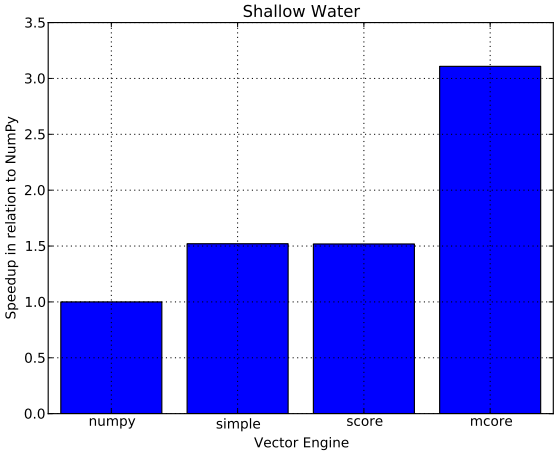


Fig. 5: *Relative speedup of the Shallow Water Equation. The job consists of 10.000 grid points that simulate 120 time steps.*

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/CUDACProgrammingGuide.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.

7.2 Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster

Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter.
4th Workshop on Python for High Performance and Scientific Computing. (PyHPC 2013)

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 [9]. 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[10] 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 [11], 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 [12] 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) [13] 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 [14] 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 [15] 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.

```
1  import bohrium as numpy
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*(center+north+south+east+west)
11     delta = numpy.sum(numpy.abs(tmp-center))
12     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], [16], [17]. 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

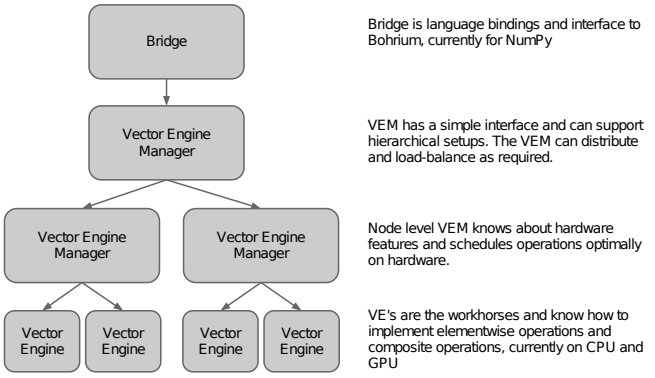


Fig. 2: Bohrium Overview

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`.

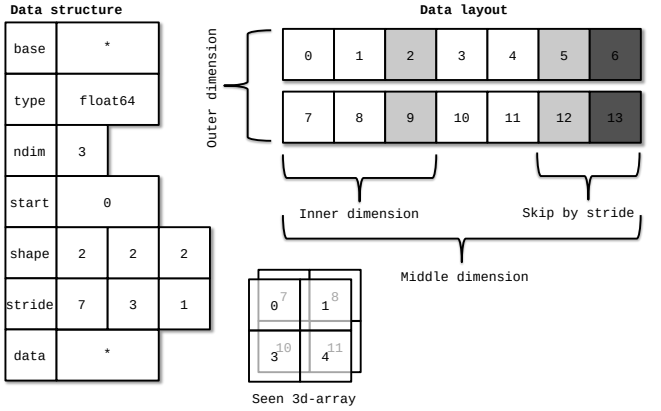


Fig. 4: Descriptor for n-dimensional array and corresponding interpretation

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[18].

[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[19] with static data decomposition and no communication latency hiding. We know from previous work than such optimizations are possible[20].

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[21]. 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[22].

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.

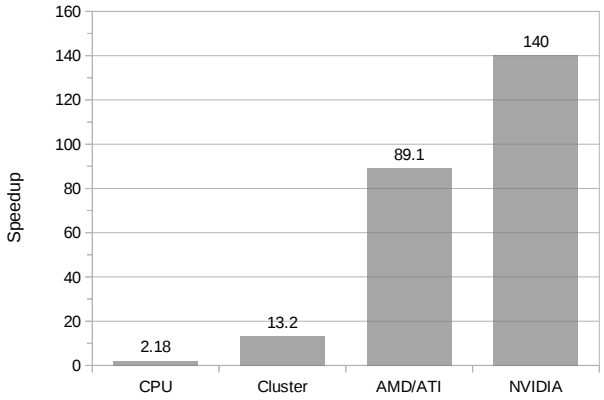


Fig. 6: Relative speedup of the Shallow Water application. For the CPU and Cluster, the application simulates a 2D domain with $25k^2$ value points in 10 iterations. For the GPUs, it is a 2k $\times$ 4k domain in 100 iterations.

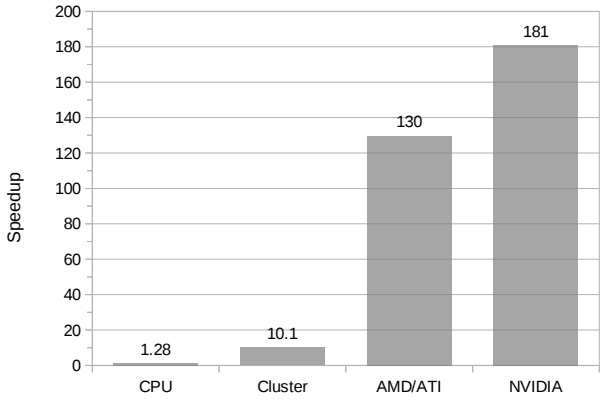


Fig. 7: Relative speedup of the Black Scholes application. For the CPU and Cluster, the application generates 10m element arrays using 10 iterations. For the GPUs, it generates 32m element arrays using 50 iterations.

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 [23].

Black Scholes The Black-Scholes pricing model is a partial differential equation, which is used in finance for calculating price variations over time[24]. 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

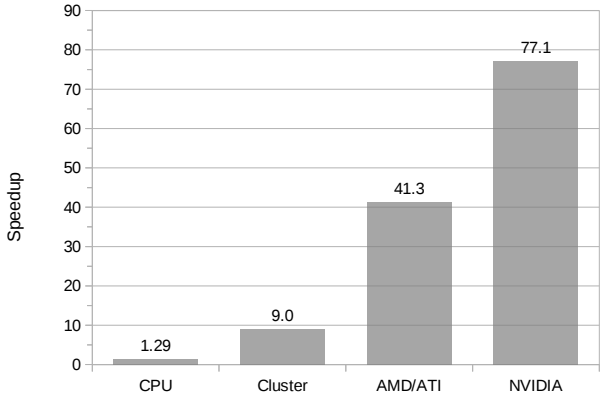


Fig. 8: Relative speedup of the N-Body application. For the CPU and Cluster, the application simulates 25k bodies in 10 iterations. For the GPUs, it is 1600 bodies and 50 iterations.

(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

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[25], [26] 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[27].

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] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] R. Andersen and B. Vinter, "The scientific byte code virtual machine," in *GCA'08*, 2008, pp. 175–181.

[16] 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

[17] 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.

[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] 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.

[21] 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.

[22] 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.

[23] J. Burkardt, "Shallow water equations," people.sc.fsu.edu/\~jburkardt/m_src/shallow_water_2d/, [Online; accessed March 2010].

[24] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *The journal of political economy*, pp. 637–654, 1973.

[25] 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.

[26] 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.

[27] 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.

7.3 Bohrium: a Virtual Machine Approach to Portable Parallelism

Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter.
28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014).

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 paper 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 compiled 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 criteria 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 multi-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 scientific community. 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[2]. 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], 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]also known as Microsoft .NET

[2]The implementation is open-source and available at www.bh107.org

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 [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 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 [10] 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 [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.

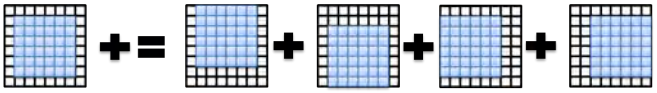


Fig. 1: A computation that makes use of views to implement a 5-point stencil.

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. An implementation that solves the heat equation iteratively using the Jacobi Method, we use 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

```
#include <bh/bh.hpp>
double solve(multi_array<double> grid, size_t epsilon)
{
  multi_array<double> center,north,south,east,west,tmp;
  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)];
  double delta = epsilon+1;
  while(delta > epsilon){
    tmp = 0.2*(center+north+east+west+south);
    delta = scalar(sum(abs(tmp-center)));
    center(tmp);
  }
}
```

Fig. 2: Bohrium C++ implementation of the heat equation solver. The `grid` is a two-dimensional Bohrium array and the `epsilon` is a regular C/C++ scalar.

```
using NumCIL.Double;
using R = NumCIL.Range;
double Solve(NdArray grid, double epsilon)
{
  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)];
  var delta  = epsilon+1;
  while(delta > epsilon){
    var tmp = 0.2*(center+north+east+west+south);
    delta = (tmp-center).Abs().Sum();
    center[R.All] = tmp;
  }
}
```

Fig. 3: NumCIL C# implementation of the heat equation solver. The `grid` is a two-dimensional NumCIL array and the `epsilon` is a CIL scalar.

improved by a single component and not in every language-bridge. A positive side-effect of avoiding expression-templates in the C++ bridge are better compile-time error-messages for the application programmer.

Figure 2 illustrates the heat equation solver implemented 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 extends 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

```
1  import bohrium as numpy
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*(center+north+south+east+west)
11     delta = numpy.sum(numpy.abs(tmp-center))
12     center[:] = tmp
```

Fig. 4: 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.

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 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, 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*. 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):

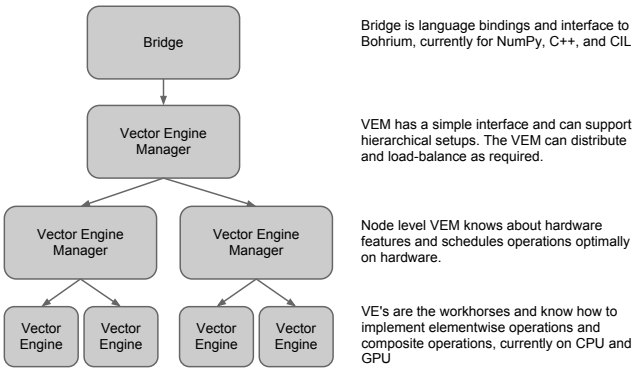


Fig. 5: Bohrium Overview

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 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.

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. 6). 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 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

```
# 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`.

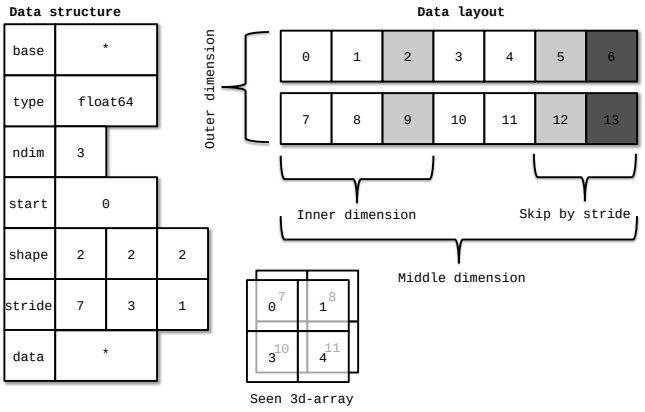


Fig. 7: Descriptor for n-dimensional vector and corresponding interpretation

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 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[3] and the VEM through SPMD[4].

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. 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 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[5]. 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 vectors, and consists of three different bytecodes. The synchronization bytecode orders a child component to place the vector data in the address space of its parent component. The free bytecode orders a child component to free the data of a given vector in the global address space. Finally, the discard operator that orders a child component to free the 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.

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[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

[3]Single Instruction, Multiple Data
[4]Single Program, Multiple Data

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

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 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 currently quite naïve; it uses the bulk-synchronous parallel model[19] with static data decomposition and no communication latency hiding. We know from previous work than such optimizations are possible[20].

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 vector 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 vector 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 vectors – 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 targets shared-memory multi-core CPU architectures through OpenMP. The CPU-VE is implemented as an in-order interpreter of vector 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 dimensionality of operands.

The CPU-VE just-in-time compiles each vector bytecode into individual computation kernels that consists of N nested loops where N is the dimensionionality of the operands. The current parallelization strategy simply applies `#pragma omp for` to the outer-most loop, which tells OpenMP to parallelize over the first dimension. This naïve approach requires that the size of the first dimension is greater than the number of CPU-cores to utilize. As a special case, the CPU-VE solves this limitation for operands with contiguous memory-access by collapsing the loops into a single loop.

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, vector programming involves the use of many temporary vectors and thus uses many consecutive equally sized memory allocations and de-allocations. In order to reduce the overhead associated with these memory allocations and de-allocations, we introduces a reusing scheme similar to a Victim Cache[21]. 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.

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 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 vector 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.

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 vector 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.

```
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. 8: Bytecode generated in each iteration of the Python/NumPy implementation of the heat equation solver (Fig. 4). 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 `while` condition (Fig. 4, line `9`).

F. Python/NumPy Example

Figure 8 illustrate 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. 4). 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 not be possible on the GPU due to the memory bandwidth bottleneck.

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. The implementation supports 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. Below we conduct 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]. We use this benchmark for studying the Python/NumPy performance in Bohrium where the Bohrium execution uses the Bohrium back-end and the baseline execution uses the native NumPy back-end.

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. In order to study the performance of Bohrium in C++, we compare an implementation that uses Bohrium with a baseline implementation that uses Blitz++. The two C++ implementations are very similar and both uses vector operations almost exclusively.

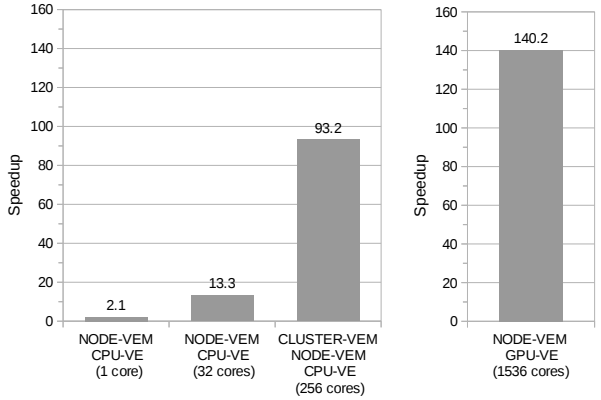
N-Body A Newtonian N-body simulation is one that studies how bodies, represented by a mass, a location, and a velocity, move in space according to the laws of Newtonian physics. We use a straightforward algorithm that computes all body-body interactions, $O(n^2)$, with collisions detection. It is a C# implementation that uses the NumCIL vector library[17]. We use this benchmark for studying the NumCIL performance in Bohrium where the Bohrium execution uses the Bohrium back-end and the baseline execution uses the native NumCIL back-end.

For the performance study, we use two different hardware installations: an installation of eight AMD cluster-nodes and an installation of a single AMD GPU-node (Table I & II).

For each scientific application kernel, we execute using four different Bohrium setups:

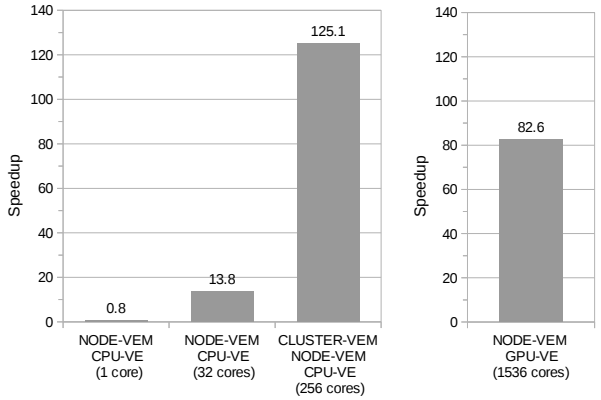
- A setup with the Node-VE and the CPU-VE where the execution utilize a single CPU-core on one of the AMD cluster-nodes.
- A setup with the Node-VE and the CPU-VE where the execution utilize all 32 CPU-cores on one of the AMD cluster-nodes through OpenMP.
- A setup with the Cluster-VEM, the Node-VE, and the CPU-VE where the execution utilize all 32 CPU-cores on each of the eight AMD cluster-nodes – 256 CPU-cores in total. The setup make use of the Hybrid Programming Model[24] where the Cluster-VEM spawns four MPI-processes per node and the CPU-VE in turn spawns eight OpenMP threads per MPI-process.
- A setup with the Node-VE and the GPU-VE where the execution utilize the GPU on the GPU-node.

Please note that none of the setups requires any change to the scientific applications. It is simply a matter of changing the Bohrium configuration ini-file (Fig. 6).



(a) (b)

Fig. 9: Relative speedup of the Shallow Water application. For the execution on the Cluster-node (a), the application simulates a 2D domain with $25k^2$ value points in 10 iterations. For the execution on the GPU-node (b), it is a $4k^2$ domain in 100 iterations.



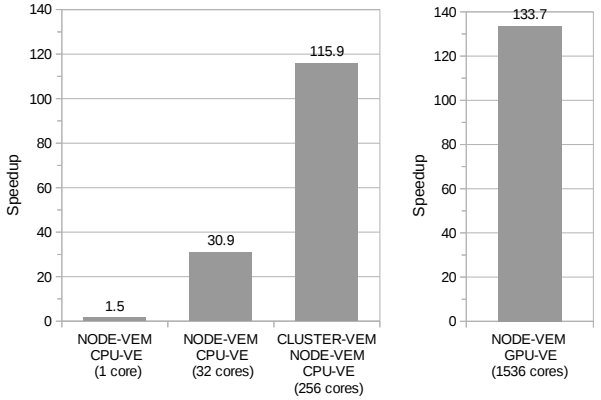
(a) (b)

Fig. 10: Relative speedup of the Black Scholes application. For the execution on the Cluster-node (a), the application generates 100m element vectors using 10 iterations. For the execution on the GPU-node (b), it generates 64m element vectors using 50 iterations.

For each scientific application kernel, we compare the Bohrium execution with the baseline execution using the same installation e.g. we compare a Node-VE/GPU-VE execution on the GPU-node with a baseline execution on the GPU-node. We calculate the relative strong scale speedup based on the average wall clock time of five executions and the input and output data is 64bit floating point for all executions. While measuring the performance, the variation of the measured wall clock timings did not exceed 5%.

A. Discussion

Shallow Water (Fig. 9): 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-core. The Bohrium execution is 2.1 times faster than the Native NumPy execution primarily because of memory allocation reuse of temporary arrays. The CPU-VE achieves limited scalability within one



(a) (b)

Fig. 11: Relative speedup of the N-Body application. For the execution on the Cluster-node (a), the application simulates 15k bodies in 10 iterations. For the execution on the GPU-node (b), it is 3200 bodies and 50 iterations.

Machine:	Cluster-node	GPU-node
Processor:	AMD Opteron 6272	AMD Opteron 6274
#CPUs per Node:	2	1
#Cores per CPU:	16	16
Clock:	2.1 GHz	2.2 GHz
L3 Cache:	16MB	16MB
Memory:	128GB DDR3	128GB DDR3
Peak:	134.4 GFLOPS	70.4 GFLOPS
Network:	Gigabit Ethernet	N/A
Compiler:	GCC 4.6.3	GCC 4.6.3 & OpenCL 1.1
GPU:	N/A	Nvidia GeForce (Table II)
Software:	Linux 3.2, Mono Compiler 2.10, Python 2.7, NumPy 2.6, Blitz++ 0.9	

TABLE I: Machine Specifications

node – a speedup of 13 when utilizing 32 cores through OpenMP. The problem is twofold: first, the CPU-VE execution has very poor cache utilization because it executes one vector operation at a time. Since the vector operations are very large and cannot fit in the cache, the CPU cannot exploit temporal locality. The result is a very memory bound execution that is limited by the Von Neumann bottleneck[25]. Secondly, the Shallow Water application uses mostly two-dimensional non-contiguous vector operations, which hinder the current OpenMP parallelization significantly.

The Cluster-VEM together with the CPU-VE demonstrates good scalable performance. Even without communication latency hiding, it achieves a speedup of 7.0 when going from utilizing one cluster-node fully to utilizing all eight cluster-nodes fully.

Finally, the GPU shows an impressive 140 speedup, which demonstrates the efficiency of parallelizing vector operations on a vector machine while removing temporary arrays and

Processor:	Nvidia GeForce GTX 680
#Cores:	1536
Core clock:	1006 MHz
Memory:	2GB DDR5
Memory bandwidth:	192 GB/s
Peak (single-precision):	3090 GFLOPS
Peak (double-precision):	128 GFLOPS

TABLE II: GPU Specifications

compiling multiple vector instructions into single kernels.

Black Scholes (Fig. 10): The Black Scholes application is embarrassingly parallel, which the scalability of the CPU-VE and the Cluster-VEM confirms. Similar to Shallow Water, the Black Scholes execution has very limited temporal locality. However, Black Scholes uses contiguous vector operations exclusively, which fits the current OpenMP parallelization very well. CPU-VE archives a speedup of 14 compared to Blitz++ and 17 compared to the CPU-VE single core execution.

The Cluster-VEM archives a speedup of 9.1 compared to the CPU-VE that utilize all 32 cores on a single node. The Hybrid Programming Model makes this superlinear speedup possible. Since the cluster-nodes are Non-Uniform Memory Access (NUMA) architectures with four NUMA nodes each, the Hybrid Programming Model confines the eight OpenMP threads within a MPI-process to a single NUMA node. The result is enhanced load-balance among the CPU memory channels.

Compared to Shallow Water, the GPU-VE achieves a slightly lower speedup mainly because of the faster baseline, i.e. Blitz++ is significantly faster than native NumPy and 20% faster than CPU-VE single core execution.

N-Body (Fig. 11): Unlike the two previous applications, Shallow Water and Black Scholes, the N-Body application uses many vector-operations that fit in the CPU cache, which makes temporal locality exploitation possible. The result is a speedup of 20 when the CPU-VE goes from a one core to 32 cores execution.

At first sight, the Cluster-VEM does not scale very well with a speedup of 3.7 going from one to eight cluster-nodes. However, an execution that uses four MPI-processes per cluster-node, which is one MPI-process per NUMA node and 32 MPI-process in total, and no OpenMP parallelization, archives an almost linear speedup of 31 (not shown in the figure). The problem appears when combining MPI and OpenMP parallelization because the Cluster-VEM may divide contiguous vector operation into non-contiguous vector operations when doing communication thus hindering the OpenMP parallelization.

Finally, the GPU demonstrate a good speedup of 134 compared to NumCIL.

B. Low-level Baseline

In order to judge Bohrium against a, close to, optimal baseline, we compare Bohrium with a low-level ANSI C implementation of the Shallow Water benchmark. The C implementation outperforms Bohrium with a speedup of 5.6 (not shown in the graph) because of explicit temporal locality exploitation and reduced memory access. In contrast to Bohrium, the C implementation applies all operations within a single loop and uses no temporary arrays.

However, the speedup comes with a price: the code use for-loops and pointer arithmetic in order to index the two-dimensional domain, which reduces the readability and makes it harder to generalize into more dimensions. Figure 12 and 13

```
long n = /*<grid size in x and y-axis>*/;
double H[(n+2)*(n+2)];     //Current wave hight
double U[(n+2)*(n+2)];     //Current momentum in x-axis
double V[(n+2)*(n+2)];     //Current momentum in y-axis
double Vnew[(n+2)*(n+2)]; //New momentum in y-axis
double g = 9.8; // gravitational constant
long i,j;

for(i=0; i<n+1; i++)
 for(j=0; j<n; j++)
  Vnew[i*n+j] = (V[(i+1)*n+j+1]+V[i*n+j+1])/2 -       \
   0.5*((U[(i+1)*n+j+1]*V[(i+1)*n+j+1]/H[(i+1)*n+j+1])\
       - U[i*n+j+1] * V[i*n+j+1]/H[i*n+j+1]);
```

Fig. 12: ANSI C implementation of the first half of the y-axis momentum in the Shallow Water benchmark. The complete Shallow Water implementation consist of nine similar calculations.

```
import bohrium as np
n = #<grid size in x and y-axis>
H = np.empty((n+2,n+2)) #Current wave hight
U = np.empty((n+2,n+2)) #Current momentum in x-axis
V = np.empty((n+2,n+2)) #Current momentum in y-axis
g = 9.8 # gravitational constant

Vnew = (V[1:,1:-1]+V[:-1,1:-1])/2 -          \
  0.5*((U[1:,1:-1]*V[1:,1:-1]/H[1:,1:-1]) -  \
       (U[:-1,1:-1]*V[:-1,1:-1]/H[:-1,1:-1]))
```

Fig. 13: Python/NumPy implementation of the first half of the y-axis momentum in the Shallow Water benchmark. The complete Shallow Water implementation consist of nine similar calculations.

illustrate a small subset of the calculation in the Shallow Water benchmark implemented in C and Python/NumPy respectively.

In conclusion, the C implementation outperforms the current version of Bohrium on a single CPU-core. However, through Bohrium the user may utilize multi-core architectures seamlessly thus on 32 CPU-cores, Bohrium surpass the performance of the C implementation.

VI. Future Work

From the presented experiments, we can see that the performance is generally good. However, 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, fuse computation loops, and eliminate temporary arrays. 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. We see similar results in the ANSI C implementation of the Shallow Water benchmark (Sec. V-B). With no temporary arrays and a single computation loop, the C implementation outperforms the current Bohrium implementation.

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.

In order to improve the Cluster performance, the internal

representation will facilitate optimization techniques, such as communication latency hiding and message aggregation, that can improve the scalability[26], [27].

The 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, enabling 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 aware multi-core CPUs, FPGAs, 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[28].

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 – multi-core CPUs, distributed memory Clusters, and GPUs. The preliminary results are very promising – a *Shallow Water* simulation achieves 140.2 speedup when comparing a Native NumPy execution and a Bohrium execution that utilize the GPU back-end.

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. E. 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 run-time 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] 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.

[21] 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.

[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] F. Cappello and D. Etiemble, "Mpi versus mpi+openmp on the ibm sp for the nas benchmarks," in *Supercomputing, ACM/IEEE 2000 Conference*, 2000, pp. 12–12.

[25] J. Backus, "Can programming be liberated from the von neumann style?: a functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978.

[26] 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.

[27] 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.

[28] 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.

7.4 Transparent GPU Execution of NumPy Applications

Troels Blum, Mads R. B. Kristensen, and Brian Vinter.
28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014).

Transparent GPU Execution of NumPy Applications

Troels Blum, Mads R. B. Kristensen and Brian Vinter
Niels Bohr Institute, University of Copenhagen, Denmark
{blum/madsbk/vinter}@nbi.dk

***Abstract*—In this work, we present a back-end for the Python library NumPy that utilizes the GPU seamlessly. We use dynamic code generation to generate kernels, and data is moved transparently to and from the GPU. For the integration into NumPy, we use the Bohrium runtime system. Bohrium hooks into NumPy through the implicit data parallelization of array operations, this approach requires no annotations or other code modifications.**

The key motivation for our GPU computation back-end is to transform high-level Python/NumPy applications to the low-level GPU executable kernels, with the goal of obtaining high-performance, high-productivity and high-portability, HP^3.

We provide a performance study of the GPU back-end that includes four well-known benchmark applications, *Black-Scholes*, *Successive Over-relaxation*, *Shallow Water*, and *N-body*, implemented in pure Python/NumPy. We demonstrate an impressive 834 times speed up for the Black-Scholes application, and an average speedup of 124 times across the four benchmarks.

I. Introduction

Computer simulations, which are widely used in both academia and the industry, often consists of large compute intensive tasks. This makes them good candidates for harvesting the computing power of modern, highly parallel computing systems, such as GPUs. The challenge lies in the fact, that these systems must be programmed using specialized programming models, which, even for skilled programming professionals, make the development cycle very long. This is a big and costly problem for both academic and industrial communities, which rely on an iterative method of developing new models.

The Python programming language and its de-facto scientific library NumPy targets the academic and the industrial community as a high-productivity framework with a very short development cycle[1]. Python/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 frequently used in programming languages and libraries that target the scientific community and the high-technology industry, e.g. HPF[2], MATLAB[3], Armadillo[4], and Blitz++[5].

In this paper, we describe a new computation back-end for the NumPy library that utilizes the GPU seamlessly. The idea is to offload all array operations to the GPU without any change to the original sequential Python code. In order to hook into the NumPy library, we make use of the Bohrium run-time system[6], which translates NumPy vector operations into an intermediate vector bytecode suitable for GPU parallelization.

II. Related Work

A framework such as pyOpenCL/pyCUDA[7] provides tools for writing GPU kernels directly in Python. The user writes OpenCL[8] or CUDA[9] specific kernels as text strings in Python, which simplifies the utilization of OpenCL or CUDA compatible GPUs. We have a similar goal – our approach, however, is entirely different. Instead of handling some management code specific for a given GPU API, we handle the complete process of writing GPU specific application automatically. The user needs not any GPU specific, or indeed any parallelization specific, knowledge.

In combination with Bohrium, we provide a framework 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. Particularly, the user must annotate variables with C data types. Similarly, the project Copperhead[11] relies on Python decorators, when compiling and executing a restricted subset of Python through CUDA. Because of the Bohrium runtime system, our GPU backend does not require any modifications to the Python code.

Python libraries such as CUDAMat[12] and Gnumpy[13] provide an API similar to NumPy for utilizing GPUs. The API of Gnumpy is almost identical with the API of NumPy. However, Gnumpy does not support arbitrary slicing when aliasing arrays.

III. The Python/NumPy Integration

In this work, we take advantage of the work done through 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. In our case, we use Bohrium to integrate a GPU specific implementation of NumPy array operations with the Python programming language.

The Python/NumPy support in Bohrium consists of an extension of NumPy version 1.6, which seamlessly implements a new array back-end that inherits the manipulation features, such as *view*, *slice*, *reshape*, *offset*, and *stride*. As a result, the user only needs to modify the import statement of NumPy in order to utilize the GPU back-end.

The concept of *views* is essential to NumPy programming. A view is a reference to a subpart of an array that appears as a regular array to the user. Views make it possible to implement a broad range of applications through element-wise vector (or array) operations. In Figure 1, we implement a heat equation solver that uses views to implement a 5-point stencil computation of the domain.

```
import bohrium as numpy
solve(grid, epsilon):
  center = grid[1:-1, 1:-1]
  north  = grid[ :-2, 1:-1]
  south  = grid[2:  , 1:-1]
  west   = grid[1:-1,  :-2]
  east   = grid[1:-1, 2:  ]
  delta  = epsilon+1
  while delta > epsilon:
    work = 0.2*(center+north+south+east+west)
    delta = numpy.sum(numpy.abs(work-center))
    center[:] = work
```

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 Bohrium and our GPU backend.

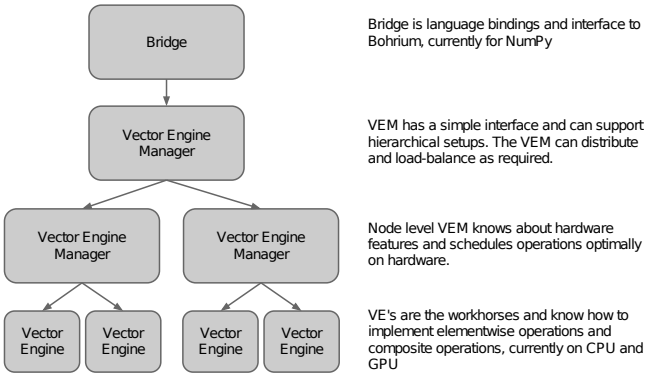


Fig. 2. Bohrium Overview

A. The Bohrium Runtime System

Bohrium consists of a number of components that communicate by exchanging a *Vector Bytecode*[1]. Components are allowed to be architecture-specific, but they are all interchangeable since all components use the same communications 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

[1]The name vector is roughly the same as the NumPy array type, but from a computer architecture perspective vector is a more precise term.

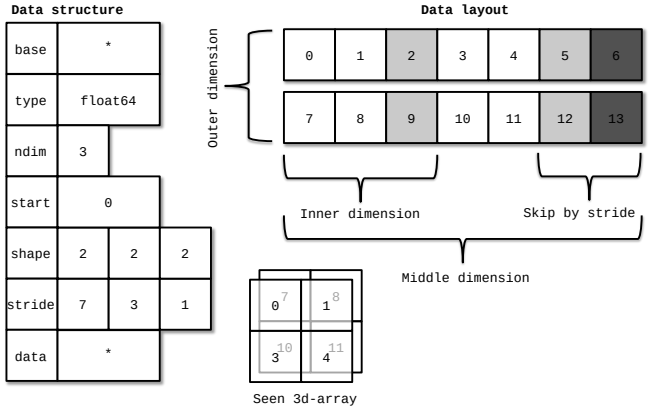


Fig. 3. Descriptor for n-dimensional array and corresponding interpretation

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.

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. The arrays can also be shaped into multi-dimensional arrays, to avoid excessive memory copying. 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 3 shows how the shape is implemented and how the data is projected.

Figure 4 illustrates a list of vector bytecode 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 bytecode. 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.

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 section, we will go through the four types of vector bytecodes in Bohrium.

1) Element-wise: Element-wise bytecodes perform 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, Bohrium only allows data

[2]Single Instruction, Multiple Data

[3]Single Program, Multiple Data

```
1  ADD t1, grid[center], grid[north]
2  ADD t2, t1, grid[south]
3  FREE t1
4  DISCARD t1
5  ADD t3, t2, grid[east]
6  FREE t2
7  DISCARD t2
8  ADD t4, t3, grid[west]
9  FREE t3
10 DISCARD t3
11 MUL work, const(0.2), t4
12 FREE t4
13 DISCARD t4
14 MINUS t5, work, grid[center]
15 ABS t6, t5
16 FREE t5
17 DISCARD t5
18 ADD_REDUCE t7, t6
19 FREE t6
20 DISCARD t6
21 ADD_REDUCE delta, t7
22 FREE t7
23 DISCARD t7
24 IDENTITY grid[center], work
25 FREE work
26 DISCARD work
27 SYNC delta
```

Fig. 4. Bytecode generated in each iteration of the Python/NumPy implementation of the heat equation solver (Fig. 1). For convenience, the arrays and views are given readable names. Views are annotate in "[]". Note that the `SYNC` instruction at line 27 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`).

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, Bohrium does 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 is 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 consist 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 memory allocations and copies.

4) Extension methods: The bulk of a Bohrium execution consists mainly of the above three types of bytecode. 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 a fourth bytecode type: extension methods. Bohrium imposes no restrictions to this generic operation; the extension writer has total freedom. However, Bohrium does not require 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[14].

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.

Due to the nature of Bohrium, being a runtime system, and Python being an interpreted language, the Python bridge will need to synchronize with Bohrium every time the `while` statement in figure 1 line 9 is evaluated i.e. for each loop iteration. This means that the body of the loop (Fig. 1 l. 10 – 12) resulting in the bytecode in figure 4 will be sent repeatedly by the bridge. Every time as a separate batch. In fact, Bohrium does not provide any control instructions at all.

D. Vector Engine Manager

In order to couple the Bridge and the hardware specific Vector Engine, Bohrium uses a third component: the Vector Engine Manager (VEM). The VEM is responsible for one memory address space in the hardware configuration. In our configuration, we only use a single machine (Node-VEM) thus the role of the VEM component is insignificant. The Node-VEM will simply forward all instruction from its parent to its child components.

E. Vector Engine

The Vector Engine (VE) is the only component that does computations, specified by the user application. It has to execute the instructions it receives in a valid order; that comply with the dependencies between instructions, i.e. program order. Furthermore, it has to ensure that its parent VEM has access to the results as governed by the Data Management bytecodes.

The VE is the focus of this paper – in order to utilize the GPU, we implement a GPU-VE that execute Bohrium bytecode on the GPU. In the following section, we will describe our GPU-VE in detail.

IV. The GPU vector engine

The Bohrium vector bytecode with its SIMD properties is a good match for the GPU. In this section, we will explain how we have chosen to convert the vector bytecode into code that executes on the GPU.

A. Implementation framework

In order to implement the GPU-VE, we use the OpenCL[8] framework. There are many possible choices for an implementation framework. One of the main goals of the Bohrium project is to deliver high performance. For this reason, we want a framework that allows for fairly low-level control of the hardware and relatively close mapping between the code we generate and the operations the hardware executes. The other candidates for this level of control are CUDA[9] and LLVM[15]. While LLVM could be a good choice for the code generated for the GPU kernels, LLVM depends on CUDA or OpenCL, to drive the GPU i.e. compiling and loading kernels and moving data to and from GPU memory.

The obvious alternative is the CUDA framework though it has a couple of drawbacks compared to OpenCL. Firstly, CUDA imposes a vendor lock in since it only supports devices by NVIDIA. We would prefer our solution to be vendor independent. Secondly, CUDA uses the pseudo-assembly language, PTX, which is more complex, requiring address calculations, and explicit load/store operations and register management. Alternatively, it is possible to compile C/C++ code to PTX using the external compiler, nvcc, and pay the relatively expensive cost of a system call. CUDA allegedly has a performance advantage (on NVIDIA devices) but studies[16] have shown that OpenCL achieves comparable performance under fair comparison. On top of this, we expect that the simplicity of the GPU kernels we generate makes the effect of advanced optimizations negligible.

B. Executing Bohrium bytecode on the GPU

The element-wise operations (Sec. III-B1) are considered the basic operations of Bohrium, as these are in effect vector operations. They are also the most common bytecodes received by the vector engine. A notable property of these vector operations is that all input and output operands have the same size even though the underlying base array might have different sizes. The Bridge will enforce this property through dimension duplications and/or vector slicing. Because of this property, mapping the element-wise operations to the GPU is straightforward through the SIMD execution model. Since the SIMD execution model is often recommended as an implementation model for the SIMT architecture of the GPU[9], it seems like a simple and logical choice. We map one thread to each output data element where each thread is responsible for fetching the required input data, doing the required calculations, and saving the result to the correct index in the output data.

1) Detecting kernel boundaries: A Bohrium bytecode represents a relatively simple operation, addition, subtraction, square root, less than, etc. The execution time for such an operation is very small compared to the time required to fetch the data elements from global memory, and writing the result back to global memory. Even when spawning millions of threads, and thus implementing latency hiding on the GPU, fetching and writing data is going to be the dominant time factor. The result is that single operation *microkernels* is not a viable solution for the GPU vector engine.

We need a scheme for collecting multiple operations into compound kernels for the GPU to run. Finding the optimal execution order for all the instructions in a batch is an NP-hard problem[17]. However, there exist different heuristic methods for finding *good* execution orders, while obeying the inter-instruction dependencies[18]. It is, however, out of the scope of this paper to do a performance analysis of these methods for our given setup. We have chosen to implement a simple scheme that guarantees a legal ordering of instruction since it does not reorder the instructions.

This scheme keeps adding instructions, in order, from the batch to the current kernel for as long as the instructions are compatible. When the scheme encounters a non-compatible instruction, it schedules the current kernel under construction for execution. Subsequently, the scheme initiates a new kernel with the non-compatible instruction as the first instruction. The scheme repeats this process until all instruction has been scheduled and executed. The criteria for a compatible instruction are:

1) The instruction has to be one of the element-wise operations. If it is a *reduction* or an *extension method* the current kernel will be executed before executing the reduction- or extension method instruction.
2) The shape and size of the data elements that are accessed must be the same as that of the kernel.
3) Adding the instruction can not create a data hazard. A data hazard is created if the new instruction reads a data object, that is also written in the same kernel, and the access patterns are not completely aligned. This is similar to loop fusion strategies implemented by compilers.

When the rules, outlined above, are applied to the bytecode example shown in figure 4, four kernel boundaries will be found. Resulting in four kernels. Of these four kernels only three of them will be executed on the GPU. The bytecode shown in figure 4 is produced by the loop body in figure 1, line 9 – 12, and is repeated for as long as `delta > epsilon`, although every iteration will produce a separate bytecode batch as explained in sec. III-C.

At the beginning of the batch, the current kernel is empty thus the first operation (the `ADD` in line 1) is added. The first operation that requires the insertion of a boundary and triggers kernel compilation and execution is the `ADD_REDUCE`-instruction in line 18, because this is not an element-wise operation. Reduction is handled separately, and it is compiled into its own kernel. The second reduction will also be handled as a separate kernel, except, it is not executed on the GPU. Rather the result of the previous reduction is transferred to the host memory, and the second reduction is executed on the host CPU. This is done for efficiency, as the second reduction is too small to utilize the GPU. Finally, the `IDENTITY`-operation in line 24, to copy the intermediate result back to the main grid, is compiled into its own kernel due the fact that the batch of instructions ends after the `SYNC`-instruction in line 27.

It is worth noticing that if the while loop from figure 1 was a for-loop instead, or in another way did not depend on the calculations within the loop, the batches would be concatenated into one, i.e. the bytecode in figure 4 would repeat with line 1 again after line 27. If this were the case, the `IDENTITY`-operation in line, 24 would still end up in its own kernel, because trying to add the following `ADD`-operation would break

```
__kernel void kernelb981208bc41e203a(
                      __global float* grid
                    , __global float* work
                    , const float s0
                    , __global float* t6)
{
    const size_t gidy = get_global_id(1);
    if (gidy >= 1000)
        return;
    const size_t gidx = get_global_id(0);
    if (gidx >= 1000)
        return;
    float center = grid[gidy*1002 + gidx*1 + 1003];
    float north = grid[gidy*1002 + gidx*1 + 1];
    float south = grid[gidy*1002 + gidx*1 + 1004];
    float east = grid[gidy*1002 + gidx*1 + 1002];
    float west = grid[gidy*1002 + gidx*1 + 2005];
    float t1;
    t1 = center + north;
    float t2;
    t2 = t1 + south;
    float t3;
    t3 = t2 + east;
    float t4;
    t4 = t3 + west;
    float work_;
    work_ = s0 * t4;
    float t5;
    t5 = work_ - center;
    float t6_;
    t6_ = fabs(t5);
    work[gidy*1000 + gidx*1 + 0] = work_;
    t6[gidy*1000 + gidx*1 + 0] = t6_;
}
```

Fig. 5. First of three kernels generated by the GPU-VE from the bytecode shown in Fig. 4. The kernel inplements line 1 – 17 of the bytecodes.

the third rule, listed above. As the input `grid[north]` is unaligned with the output `grid[center]` of the `IDENTITY`-operation.

Of the *data management* instructions, only the `DISCARD` instruction potentially effects the kernel. If it is one of the output arrays of one of the instructions of the current kernel, that means that the result is not used outside the kernel, and; therefore it does not need to be saved for later use. This saves the memory write of one data element per thread.

2) Moving data to and from GPU: The remaining *data management* instructions, `FREE` and `SYNC`, do not effect the content of the kernel. A `SYNC`-instruction may force the execution of a kernel, if the array in question is being written by the current kernel, and the data is the copied from the GPU to the main memory for availability to the rest of the system.

A `FREE`-instruction only concerns the main memory of the system thus; it is just executed when encountered.

There is no instruction for signaling that data needs to be copied to the GPU from main memory. The data will simply be copied to the GPU as it is needed.

3) Source code generation: Before a kernel can be scheduled for execution, it needs to be translated into an OpenCL C kernel function. The bytecode sequence shown in figure 4 will create three GPU kernels. Since the bytecode sequence is repeated for each iteration of the while loop of the Python/NumPy code from figure 1, the three kernels could be generated for each iteration. While generation of the OpenCL source code is relatively inexpensive, calling the OpenCL compiler and translating the source code into a hardware specific

```
__kernel void reduce6020b3d120d0ec9a(
                      __global float* t7
                    , __global float* t6)
{
  const size_t gidx = get_global_id(0);
  if (gidx >= 1000)
    return;
  size_t element = gidx*1 + 0;
  float accu = t6[element];
  for (int i = 1; i < 1000; ++i)
  {
    element += 1000;
    accu = accu + t6[element];
  }
  t7[gidx*1 + 0] = accu;
}
```

Fig. 6. Source code for the reduction kernel produced by line 18 of the bytecode in Fig. 4

```
__kernel void kernela016730fd6e00085(
                      __global float* grid
                    , __global float* work)
{
  const size_t gidy = get_global_id(1);
  if (gidy >= 1000)
    return;
  const size_t gidx = get_global_id(0);
  if (gidx >= 1000)
    return;
  float work_ = work[gidy*1000 + gidx*1 + 0];
  float center;
  center = work_;
  grid[gidy*1002 + gidx*1 + 1003] = center;
}
```

Fig. 7. Source code for the copy back kernel produced by line 24 of the bytecode in Fig. 4

kernel comes at a cost that has a significant impact on the total execution time of the program. To minimize the time spent on compilation, the GPU-VE will cache all compiled kernels for the lifetime of the program. The recurrence of a byte-code sequence is registered, and the compiled kernel is reused by simply calling it again, possibly with new parameters.

Every single bytecode is trivially translated into a single line of OpenCL C code, since every unique array-view just needs to be given a unique name. This way line 1 – 15 from figure 4 is translated into the calculation body an OpenCL function kernel: line 18 – 31 of figure 5, which is then prepended with the code for loading the needed data (line 13 – 17), and appended with code for saving the results (line 32 – 33). Notice that t1 ... t5 are not saved, as they are not needed outside the kernel. Code to make sure surplus threads do not write to unintentional addresses is inserted (line 8 – 9 & 11 – 12). Finally, the function header, with call arguments consisting of input and output parameters (line 1 – 5) are added to the code block. The kernel is then compiled and executed. The kernel code for the reduction is included in figure 6. Figure 7 shows the code for the copy-back-function generated by the `IDENTITY`-instruction in line 24 of figure 4.

V. Performance Study

We have conducted a performance study in order to evaluate how well the GPU-backend performs, compared to regular sequential Python/NumPy execution. This is by no means

Machine:	Workstation	Laptop
Processor:	Intel Core i7-3770	Intel Core i5-2410M
Clock:	3.4 GHz	2.3 GHz
#Cores:	4	2
Peak performance:	108.8 GFLOPS	37 GFLOPS
L3 Cache:	16MB	3MB
Memory:	128GB DDR3	4GB DDR3

TABLE I. SYSTEM SPECIFICATIONS

Machine:	Workstation		Laptop
Vendor:	AMD	NVIDIA	NVIDIA
Model:	HD 7970	GTX 680	GT 540M
#Cores:	2048	1536	96
Clock:	1000 MHz	1006 MHz	672 MHz
Memory:	3GB GDDR5	2GB DDR5	1GB DDR3
-bandwidth:	288 GB/s	192 GB/s	28.8 GB/s
Peak perf.:	4096 GFLOPS	3090 GFLOPS	258 GFLOPS

TABLE II. GPU SPECIFICATIONS

a study of how well Bohrium with the GPU-backend, or NumPy utilize the hardware. It is simply an illustration of the magnitude of speedup the end user can expect to experience, when using Bohrium with the GPU-backend. Keeping in mind that the transition from native Python/NumPy to Bohrium is completely seamless and requires no effort of the user. Wall clock time is measured for all benchmark executions, which include data transfers between the CPU and GPU. For the performance study, we use the following four well-known benchmark applications implemented in Python/NumPy:

Black Scholes The Black-Scholes pricing model is a partial differential equation, which is used in finance for calculating price variations over time[19]. This implementation uses a Monte Carlo simulation to calculate the Black-Scholes pricing model.

Successive Over-relaxation (SOR) A large 2D heat equation using Successive Over-relaxation. This is a variant of the Gauss–Seidel method for solving a linear system of equations, with faster convergence.

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 [20].

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. The algorithm is straightforward, and computes all body-body interactions, $O(n^2)$, with simple collision detection. We have chosen this naive implementation in order to represent a whole class of problems calculating forces of all pairs, often used in molecular dynamics.

The initial data for all benchmarks are constructed within the Python program, and as such can be initialized on the GPU when running the Bohrium enable interpreter.

All four benchmarks iterate through consecutive time steps, where the calculations for times step t_n requires the result for time step t_{n-1}.

All four applications are executed on both a workstation system (see table I), in two different GPU setups (see table II), and on a laptop system (see table I and II). The two workstation setups are chosen with two purposes: Firstly to demonstrate the magnitude of speedup one can expect from the Bohrium GPU implementation, in this first naive approach while demonstrating that the Bohrium GPU-VE is cross platform. Secondly to explore how GPUs from the two major vendors (NVIDIA and AMD) compare when used as Bohrium GPU execution engines. The two GPUs are reasonably new, and the prices are comparable. We also chose to run the benchmark application on a laptop system to demonstrate that such a system also benefits from running Python/NumPy through the Bohrium/GPU run-time system.

The four benchmarks are run using both 32 bit floats, and 64 bit floats (doubles). It is our experience, that most scientific applications are written using 64 bit floats. This is seldom a conscious choice on the part of the programmer, but the result of the fact that that is the default data type, of both NumPy, MATLAB, and other frameworks used by the scientific community. When moving the applications to the GPU, this choice of data type comes at a cost. The Fermi (NVIDIA) architecture performs eight single precision floating point operations for every double precision (8:1 ratio), and on the Kepler (NVIDIA) architecture it is a 24:1 ratio. The AMD Radeon HD 7900 series delivers double precision throughput at a 4:1 ratio relative to its single precision throughput. While one of the goals of the Bohrium project is to hide hardware specific choices from the programmer, the requirement of the data type is application specific. As such, it has to be the choice of the programmer. Maybe moving more applications to GPU will force scientific application developers to think more actively about the data type they chose.

For each benchmark, a Bohrium execution that uses the GPU-backend is compared with a native NumPy execution. Speedup is calculated based on the average wall clock time of five executions. While measuring the performance, the variation of the measured wall clock timings did not exceed 5% within any of the benchmarks. All measurements are preceded by a warm-up run.

Native NumPy/Python execution time is used as a baseline for calculating the speedup of the Bohrium/GPU engine. We view this as fair and relevant comparison since it is exactly the same code (Python script) that is executed in both setups. Still, it is relevant to ask: "How efficient is Python/NumPy code" This is a very open question, and, therefore, hard to answer. To quantify Python/NumPy's execution power, we have implemented the Black-Scholes benchmark using the Armadillo[4] library, and the Blitz++[5] library, for comparison. When running these benchmarks on datasets of the same size as for the GPU benchmarks, Armadillo is 24% faster than Python/NumPy and Blitz++ is 34% faster than Python/NumPy. Both numbers are the best achieved speedups, obtained on the biggest problems. Even though these are not negligible speedups, Python/NumPy's performance is still in the same order of magnitude. Acquiring these speedups requires rewriting the code. Bohrium with the GPU backend delivers speedups that are orders of magnitude better, while running the same exact code.

A. Results

The Black-Scholes- and N-body applications are both run for 50 time loop iteration, and the SOR- and shallow water

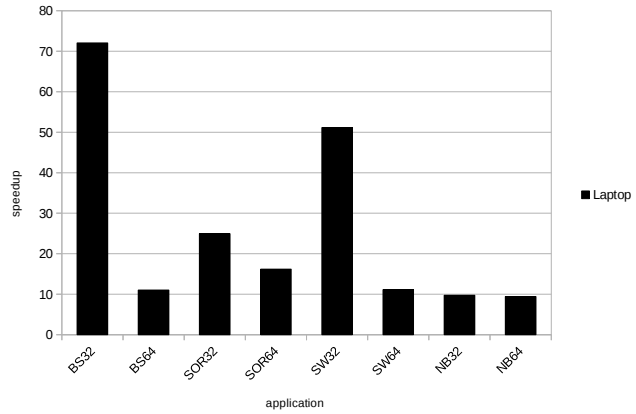


Fig. 8. Relative speedup of all application running on the laptop. **BS**: Black-Scholes, **SOR**: Successive Over-Relaxation, **SW**: Shallow Water, **NB**: N-Body. **32/64**: 32 or 64 bit floating point numbers.

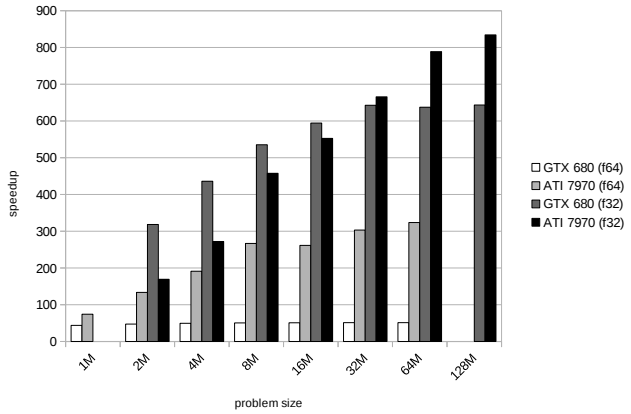


Fig. 9. Relative speedup of the Black the Scholes application running on the workstation

applications are both run for 100 iterations. This is far less than the applications would run in at "real" scenario, but enough that the majority of application run time is spent execution kernels on the GPU, thereby giving realistic and stable results.

Figure 8 shows the relative speedup of running the four applications with both double and single precision floats, on the laptop setup (see table I and II). It is clear to see the penalty payed for double precision in the Black-Scholes application. All of the applications are run with a problem size that allow them to fit within the 1GB of memory on the GPU. When running the applications with double precision data type the problem size is halved. Making all the data structures of the application take up the same amount of space independent of data type. Thus putting the same amount of stress on the memory bandwidth. The run-time for the BS64 is 3.92 times that of the BS32, requiring half the amount of floating point operations (half problem size). This comes pretty close to the specs of 1:8 single to double precision ratio. Indicating that Black-Scholes application running on the GPU is largely computational bound. This results in a speedup on the given laptop of 72 times for single precision, and 11 times for the double precision. Our assessment is that this is close to the optimum of what one can expect from running Python/NumPy applications through the Bohrium/GPU system, with the current implementation. The rest of the applications are to a larger, but varying, degree more memory bound. This results in a smaller but still significant speedup for these applications, and a smaller difference between using double or single point precision. It is worth noting that even the N-Body simulation, faring the worst, still delivers a 9.4 to 9.7 times speedup.

The Black-Scholes application is embarrassingly parallel, which makes it perfect for porting to the GPU. Even with the relatively simple scheme for kernel generation, the GPU-VE currently implements; it generates only *one* kernel per iteration of the main loop. The result is a very effective execution that achieves a speedup of 834 times (ATI) and 643 times (NVIDIA) respectively for the largest 32bit float problems (Fig. 9). Additionally, it clearly demonstrates the comparably poor 64bit performance of the Kepler architecture (NVIDIA). Remember: the GTX 680 deliver 1/24 double

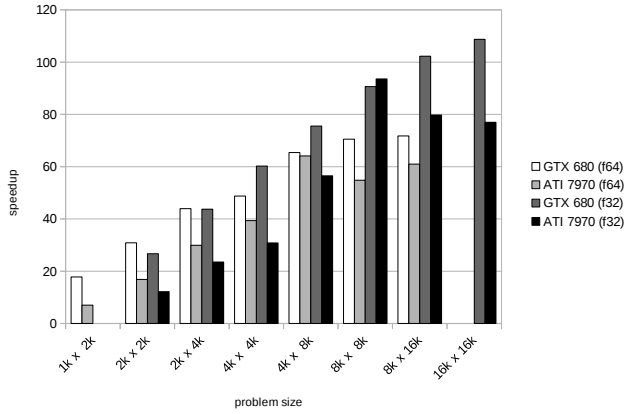


Fig. 10. Relative speedup of the SOR application running on the workstation

precision operation per single precision operation according the specifications, which is worse than the ratio of 1:14 seen in the Black-Scholes benchmark. This indicates that even in the embarrassingly parallel Black-Scholes application, which generates the largest kernel and has the best operation to calculation ratio, memory bandwidth still plays a role as a limiting factor.

The SOR application is the most memory bound and the least compute intensive of the four applications. Still, it is clearly beneficial to utilize the GPU through Bohrium (Fig. 10). For the largest problem size, it achieves a speedups of 109 and 94 times for the single precision versions and 72 and 61 times for the double precision versions. Even for the smallest problem size, it achieves a significant speedup. The drop-off in performance for the ATI GPU for single precision from 8k x 8k to 16k x 16k is something that needs further investigation.

The shallow water application works on several distinct arrays and has more complex computational kernels, compared to the SOR application. The more complex kernel is why we are able to get better performance. Again we observe the same drop off in performance for the largest problem size – though this time on the NVIDIA GPU (see fig. 11). The more curious

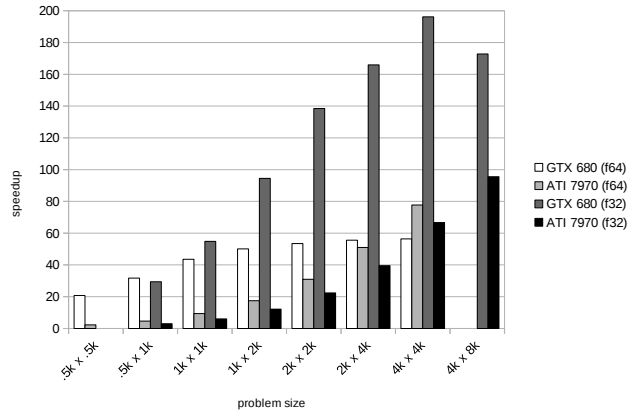


Fig. 11. Relative speedup of the shallow water application running on the workstation

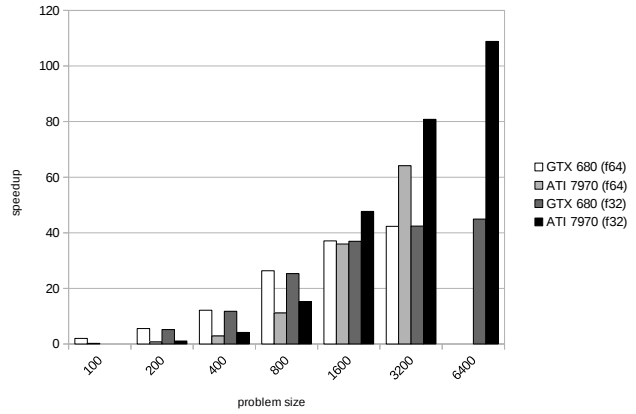


Fig. 12. Relative speedup of the N-body application running on the workstation

observation one can make from the graph in figure 11 is that the ATI GPU performs much poorer than NVIDIA. ATI has both better specifications in both memory bandwidth and peak performance. We will have to investigate whether the code we generate favors NVIDIA GPUs, and if we can do something to remedy this.

The straight forward algorithm used in the N-body simulations computes distances of all pairs. Expanding the N-body data to $O(N^2)$ data points. While calculating the forces, the data is reduced back to the original $O(N)$ size. Due to the simple algorithm use in the GPU-VE, outlined in section IV-B1, the reduction will force a kernel boundary. Resulting in the expanded data being written back to global memory, before being read again by another reduction kernel – thereby being reduced. The large space requirements, due to the all pairs expansion, also puts an unfortunate limitation on the problem sizes Numpy is able to run. Only the two largest problem sizes are theoretically able to use all the core on the two GPUs, which leaves little room for latency hiding. Still, the Bohrium GPU backend is able to achieve up to 40–100 times speedup as Figure 12 illustrate.

One of the future plans for the GPU-VE is to be able to generate kernels, where the expansion and reduction happen within the same kernel. This would dramatically reduce both the space requirements and the load on the limited memory bandwidth for this type of application.

It is clear from the graphs in figure 9–12 that the bigger the problem size, the better suited it is for execution on the GPU. This is no surprise since a bigger problem, will instantiate more threads, better utilizing the many cores of the GPUs, and at the same time enabling better latency hiding for the memory fetches. It is also expected, that there is a certain initialization cost for calling an external library, generating and decoding the bytecode, generation kernels and source code and invoking the GPU kernels. All of the experiments above have been run for a small, but sufficient number of iterations that the initial costs are amortized. To illustrate that the initialization costs are not excessively large, all four benchmark applications were run for just a single iteration. The Black-Scholes application still shows a speedup of 10–500 times dependent on the problem size for a single iteration. The SOR and Shallow water applications show speedup for all, but the two smallest problem sizes (up to 30 times). Finally, the N-body application only shows speedup for the two largest problem sizes with a single iteration – keeping in mind that it is only these problem sizes that theoretically are able to utilize all cores. All the experiments that do not show a speedup for a single iteration has a total execution time of less than 0.4 seconds. We feel that this is a very good result.

VI. Future Work

As the performance study demonstrates, we are able to show some significant performance improvements from running the Python/NumPy applications on the GPU. The improvements are possible through relatively simple strategy and implementation thus we expect to improve the GPU utilization even further.

The Bohrium project is currently working on a directed acyclic graph (DAG) representation for bytecode dependencies that will enable out-of-order execution and improved temporary array elimination. Furthermore, the DAG will enable the generation of larger, more complex kernels and better kernel reuse while easing the detection of code blocks that expands and reduces data, as the N-body application does. We expect to be able to contain these expansions/reductions within a kernel, effectively boosting performance and drastically reducing the memory footprint of such applications.

For now, the GPU-VE is limited to executing an application where the maximum memory footprint never exceeds the available memory on the GPU. The goal is to enable the GPU-VE to work on data sets that do not fit within the GPU memory, by employing an array splitting and buffering scheme. This will in turn enable us to experiment with utilizing multiple GPUs that collaborate on solving the same task.

VII. Conclusion

In this work, we show that it is possible to run unmodified Python/NumPy code on modern GPUs. We use the Bohrium runtime system to translate the NumPy array operations into an array based bytecode sequence. Executing these bytecodes

on two GPUs from different vendors shows great performance gains. In particularly the Black-Scholes application achieved a speedup of 834 times compared with native NumPy execution.

We believe that scientist working with computer simulations should be allowed to focus on their field of research and not spend excessive amounts of time learning exotic programming models and languages. We have achieved very promising results with a relatively simple approach. We expect to be able to improve further in these results in the future making well-known, readily available programming languages with a high abstraction level a viable choice for high performance computer simulations.

REFERENCES

[1] T. E. Oliphant, *A Guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[2] D. Loveman, "High performance fortran," *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 1, no. 1, pp. 25–42, 1993.

[3] W. Yang, W. Cao, T. Chung, and J. Morris, *Applied numerical methods using MATLAB*. Wiley-Interscience, 2005.

[4] 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.

[5] 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.

[6] 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.

[7] A. Klöckner, 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.

[8] A. Munshi *et al.*, "The OpenCL Specification," *Khronos OpenCL Working Group*, vol. 1, pp. 11–15, 2009.

[9] C. Nvidia, "Programming guide," 2008.

[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] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: compiling an embedded data parallel language," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 47–56. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941562

[12] V. Mnih, "Cudamat: a cuda-based matrix class for python," *Department of Computer Science, University of Toronto, Tech. Rep. UTML TR*, vol. 4, 2009.

[13] T. Tieleman, "Gnumpy: an easy way to use gpu boards in python," 2010.

[14] L. S. Blackford, "ScaLAPACK," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing 96 Supercomputing 96*, 1996, p. 5.

[15] 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.

[16] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of cuda and opencl," in *Proceedings of the 2011 International Conference on Parallel Processing*, ser. ICPP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 216–225. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2011.45

[17] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[18] A. A. Khan, C. L. McCreary, and M. S. Jones, "A comparison of multiprocessor scheduling heuristics," in *Proceedings of the 1994 International Conference on Parallel Processing - Volume 02*, ser. ICPP '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 243–250.

[19] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *The journal of political economy*, pp. 637–654, 1973.

[20] J. Burkardt, "Shallow water equations," people.sc.fsu.edu/~jburkardt/m_src/shallow_water_2d/, [Online; accessed March 2010].

7.5 Separating NumPy API from Implementation

Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, and Kenneth Skovhede.
5th Workshop on Python for High Performance and Scientific Computing (PyHPC 2014).

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

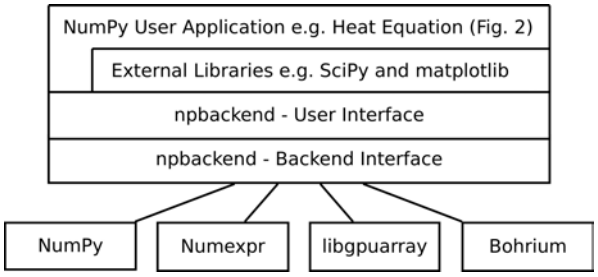


Fig. 1: The Software Stack

.

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.

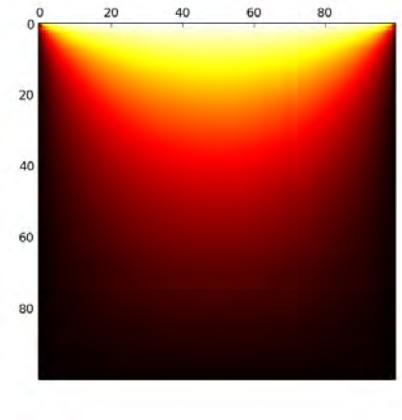


Fig. 3: The *matplotlib* result of executing the heat equation solver from figure 2: `solve(100,100)`

.

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.

```
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.

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.

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 `memmoves` 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.

[2]Using a victim cache

```
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()`

```
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.

```
import pygpu
import backend_numpy
class base(backend_numpy.base):
  def __init__(self, size, dtype):
    self.clary = pygpu.empty((size,), dtype=dtype, cls=elemary)
    super(base, self).__init__(size, dtype)

class view(backend_numpy.view):
  def __init__(self, ndim, start, shape, stride, base):
    super(view, self).__init__(ndim, start, shape, stride, base)
    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)

def get_data_pointer(ary, allocate=False, nullify=False):
  ary.ndarray[:] = np.asarray(ary.clary)
  return ary.ndarray.ctypes.data

def set_bhc_data_from_ary(self, ary):
  self.clary[:] = pygpu.asarray(ary)

def ufunc(op, *args):
  args = [a.ndarray for a in args]
  out=args[0]
  i1=args[1];
  if len(args) > 2:
    i2=args[2]
  cmd = "out[:] = %s"%ufunc_cmds[op]
  exec cmd

def extmethod(name, out, in1, in2):
  (out, in1, in2) = (out.clary, in1.clary, in2.clary)
  if name == "matmul":
    pygpu.blas.gemm(1, in1, in2, 1, out, overwrite_c=True)
  else:
    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.

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

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`).

VI. Benchmarks

In order to evaluate the performance of npbackend, we perform a number of performance comparisons between a

[3]Common Intermediate Language

```
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

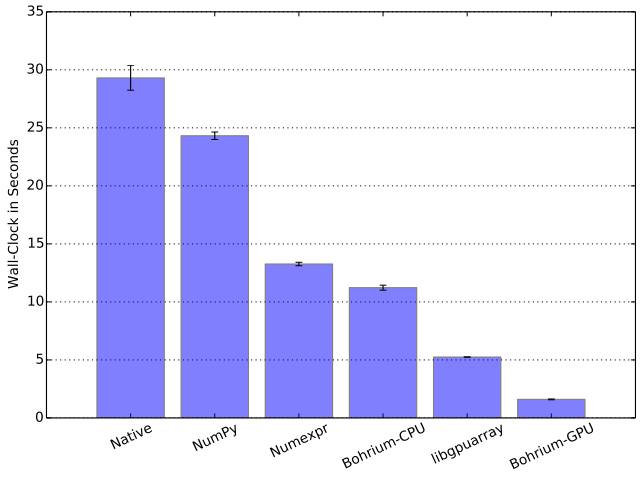


Fig. 9: The Heat Equation Benchmark where the domain size is 3000^2 and the number of iterations is 100.

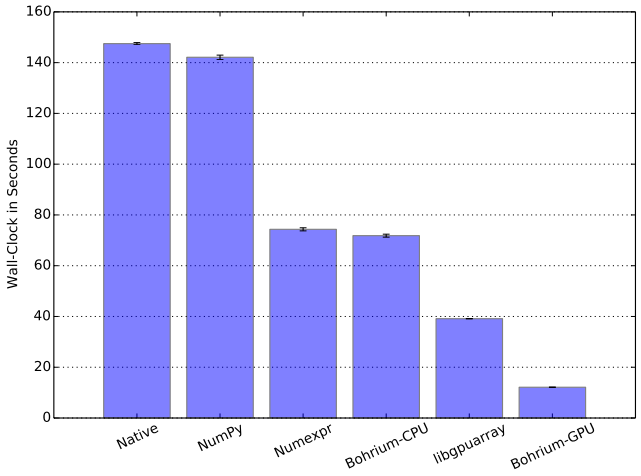


Fig. 10: The Shallow Water Benchmark where domain size is 2000^2 and the number of iterations is 100.

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,

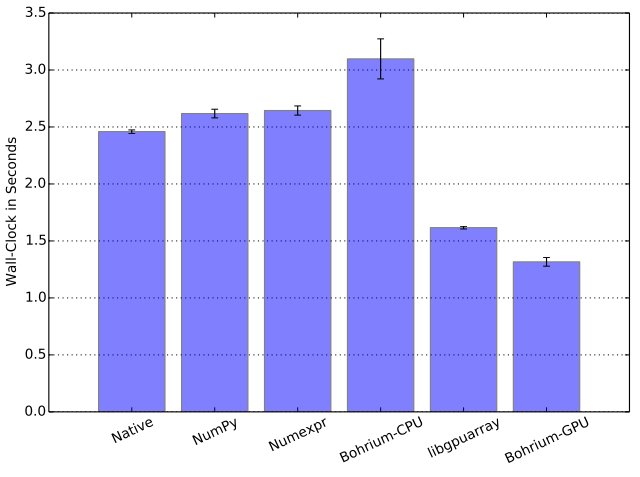


Fig. 11: The Snakes and Ladders Benchmark where the domain size is 1000^2 and the number of iterations is 10.

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%.

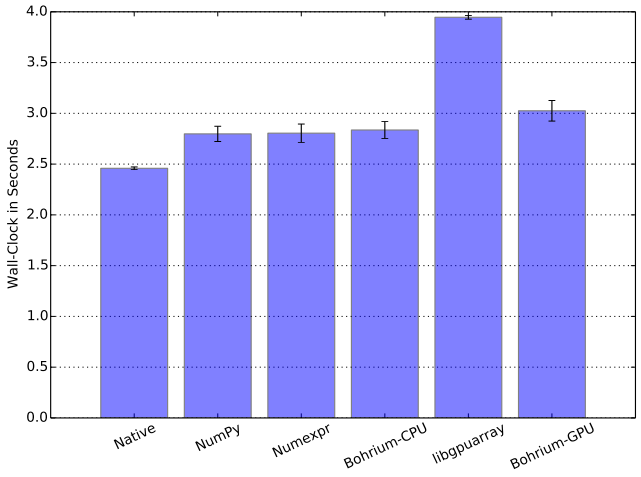


Fig. 12: The Snakes and Ladders Benchmark where the backends does not have matrix multiplication support. The domain size is 1000^2 and the number of iterations is 10.

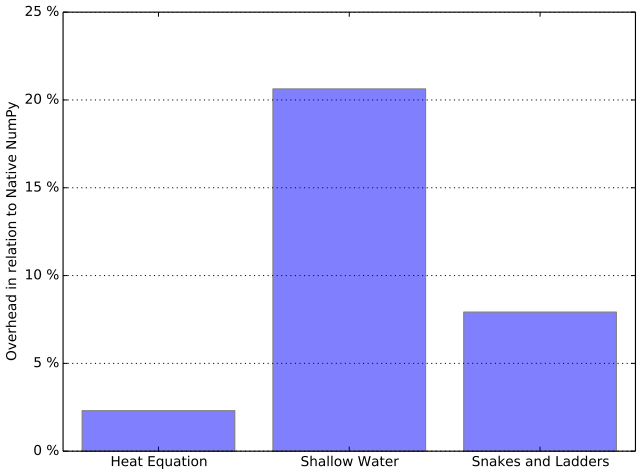


Fig. 13: Overhead of npbackend where we compare the NumPy backend with the native NumPy execution from the previous benchmarks.

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 run-time 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.

7.6 Code Specialization of Auto Generated GPU Kernels

Troels Blum and Brian Vinter.
Communicating Process Architectures 2015.

Communicating Process Architectures 2015
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2015

Code Specialization of Auto Generated GPU Kernels

Troels BLUM [a] and Brian VINTER [a],
[a] *University of Copenhagen, Niels Bohr Institute*

Abstract. In this work we explore and evaluate the effect of automatic code specialization on auto generated GPU kernels. When combining the high productivity coding environment of computational science with the Just-In-Time compilation nature of many GPU runtime systems there is a clear cut opportunity for code optimization and specialization. We have developed a hybrid kernel generation method which is shown to be useful and competitive across very different use cases, and requires minimal knowledge of the overall structure of the program.

Stencil codes which are commonly found at the core of computer simulations are ideal candidates for this type of code specialization. For exactly this type of application we are able to achive speedups of up to 2.5 times with the implemented strategy.

Keywords. Automatic Code Optimization, Code Generation, GPU, JIT, Python, Scientific Byte Code

1. Introduction

There is a continuing, and growing, interest in constructing mathematical models and quantitative analysis techniques, i.e. computational science, in both academia and in industry. This results in an increasing number of programs that are written by domain experts, as opposed to trained programmers. At the same time the availability and power of accelerator cards, e.g. Graphics processing units (GPU), and coprocessors, e.g. Xeon Phi, is also increasing. These technologies promise to deliver more bang for the buck over conventional CPUs. However, these technologies require programming experts, i.e. engineers and computer scientists, to program.

The gap between these fields of expertise can be overcome by employing both domain experts and programming experts. However, this solution is often both expensive and time consuming, since coordination between scientists and programmers constitutes an overhead. Other possible solutions include developing domain specific languages (DSLs). Such DSLs may either be compiled or interpreted, in both cases JIT compilation may be involved, for example Fortress[1] and X10[2] running on the Java Virtual Machine. One may port or develop accelerator enabled libraries like QDP-JIT/PTX[3], which is a lattice quantum chromodynamics (QCD) calculation library, a port of the QDP++[4] library, which may help domain experts to leverage the power of accelerators without learning accelerator based programming, naturally the performance imprpovements is then limited to the portion of the code that uses accelerated libraries.

For general purpose programming languages such as C/C++ or Fortran annotation in the shape of pragmas as in OpenACC[5] is often used to annotate parts of the code that is well suited for execution on the GPU. This may not in fact bridge the gap between domain and programming expertise, as both the languages and correct use of pragmas require a high level of programming and architecture knowledge. Template libraries like Thrust[6] from NVIDIA or Bolt[7] from AMD are also available. They save the programmer the trouble of

writing some boiler plate code, and contain implementations of standard algorithms. Again some architecture knowledge is required, and most problems can not simply be solved by gluing standard algorithms together. SyCL[8], a specification from the Khronos group, and C++ AMP[9], from Microsoft, are both single source solution compilers for parallel, heterogeneous hardware. This allows for GPU kernel code to be templated, and methods can be implemented on vector data types for seamless parallelization. All of these technologies, while useful for skilled programmers, are of little help to domain experts as they still require advanced programming skills and hardware knowledge.

An increasingly popular choice for domain experts is to turn to interpreted languages[10] like MATLAB[11] or Python with the scientific computing package NumPy[12]. These languages allow the scientist to express their problems at higher level of abstraction, and thus improves their productivity as well as their confidence in the correctness of their code. The function decorators of SEJITS[13] is one way to utilize accelerators (GPUs) from Python, another is project Copperhead[14] which rely on decorators to execute parts of the Python code on GPUs through CUDA[15]. It is also possible is to use a more low level framework such as pyOpenCL/pyCUDA[16], which provides tools for writing GPU kernels directly in Python. The user writes OpenCL[17] or CUDA specific kernels as text strings in Python. This allows for the control structure of the program to be written in Python while also avoiding writing boilerplate OpenCL- or CUDA code. This still required programming knowledge of OpenCL or CUDA.

The authors have previously presented a Python/NumPy backend framework[18], which simplifies changing the execution engine of Python/NumPy to a number of low level APIs, including pyOpenCL. The Bohrium project[1] contains several backends for Python/Numpy including one the utilizes the GPU[19]. This is completely transparent to the programmer, no change to the Python/Numpy code is required. This approach provides speedup levels in the hundreds when utilizing the GPU. Both the GPU backend replacement and Bohrium library, are part of the Bohrium[20] project, which is a drop-in replacement for NumPy, and makes the use of accelerators completely transparent to the user. The work described in this paper is part of the Bohrium project, where the optimizations are implemented in the GPU execution engine.

Common for many of the approaches to accelerator enabling code is that they involve JIT compilation for the target hardware. In OpenCL[17], which is the de facto standard for cross platform, vendor independent, GPU programming, JIT compilation is the default mode of operation. When using the OpenCL framework GPU kernels are included as text strings in the program. Which are then compiled at into hardware specific binaries by the runtime API at runtime.

Even though GPU code is often compiled at runtime the source code that is compiled is generally static. Optimizations are implemented by the compiler thus they are vendor specific, and may even be target specific. Compiling at runtime ensures that the compiler can know more about the problem, that we are trying to solve, than the programmer did at the time of programming. This is due to the fact that we write code to solve a type of problem, but when we are running the code we are solving a specific instance of that problem, making it possible for us to do automatic code specialization at runtime without altering the original program code.

In this work we investigate the benefits of code specialization at runtime. Providing the compiler with more information to work with, should enable it to generate more efficient code. As we show in Section 4 a simple specialization scheme can result in a significant speedup. First we will be setting the scene by giving a quick overview of how Bohrium works

[1]Bohrium is an open source project. The source code can be downloaded from `http://www.bh107.org`. Further documentation is also available at the website.

and how it views data in Section 2. Then we will outline our strategy for exploiting JIT compilation for the GPU, using the Bohrium library in Section 3. In Section 5 we will speculate on where to go from here, and in Section 6 we will summarize on the work described in this paper.

2. Bohrium

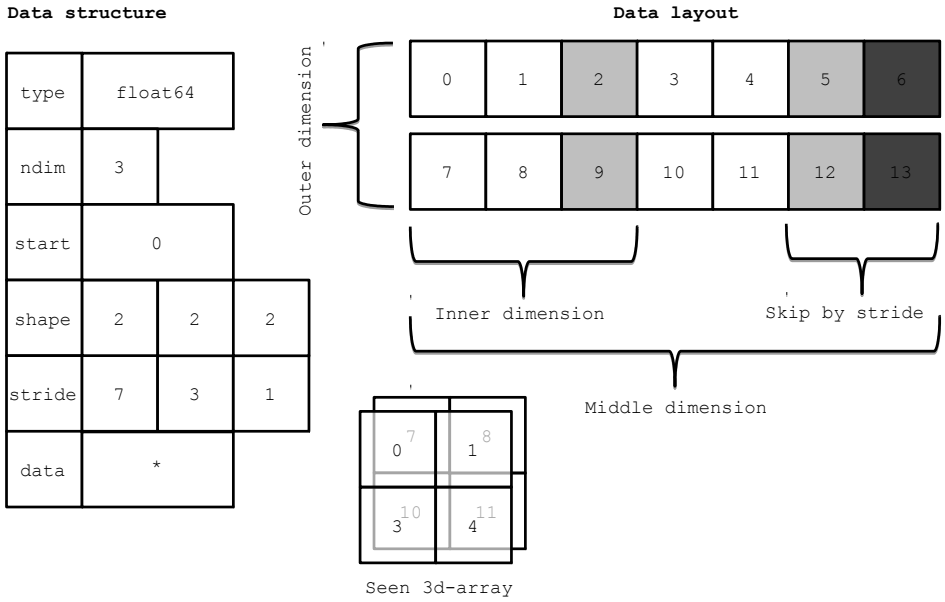


Figure 1. Bohrium view descriptor for an n-dimensional array and corresponding interpretation.

The Bohrium project builds upon a common, vector based, byte-code language. The Bohrium byte-code supports operations on multidimensional arrays, which can be sliced into sub-arrays, broadcast into higher dimensions, or viewed in other exotic ways. High level languages construct and operations are translated into this byte-code, that in turn will be executed by the different Bohrium vector engines. Every bytecode contains information on how the data of each array operand should be accessed (*viewed*). Each *view* consists of a data type, number of dimensions, an offset, and number of elements and a stride[2] for each dimension. See Figure 1 for a graphical representation of a Bohrium view.

The Bohrium operator access patterns are prime candidates for code specialization. The GPU vector engine of Bohrium translates the vector byte code into OpenCL bases GPU kernels at run-time. We can choose to make the generated kernels more general by including the access patterns as parameters, or specialize the kernels by including the access patterns as literals. It is our expectation that specialized kernels will perform better than non-specialized due to the fact the compiler will have more information to work with.

2.1. Limitations

Bohrium byte-code does not support loops or control structures. Bohrium relies on the host language for these programming constructs. When iterating through a loop the same byte-code sequence will simply be repeated. Though operands and access patterns may change with each iteration. It is important for the vector engine to detect these repetitions since JIT

[2]Number of base elements to skip ahead, to access the next data element in the given view, may be negative.

compiling OpenCL source code at run-time is a time consuming task, compared to the execution time of the generated kernel. That way JIT compiling cost is amortized with repeated calls to same OpenCL kernel. We need to ensure that our specialization of the generated GPU kernels do not prohibit them from being reused. If we consider the loop body of LU decomposition, as it may implemented in Python/NumPy, as shown in Figure 2. The *views* on both `l` and `u` change with each iteration. When translated into Bohrium bytecode the specific view attributes are concretized, i.e. no longer symbolic.

```
1 def lu(a):
2     u = a.copy()
3     l = numpy.zeros_like(a)
4     numpy.diagonal(l)[:] = 1.0
5     for c in xrange(1,u.shape[0]):
6         l[c:,c-1]  = u[c:,c-1] / u[c-1,c-1:c]
7         u[c:,c-1:] = u[c:,c-1:] -
8                      l[c:,c-1][:,None] * u[c-1,c-1:]
9     return (l,u)
```

Figure 2. Python/NumPy implementation of LU decomposition.

Bohrium is designed to support interpreted languages. As a result of this design choice we may need to execute calculations based on very limited knowledge of the general structure of the host program — since we have no knowledge of future calculations. This needs to be taken into account when devising a strategy for code specialization. Consider the Jacobi stencil code in Figure 3: The Python interpreter evaluates the boolean clause of the while statement in line 3, thus the value of `delta` is returned to the host-language bridge together with the control for each iteration of the loop. So from Bohriums point of view we do not know that the views do not change. At least not for the first calculation of `delta`. With each iteration we could assume static views with higher and higher confidence.

```
1  def jacobi_2d(grid, epsilon=0.005):
2      delta  = epsilon + 1
3      while delta > epsilon:
4          work = (grid[1:-1,1:-1] + grid[0:-2,1:-1] +
5                  grid[1:-1,2:] + grid[1:-1,0:-2] +
6                  grid[2:,1:-1]) * 0.2
7          delta = numpy.sum(numpy.absolute(work -
8                                           grid[1:-1,1:-1]))
9          grid[1:-1,1:-1] = work
10     return grid
```

Figure 3. Python/NumPy implementation of 2D-Jacobi stencil.

3. Specialization Strategy

There are two extremes of code specialization, focusing on the views or access patterns of Bohrium. The first is of course to specialize everything found in the Bohrium view descriptor, i.e. use only literals. The other extreme is to parametrisize the generated kernels with the Bohrium view descriptor, i.e. use only function variables. Choosing no specialization gives us the best chance of code reusability, resulting in less time spent on the compiler compiling code if the indexes change, but *perhaps* more time calculatng data indexes during execution. Symmetrically, the effect of full specialization is the reverse: Giving the compiler as much

information as possible to work with, hopefully enabling better optimized code. The optimal solution would be to include exactly those values that do not change over the lifetime of the program as literals. In our scenario, however, we do not have the global knowledge required for implementing this optimal solution. We may, depending on the application, actually have very limited knowledge — it would be desirable with a strategy that is both simple, and works well in most situations.

The strategy we have chosen for this work, testing the benefits of code specialization, if any, is as follows:

> The first time a unique set of byte codes, which constitute a GPU kernel, are encountered: We compile both a completely specialized kernel and a generalized kernel, i.e. the two extremes. The compilations are done separately and asynchronously. We enqueue the fully specialized kernel for execution. As we expected this to be the most efficient version of the kernel. We also expected the compilation time for both kernels to be largely the same.[3]
> Upon receiving the same pattern of byte codes, constituting a kernel, again, we know that we have a matching kernel i.e. the generalized, parametrizied, kernel. The specialized version of the kernel is tested for fitness, i.e. the access pattern is the same as when the code was generated. If the specialized kernel matches it is scheduled for execution, otherwise the generalized kernel is scheduled.

Most of the code for the generated kernels are the same, independent of specialization, since the core functionality is the same. So we found it most convinient to generate one source code, and use the c preprocessors `#define` directives to generate the different GPU kernels. The Python code shown in Figure 3 generates two GPU kernels. The main body of the generated OpenCL source code is shown in Figure 4, and the define and size parameter parts are shown in Figure 5.

4. Performance Study

We have conducted a small performance study in order to evaluate:

- What are the benefits of doing code specialization on auto generated code.
- How well is our simple strategy for code specialization suited for taking advantage of these benefits.

We will in this Section use the following terminology:

Dynamic is the fully parameterized kernel. This is what we regard as our base scenario.

Fixed is the fully specialized kernels, where all view parameters are literalized.

Selected represents the case where we generate and compile both kernels, and then select the best suited for execution, as explained in Section 3.

4.1. Benchmarks applications

We have chosen six benchmark applications for the performance study. All of the applications are implemented in Python using the Bohrium NumPy bridge. The applications are simple, and have been chosen to represent one of two categories:

- Applications where the views do not change throughout the execution, and therefore will benefit from specialization.

[3] As shown in Table 3 and 4 compilation time for dynamic and fixed kernels are often similar but not always.

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#ifdef FIXED_SIZE
// defines go here
#endif
__kernel __attribute__((work_group_size_hint(64, 4, 1))) void
#ifndef FIXED_SIZE
kernel429d67e832590722
#else
kernel429d67e832590722_
#endif
(
      __global double* a1
    , __global double* a5
    , __global double* a7
    , const double s0
#ifndef FIXED_SIZE
// size parameters go here
#endif
)

{
    const size_t gidx = get_global_id(0);
    if (gidx >= ds0)
        return;
    const size_t gidy = get_global_id(1);
    if (gidy >= ds1)
        return;
    double v1 = a1[gidy*v1s2 + gidx*v1s1 + v1s0];
    double v2 = a1[gidy*v2s2 + gidx*v2s1 + v2s0];
    double v4 = a1[gidy*v4s2 + gidx*v4s1 + v4s0];
    double v6 = a1[gidy*v6s2 + gidx*v6s1 + v6s0];
    double v8 = a1[gidy*v8s2 + gidx*v8s1 + v8s0];
    double v0;
    v0 = v1 + v2;
    double v3;
    v3 = v0 + v4;
    double v5;
    v5 = v3 + v6;
    double v7;
    v7 = v5 + v8;
    double v9;
    v9 = v7 * s0;
    double v10;
    v10 = v9 - v1;
    double v11;
    v11 = fabs(v10);
    a5[gidy*v9s2 + gidx*v9s1 + v9s0] = v9;
    a7[gidy*v11s2 + gidx*v11s1 + v11s0] = v11;
}
```

Figure 4. Generated OpenCL kernel created by 2D-Jacobi stencil.

- Applications where the views change repeatedly. If we use specialization for this type of application, we will trigger a new compilation for every kernel execution. Which is very costly.

Four applications are dense stencil applications of increasing dimensionality, and two are algorithms used in linear algebra.

Stencil applications are good candidates for benefiting from specialization. We have chosen 1D – 4D dense stencil applications. They are all of the same structure as the Jacobi code shown in Figure 3.

```
#define ds1 1998
#define ds0 1998
#define v1s0 2001
#define v1s2 2000
#define v1s1 1
#define v2s0 1
#define v2s2 2000
#define v2s1 1
#define v4s0 2002
#define v4s2 2000
#define v4s1 1
#define v6s0 2000
#define v6s2 2000
#define v6s1 1
#define v8s0 4001
#define v8s2 2000
#define v8s1 1
#define v9s0 0
#define v9s2 1998
#define v9s1 1
#define v11s0 0
#define v11s2 1998
#define v11s1 1
```

```
    , const int ds1
    , const int ds0
    , const int v1s0
    , const int v1s2
    , const int v1s1
    , const int v2s0
    , const int v2s2
    , const int v2s1
    , const int v4s0
    , const int v4s2
    , const int v4s1
    , const int v6s0
    , const int v6s2
    , const int v6s1
    , const int v8s0
    , const int v8s2
    , const int v8s1
    , const int v9s0
    , const int v9s2
    , const int v9s1
    , const int v11s0
    , const int v11s2
    , const int v11s1
```

Figure 5. The define and parameter declaration part of the generated code in figure 4.

```
1 def gauss(a):
2     for c in xrange(1,a.shape[0]):
3         a[c:,c-1:] = a[c:,c-1:] -
4                      (a[c:,c-1]/a[c-1,c-1:c])[:,None] *
5                      a[c-1,c-1:]
6     a /= numpy.diagonal(a)[:,None]
7     return a
```

Figure 6. Python/NumPy implementation of Gaussian elimination without pivoting.

1D is a 3 point stencil application containing 9 indexing components[4].

2D is a 9 point stencil application containing 45 indexing components.

3D is a 27 point stencil application containing 189 indexing components.

4D is a 81 point stencil application containing 729 indexing components.

Gaussian elimination without row pivoting as shown in Figure 6. The different array indexes change for each loop iteration. This in not suitable for pure specialization, thus testing our kernel reuse strategy.

LU decomposition as shown in Figure 2, and discussed in Section 2. Like Gaussian elimination this will test our reuse strategy.

We ran all the benchmarks on hardware from the two major GPGPU[5] vendors, namely AMD and NVIDIA. The GPUs from the two vendors are installed in two identical machines, which are configured as shown in Table 1. The NVIDIA GPU we used is a GTX 680, and the AMD GPU is a HD7970. These two GPUs were chosen as they are similarly priced, and of the same generation of GPUs, i.e. were marketed at the same time. The specs of the GPUs are shown in Table 2. All benchmarks were run with both 32-, and 64-bit floating point data, since GPUs of this generation have very different performance in single and double preci-

[4]The index of each stencil data point will be calculates as: `gidx*vis1 + vis0`

[5]General-purpose computing on graphics processing units

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

Table 1. System specifications.

Vendor:	AMD	NVIDIA
Model:	HD 7970	GTX 680
Driver version:	1214.3 (VM)	331.38
#Cores:	2048	1536
Clock:	1000 MHz	1006 MHz
Memory:	3GB GDDR5	2GB DDR5
Memory bandwidth:	288 GB/s	192 GB/s
Peak performance:	4096 GFLOPS	3090 GFLOPS

Table 2. GPU specifications.

sion floating point arithmetic. This was done to show if the data type has any effect on the benefits from code specialization. The physical layout of the cores on the two GPUs are a little different. The NVIDIA GPU has 32 multiprocessors each of which contain 64 streaming processors, or cores. The AMD GPU has 32 multiprocessors each containing 64 cores. The important feature to consider when programming GPUs is that the hardware threads are executed in groups called wavefronts or warps. NVIDIA uses a warp size of 32 where an AMDs warp contains 64 threads. The threads within a warp are executed in lock step, a technology called Single Instruction Multiple Thread (SIMT). This means that the execution path of the threads within a warp can not diverge. If different code paths needs to be followed, i.e. given an `if` statement, all threads will need to follow both or all paths. The Bohrium byte codes do not contain loop constructs or conditionals. The strategy implemented in the GPU execution engine is to treat the GPU as a large Single Instruction Multiple Data (SIMD) machine. All GPU threads execute the same instructions even across warps.

4.2. Kernel Compilation

A key assumption for this work, is that compiling a generated OpenCL kernel is a time consuming task, compared to the execution of said kernel. This is illustrated by the application runtime of the Gaussian elimination, and LU decomposition applications, when forcing the system to use *fixed* kernels only. The application runtimes are orders of magnitude larger that the time spent executing the kernels as shown in Figure 7, 8, 9, and 10. The other main assumption is that the compilers will be able to produce better code from the *specialized* kernels — resulting in shorter execution times.

The default behaviour of the NVIDIA OpenCL driver is to cache compiled kernels in binary format to disc, so they may be reused with out recompilation if the same source code kernel is encountered again. This is all done behind the scenes, with out the need for the programmer to do any thing. When trying to compile a source code kernel which has been compiled before, the NVIDIA runtime system will simply return the previously compiled binary version, saving time. We have disabled automatic caching for this performance study, so the runtimes presented here show the performance of the implemented strategy without

influence from NVIDIAs caching strategy. The AMD runtime system does not have any automatic caching behaviour.

Kernel:	Dynamic		Fixed	
Data:	double	single	double	single
1D stencil	**725**	728	**729**	724
2D stencil	**735**	736	**732**	728
3D stencil	**773**	772	**736**	737
4D stencil	***1259***	*1257*	***2709***	*2735*
Gauss	**1105**	1094	**1104**	1089
LU	**1465**	1459	**1460**	1446

Table 3. Compile times by the NVIDIA OpenCL driver in ms.

Kernel:	Dynamic		Fixed	
Data:	double	single	double	single
1D stencil	**60.4**	60.3	**59.3**	59.2
2D stencil	**65.7**	65.4	**62.6**	62.3
3D stencil	**84.3**	83.8	**70.3**	70.0
4D stencil	***301.1***	*299.3*	***188.1***	*186.5*
Gauss	**96.9**	95.9	**94.2**	93.0
LU	**124.7**	123.5	**122.2**	121.0

Table 4. Compile times by the AMD OpenCL driver in ms.

In Table 3 and 4 we show the time spent on compiling the different types of kernels for the different applications. The compile time for the different applications is the same for the two data types used, i.e. floats and doubles. It would be surprising if the primitive data type had any influence on the time spent on compilation. This is true for both platforms.

We stated in Section 3 that we expected compile times for both kernel types, i.e. dynamic and fixed, to be very similar. As we can see in Table 3 and 4 the compile times for the 1D stencil, 2D stencil, Gaussian elimination, and LU decomposition applications are very close to the same on both the NVIDIA and the AMD platform. There is a small difference in favor of the fixed kernel for the 3D stencil application on both platforms. For the 4D stencil application on AMD we see the difference is somewhat larger, but still in favor of the fixed kernel. We can not be sure about the reason for this difference, as we do not have access to the strategies that are implemented in the two compilers. But an educated guess is: Due to some of the indexing sub expressions becoming the same, these can be replaced by a single expression. This is a relatively simple and inexpensive operation. When this is done the register allocation may become simpler, because potentially fewer registers are needed due to reuse. If one uses the defines in Figure 5 in the OpenCL kernel for the 2D-Jacobi stencil shown in Figure 4; the sub expression `gidy*2000` appears five times, `gidx*1` seven times, and `gidy*1998` two times.

The compilation time for the 4D stencil application on the NVIDIA platform stands out. The compilation time for the fixed kernel is more than twice as long as for the equivalent dynamic kernel. It is both a surprise that the difference is so large, and that it is in favor of the dynamic kernel. This fact is a disadvantage to the strategy we have chosen: Always enqueuing the fixed kernel when a new set of kernels are compiled as explained in Section 3. A better strategy in this case may be to enqueue the first available kernel, i.e. the kernel with the shortest compilation time. This would be beneficial as long as the difference the execution time for the kernel is shorter than the difference in compilation time. We do not, and can not,

know the execution time. For the benchmarks we have chosen for this work it would hold. For generality and simplicity we have chosen to stick with the previously explained strategy.

	NVIDIA	AMD	**ratio**
1D stencil	1445ms	117ms	**12.3**
2D stencil	1460ms	125ms	**11.7**
3D stencil	1503ms	151ms	**9.9**
4D stencil	3993ms	482ms	**8.3**
Gauss	2199ms	186ms	**11.8**
LU	2929ms	241ms	**12.2**
Average			***10.4***

Table 5. Comparison of combined compile times for both dynamic and static kernels by the two vendors.

The AMD driver's OpenCL compiler is much faster than NVIDIA's. On average it is 10 times faster as we have shown in Table 5. This may be due to NIVIDA doing more complex optimizations on the generated code. The difference in compilation times between the two vendors is probably a factor in that NVIDIA has implemented kernel caching, and AMD has not.

4.3. Results

The four Figures 7, 8, 9, and 10 show the application runtime for the two hardware vendors NVIDIA and AMD, running the benchmarks with both single- and double precision floating point data. We have run all six benchmarks on all combinations of vendor, data type and the three kernel generation strategies, i.e. dynamic, fixed, and selected. We show both the time spent by the GPU actually executing kernels, in a darker shade, and the over all application run time. It is clear to see from all four figures that only generating fixed kernels is not a viable solution. The application run times for the Gaussian elimination and LU decomposition benchmarks are orders of magnitude longer for the fixed kernel only strategy than for the dynamic kernel only. The significantly contributing factor is the kernel compilation overhead.

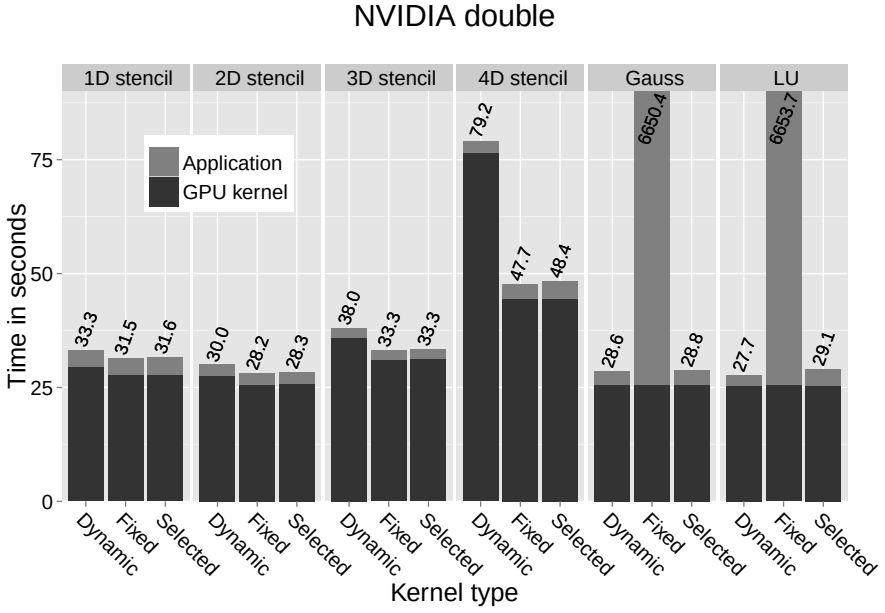


Figure 7. Application run time on NVIDIA for double precision data.

Looking at the execution times on the NVIDIA GPU for double precision data in Figure 7: We see the overall picture we were expecting. The four stencil applications benefit from the specialization. They benefit to an increasing degree with increased dimensionality. Which is what one would expect with the increasing number of index calculations required per data element, that needs to be fetched or stored. Table 6 shows the speedup of our *selected* kernel strategy over the *dynamic* kernel only strategy. We consider the dynamic kernel strategy the basis. We can see that there is no difference in the speedup of the 1D- and 2D- stencil applications, which is a little surprising.

Data type:	double		single	
	Application	Kernel	Application	Kernel
1D stencil	**1.05**	1.06	**1.01**	1.01
2D stencil	**1.06**	1.07	**1.02**	1.02
3D stencil	**1.14**	1.15	**1.01**	1.01
4D stencil	**1.64**	1.72	**2.49**	2.64
Gauss	**0.99**	1.00	**1.00**	1.00
LU	**0.95**	1.00	**0.97**	1.00

Table 6. Speed up on NVIDA GTX 680.

Turning to the single precision benchmarks, Table 6 and Figure 8, we see that there is no significant benefit from code specialization in the 1D – 3D stencil benchmarks. While we see an even greater speedup in the 4D stencil application. It is reasonable to assume that this performance increase is due to the fact that the GTX 680 is able to perform 24 single precision floating point operations for every double precision operation. Which in turn means that the integer index calculations will account for a relatively larger portion of the time spent executing the GPU kernels — thus increasing the relative benefit from code specialization. We also attribute the lack of speedup in the 1D – 3D stencil benchmarks to the 1:24 performance ratio between single- and double precision calculations. When the calculations on the data become that much faster, the memory band width becomes the limiting factor, and we are moving the same amount of data independent of kernel strategy.

There is a small loss in performance in the 4D stencil and the LU decomposition benchmarks, see Table 6. These are also the benchmarks with the longest compile times, see Table 3. There is reason to believe that the added compile time overhead would be amortizes in applications with longer execution times.

Data type:	double		single	
	Application	Kernel	Application	Kernel
1D stencil	**1.00**	1.01	**1.01**	1.02
2D stencil	**1.00**	1.00	**1.24**	1.52
3D stencil	**1.19**	1.30	**2.18**	3.23
4D stencil	**1.02**	1.02	**1.80**	2.20
Gauss	**1.00**	1.00	**1.00**	1.00
LU	**1.00**	1.00	**1.01**	1.00

Table 7. Speed up on AMD HD 7970.

When looking at the results for the AMD GPU running the benchmarks in Table 7 we see that it is mainly the single precision benchmarks that benefit from the code specialization. Again we see an increased benefit from specialization going from 1D- to 2D- and on to the 3D stencil benchmark. However, there is a drop off when going to the 4D stencil. In the 4D stencil benchmark we do introduce a loop in the kernel which is not needed for lower dimensionality.

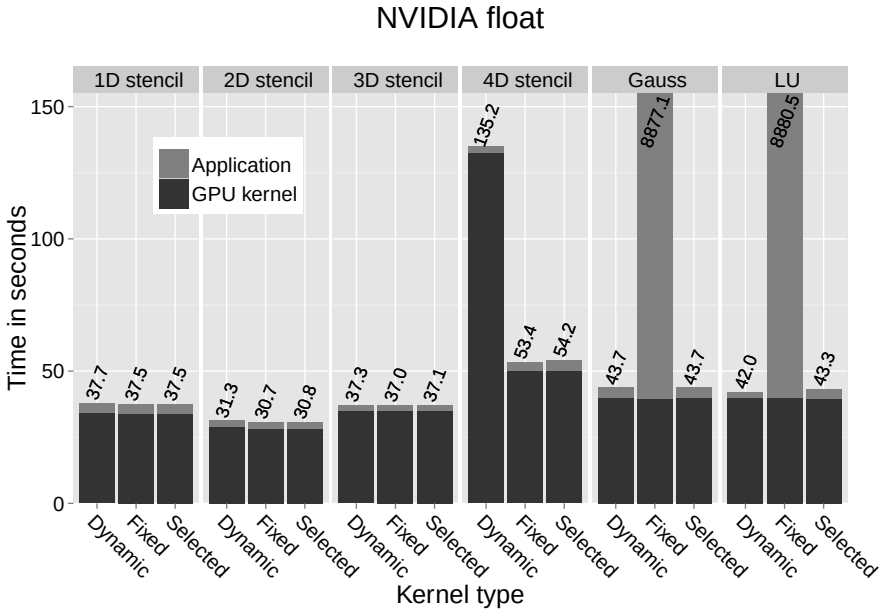


Figure 8. Application run time on NVIDIA for single precision data.

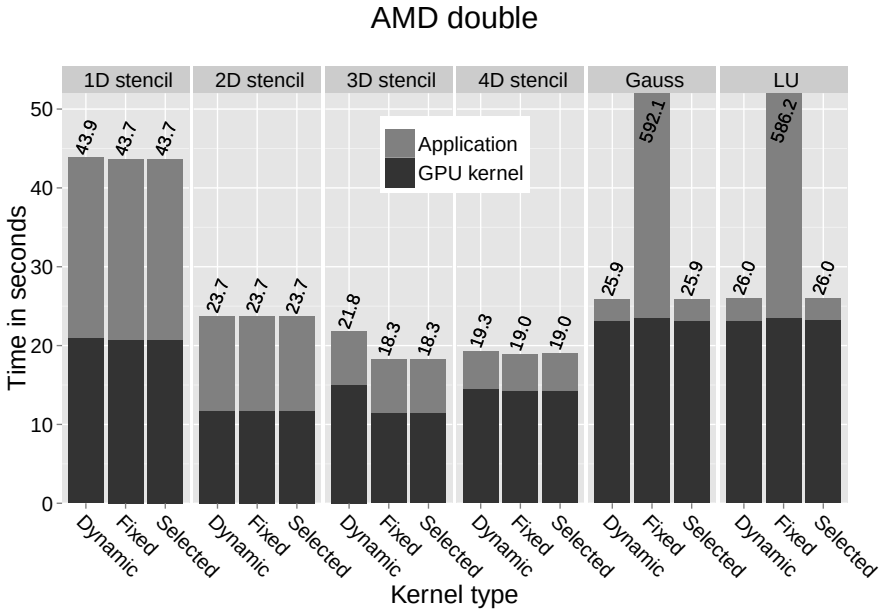


Figure 9. Application run time on AMD for double precision data.

Both the OpenCL and CUDA programming models support up to three dimensions natively. Going above three dimension necessitates the introduction of loops. The loop construct is present in both the static and dynamic kernel, so in it self it should not make a difference. It is not clear at this point if the introduction of a loop in the kernel is the reason for the reduced speedup in the 4D stencil benchmark. Why we do not see any benefits from the specialization when using double precision data on the 1D-, 2D-, and 4D stencil is not clear to us, and warrants further investigation. On the AMD platform the performance penalty for the extra kernel generation in the Gaussian elimination and LU decomposition benchmarks are so small that they do not show in the runtime graphs, see the Dynamic and Selected columns for the two benchmarks in Figure 9 and 10.

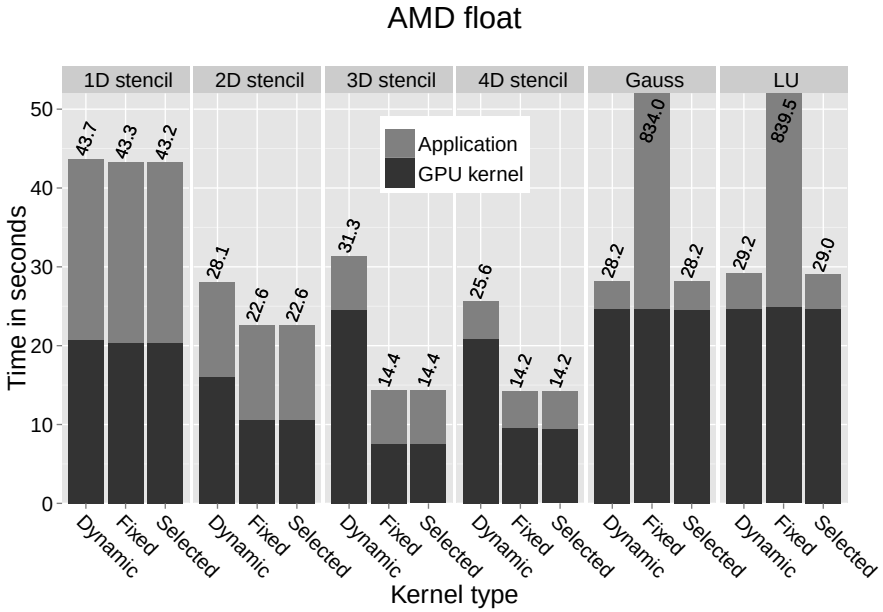


Figure 10. Application run time on AMD for single precision data.

Figure 9 and 10 show that there is a significant overhead in the AMD driver, and that it is correlated to the number of kernel calls. For the 1D stencil only half the total application runtime is spent executing kernels. Most of this exes time is spent by the AMD driver, not in the collected Bohrium system, otherwise we would see the same overhead in the benchmarks running on NVIDIA hardware.

5. Future Work

We have shown, in this work, that there are clear benefits to be gained from specialization of GPU-kernels, and that the added costs of compiling and managing the extra kernels are quickly amortized. We are convinced that there is basis for doing more advanced kernel analysis and specialization. This would require a deeper analysis of the incoming bytecode and even compiling more than two versions of the operational-wise same kernel. We envision analyzing exactly which parameters change over the course of several iterations. Potentially it would even be beneficial to analyze which parameters change synchroniously — simply to limit the function parameter space used. Available parameter space is a limited resource on GPU's, especially when moving to higher dimensionality, or when generation more complex kernels.

6. Conclusion

In this work, we show that both specialized-, and parametrizied- kernels have their benefits and recognizable use cases. We have implemented a simple hybrid method and show that it is possible to reap the benefits of both approaches in the realm of auto generated kernels. We have achieved these benefits without any significant negative impact on overall application performance. Even in the cases where we were not able to gain any performance boost by specialization the added cost, for kernel generation and extra bookkeeping, is minimal.

References

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele, and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., March 2008. Version 1.0.

[2] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[3] F.T. Winter. QDP-JIT/PTX: A QDP++ Implementation for CUDA-Enabled GPUs. *PoS*, LATTICE2013:042, 2014.

[4] Robert G. Edwards and Balint Joo. The Chroma software system for lattice QCD. *Nucl.Phys.Proc.Suppl.*, 140:832, 2005.

[5] CAPS Enterprise, Cray Inc., NVIDIA, and the Portland Group. *The OpenACC™ Application Programming Interface (v2.0)*. OpenACC-Standard.org, June 2013.

[6] Nathan Bell and Jared Hoberock (NVIDIA). Thrust: A productivity-oriented library for cuda. `https://developer.nvidia.com/Thrust`, 2012.

[7] AMD. Bolt: C++ template library with support for opencl. `http://developer.amd.com/tools-and-sdks/opencl-zone/bolt-c-template-library`, 2013.

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

[9] Microsoft Corporation. C++ amp : Language and programming model, 2012.

[10] Jeffrey M Perkel. Programming: pick up python. *Nature*, 518(7537):125–126, 2015.

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

[12] Travis E Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[13] Bryan Catanzaro, Shoaib Ashraf Kamil, Yunsup Lee, Krste Asanović, James Demmel, Kurt Keutzer, John Shalf, Katherine A. Yelick, and Armando Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.

[14] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[15] CUDA Nvidia. Programming guide, 2008.

[16] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157 – 174, 2012.

[17] Aaftab. Munshi et al. *The OpenCL Specification (v1.1)*. Khronos OpenCL Working Group, 2011.

[18] 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.

[19] 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.

[20] 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.

7.7 Fusion of Array Operations at Runtime

Mads R. B. Kristensen, Troels Blum, Simon A. F. Lund, and James Avery.
To be submitted.

Fusion of Array Operations at Runtime

Mads R. B. Kristensen, Troels Blum, Simon A. F. Lund, 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 effiect 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: $cost(P) = cost_{\text{MWC}}(\mu, P)$.

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, A, 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. 10)*
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. 11) 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.

However, before we formally define this data-parallelism property, we must introduce some notation:

Definition 9. *Given an array operation f, the notation $in[f]$ denotes the set of arrays that f reads; $out[f]$ denotes the set of arrays that f writes; and $del[f]$ denotes the set of arrays that f deletes (or memory de-allocates).*

Let us define the data-parallelism property and thus the criteria for array operation fusion:

Definition 10. *An array operation, f, in Bohrium has the data-parallelism property where each output element can be calculated independently, which imposes the following restrictions:*

$$\begin{gathered}\forall i \in in[f], \forall o \in out[f], \forall o' \in out[f] : \\ i \cap o = \emptyset \vee o = o' \wedge o \cap o' = \emptyset \vee o = o'\end{gathered} \tag{4}$$

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:*

$$\begin{gathered}\forall i' \in in[f'], \forall o \in out[f] : i' \cap o = \emptyset \vee i' = o \\ \wedge \\ \forall o' \in out[f'], \forall o \in out[f] : o' \cap o = \emptyset \vee o' = o \\ \wedge \\ \forall o' \in out[f'], \forall i \in in[f] : o' \cap i = \emptyset \vee o' = i\end{gathered}$$

Proof. It follows directly from Definition 10. □

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 11. *In bohrium, the cost of a partition, $P = \{B_1, B_2, ..., B_k\}$, of array operations is given by:*

$$0 \leq cost(P) = \sum_{B \in P} length\left(\bigcup_{f \in B} (in[f] \cup out[f])\right) \tag{6}$$

where $length(A)$ returns the total number of bytes accessed by the unique arrays in A. A exception to the cost function: the cost of `DEL` and `SYNC` is always zero.

Bohrium implements two techniques to improve data locality through array operation fusion:

[2]For a detailed description of this Python-to-bytecode translation we refer to previous work [7], [8].

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) is given by:*

$$0 \leq saving(f, f') = \\ length(out[f] \cap del[f']) + length(out[f] \cap in[f']) \tag{7}$$

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 $length(out[f] \cap del[f'])$ bytes. Meanwhile, Data Access Reuse is applicable when an output of f is also an input of f', which saves us $length(out[f] \cap in[f'])$ bytes. Thus, we have a total saving of $length(out[f] \cap del[f']) + length(out[f] \cap in[f'])$. □

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. 11). □

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 12. *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 ˆ 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. 12) 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 13. *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. □

```
1: function IGNORE(G, e)
2:     (u, v) ← e
3:     l ← 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.

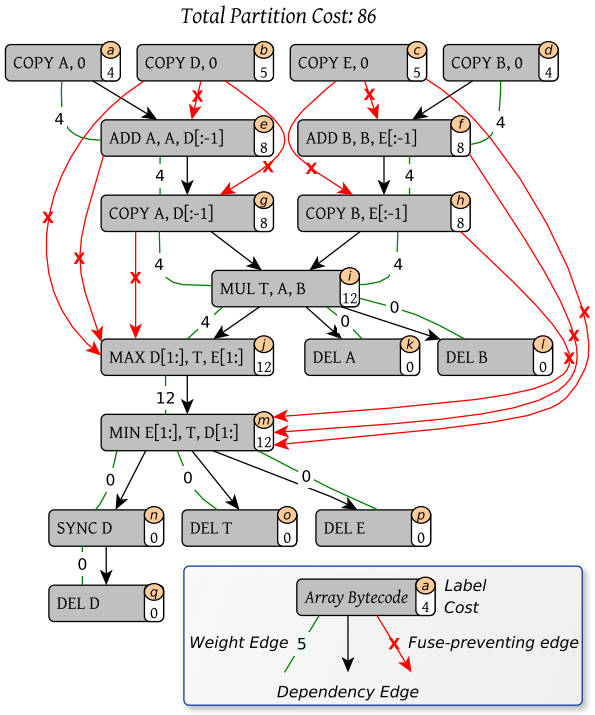


Fig. 5. A partition graph of the Python application in Fig. 3. For illustrative proposes, the graph does not include ignored weight edges (cf. Fig. 4).

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 $O(V + E)$.

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

```
1: function GREEDY(G)
2:     while E_w[G] ≠ ∅ do
3:         (u, v) ← FIND-HEAVIEST(E_w[G])
4:         if IGNORE(G, (u, v)) then
5:             Remove edge (u, v) from E_w
6:         else
7:             G ← FUSE(G, u, v)
8:         end if
9:     end while
10:    return G
11: end function
```

Fig. 6. The greedy fusion algorithm that greedily fuses the vertices connected with the heaviest weight edge in G.

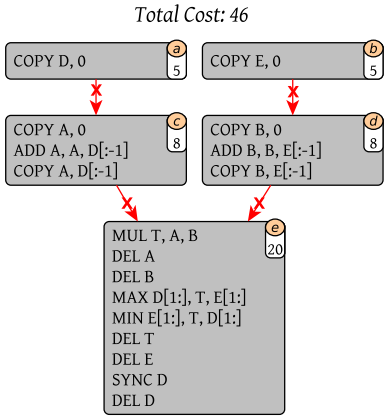


Fig. 7. A partition graph of the greedy fusion of the graph in Fig. 5.

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 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 $(V[\hat{G}], \{e \in E_w[\hat{G}] | \neg \text{IGNORE}(\hat{G}, e)\})$, 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.*

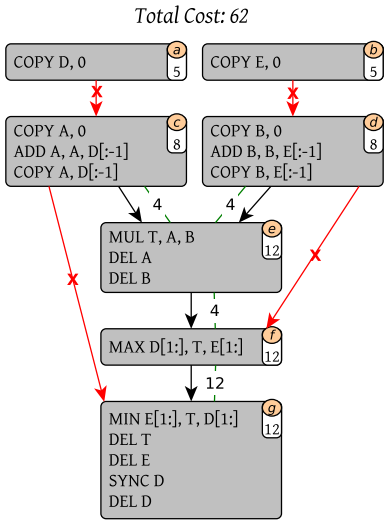


Fig. 8. A partition graph of the unintrusive fusion of the graph in Fig. 5.

Proof. The sequences of fusions that obtain an optimal partition solution cannot include the fusion of u, v into z when 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.

```
1: function FindCandidate(G)                          ▷ Help function
2:     for (v, u) ← E_w[G] do
3:         if Ignore(G, (u, v)) then
4:             Remove edge (u, v) from E_w
5:         end if
6:     end for
7:     for (v, u) ← E_w[G] do
8:         if the degree is less than 2 for either u or v
           when only counting edges in E_w[G] then
9:             if θ[u] = θ[v] then
10:                return (u, v)
11:            end if
12:        end if
13:    end for
14:    return (NIL, NIL)
15: end function
16:
17: function Unintrusive(G)
18:    while (u, v) ← FindCandidate(G) ≠ (NIL, NIL) do
19:        G ← Fuse(G, u, v)
20:    end while
21:    return G
22: end function
```

Fig. 9. The unintrusive fusion algorithm that only fuse *unintrusively fusible* vertices.

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, 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.

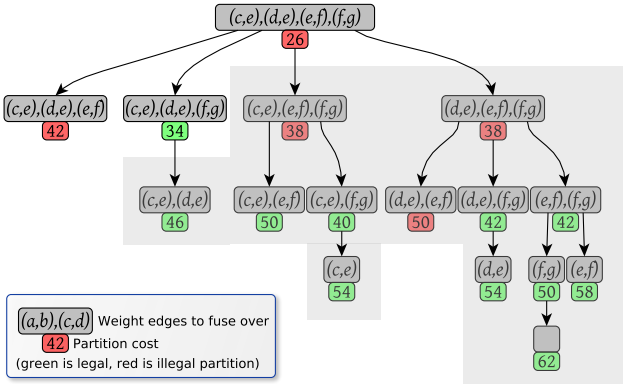


Fig. 10. A branch-and-bound search tree of the unintrusively fused partition graph (Fig. 8). Each vertex lists a sequences of vertex fusions that build a specific graph partition. The grayed out area indicates the part of the search tree that a depth-first-search can skip because of the cost bound.

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. 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

[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`.

```
1: function FuseByMask(G, M)                          ▷ Help function
2:     f ← true                                       ▷ Flag that indicates fusibility
3:     for i ← 0 to |E_w[G]| − 1 do
4:         if M_i = 1 then
5:             (u, v) ← the i'th edge in E_w[G]
6:             if not Fusible(G, u, v) then
7:                 f ← false
8:             end if
9:             G ← Fuse(G, u, v)
10:        end if
11:    end for
12:    return (G, f)
13: end function
14:
15: function Optimal(G)
16:    G ← Unintrusive(G)
17:    for (v, u) ← |E_w[G] do
18:        if Ignore(G, (u, v)) then
19:            Remove edge (u, v) from E_w
20:        end if
21:    end for
22:    B ← Greedy(G)                                  ▷ Initially best partitioning
23:    M_{0..|E_w[G]|} ← 1                            ▷ Fill array M with ones
24:    o ← 0                                          ▷ The mask offset
25:    Q ← ∅
26:    Enqueue(Q, (M, o))
27:    while Q ≠ ∅ do
28:        (M, o) ← Dequeue(Q)
29:        (G′, f) ← FuseByMask(G, M)
30:        if cost(G′) < cost(B) then
31:            if f and G′ is acyclic then
32:                B ← G′                             ▷ New best partitioning
33:            end if
34:        end if
35:        for i ← o to |M| − 1 do
36:            M′ ← M
37:            M′_i ← 0
38:            Enqueue(Q, (M′, i + 1))
39:        end for
40:    end while
41:    return B
42: 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.

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. 11) 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 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.

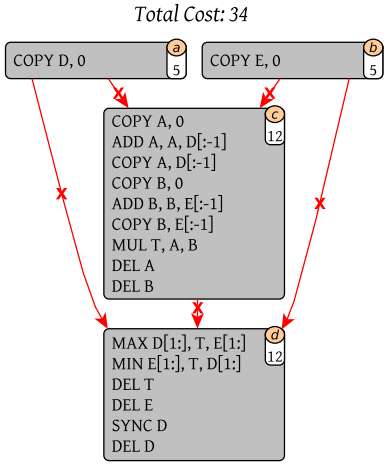


Fig. 12. A partition graph of the optimal fusion of the graph in Fig. 5.

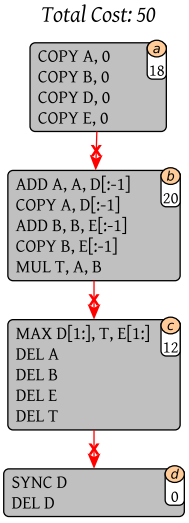


Fig. 13. A partition graph of a Naïve partition of the Python example (Fig. 3).

Benchmark	Input size (in 64bit floats)	Iterations
Black Scholes	1.5×10^6	20
Game of Life	10^8	20
Heat Equation	1.44×10^8	20
Leibnitz PI	10^8	20
Gauss Elimination	2800	2799
LU Factorization	2800	2799
Monte Carlo PI	10^8	20
27 Point Stencil	4.2875×10^7	20
Shallow Water	1.024×10^7	20
Rosenbrock	2×10^8	20
Successive over-relaxation	1.44×10^8	20
NBody	6000	20
NBody Nice	40 plantes, 2×10^6asteroids	20
Lattice Boltzmann D3Q19	3.375×10^6	20
Water-Ice Simulation	6.4×10^5	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

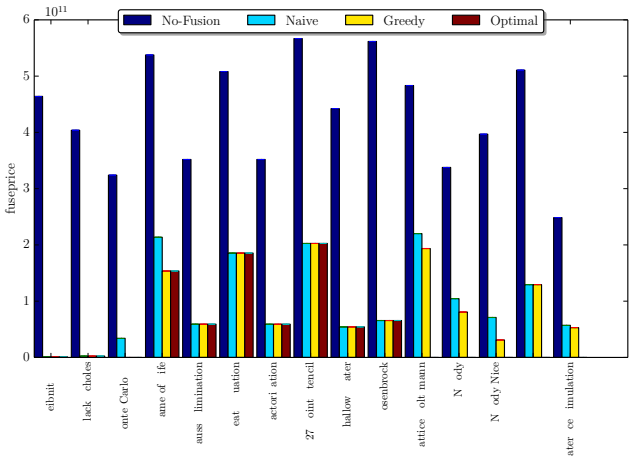


Fig. 14. Theoretical cost of the different partition algorithms. NB: the last five benchmarks, Lattice Boltzmann, NBody, NBody Nice, SOR, Water-Ice Simulation, does not show an optimal solution.

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. 11), 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

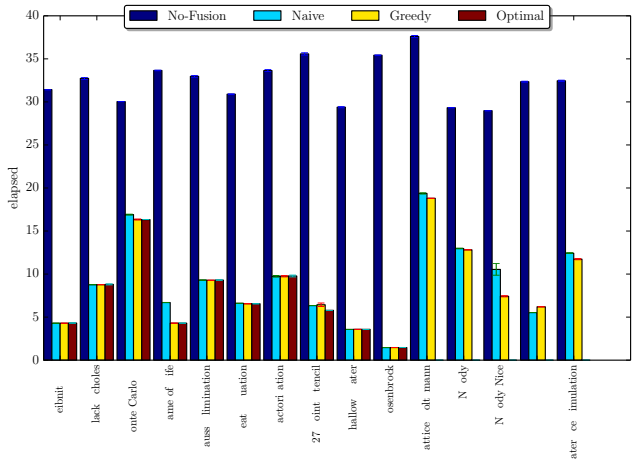


Fig. 15. Runtime of the different partition algorithms using a **warm cache**.

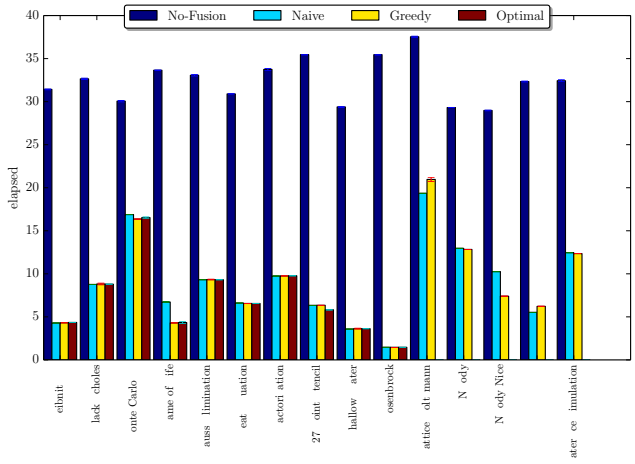


Fig. 16. Runtime of the different partition algorithms using a **cold cache**.

cache (Fig. 17), the Naïve partition algorithm outperforms both the Greedy and Optimal algorithm because of its smaller time complexity.

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.

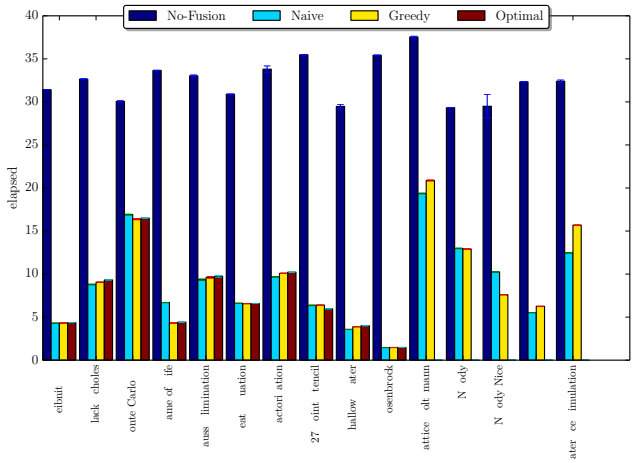


Fig. 17. Runtime of the different partition algorithms using **no cache**.

[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.

Bibliography

[1] Comparing python, numpy, matlab, fortran, etc. https://modelingguru.nasa.gov/docs/DOC-1762. [Online; accessed April 7. 2014].

[2] Java - Class Thread. http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.html. [Online; accessed 7 September 2015].

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

[4] Python - Higher-level threading interface. http://docs.python.org/2/library/threading.html. [Online; accessed 7 September 2015].

[5] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide. Technical report, 2007. [Online; accessed 21 February 2013].

[6] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele, and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., March 2008. Version 1.0.

[7] AMD. Bolt: C++ template library with support for opencl. http://developer.amd.com/tools-and-sdks/opencl-zone/bolt-c-template-library, 2013.

[8] Rasmus Andersen and Brian Vinter. The scientific byte code virtual machine. In *GCA'08*, pages 175–181, 2008.

[9] 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.

[10] Nathan Bell and Jared Hoberock (NVIDIA). Thrust: A productivity-oriented library for cuda. `https://developer.nvidia.com/Thrust`, 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] 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.

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

[14] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[15] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanovic, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. *Programming Models for Emerging Architectures*, 2009.

[16] Bryan Catanzaro, Shoaib Ashraf Kamil, Yunsup Lee, Krste Asanović, James Demmel, Kurt Keutzer, John Shalf, Katherine A. Yelick, and Armando Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.

[17] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[18] Intel Corporation. *80386 programmer's reference manual*. Intel Corporation, 1986.

[19] Intel Corporation. *80286 And 80287 Programmer's Reference Manual, revised*. Intel Corporation, 1987.

[20] Microsoft Corporation. C++ amp : Language and programming model, 2012.

[21] NVIDIA Corporation. Nvidia cuda c programming guide (version 3.2). `http://developer.nvidia.com/page/home.html`, August 2010.

[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] Robert G. Edwards and Balint Joo. The Chroma software system for lattice QCD. *Nucl.Phys.Proc.Suppl.*, 140:832, 2005.

[24] CAPS Enterprise, Cray Inc., NVIDIA, and the Portland Group. *The OpenACC^{TM} Application Programming Interface (v2.0)*. OpenACC-Standard.org, June 2013.

[25] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 216–225, Washington, DC, USA, 2011. IEEE Computer Society.

[26] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[27] Rahul Garg and José Nelson Amaral. Compiling python to a hybrid execution environment. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 19–30, New York, NY, USA, 2010. ACM.

[28] S. Gill. Parallel programming. *The Computer Journal*, 1(1):2–10, 1958.

[29] M.J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 109–118. Eurographics Association, 2002.

[30] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[31] 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*, pages 364 –373, may 1990.

[32] A. A. Khan, C. L. McCreary, and M. S. Jones. A comparison of multiprocessor scheduling heuristics. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 02*, ICPP '94, pages 243–250, Washington, DC, USA, 1994. IEEE Computer Society.

[33] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157 – 174, 2012.

[34] 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.

[35] 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.

[36] 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.

[37] Mads Ruben Burgdorff Kristensen and Brian 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.

[38] M.R.B. Kristensen, S.A.F. Lund, T. Blum, and B. 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.

[39] M.R.B. 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.

[40] 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*, pages 75–86. IEEE, 2004.

[41] 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.

[42] D.B. Loveman. High performance fortran. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(1):25–42, 1993.

[43] 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.

[44] B.J. Mailloux, J.E.L. Peck, and C.H.A. Koster. Report on the algorithmic language algol 68. *Numerische Mathematik*, 14(2):79–218, 1969.

[45] Volodymyr Mnih. Cudamat: a cuda-based matrix class for python. *Department of Computer Science, University of Toronto, Tech. Rep. UTML TR*, 4, 2009.

[46] Jeffrey Morlan. *Auto-tuning the Matrix Powers Kernel with SEJITS*. PhD thesis, MA thesis. EECS Department, University of California, Berkeley, 2012. URL: http://www. eecs. berkeley. edu/Pubs/TechRpts/2012/EECS-2012-95. html, 2012.

[47] Aaftab Munshi et al. The OpenCL Specification. *Khronos OpenCL Working Group*, 1:l1–15, 2009.

[48] Aaftab. Munshi et al. *The OpenCL Specification (v1.1)*. Khronos OpenCL Working Group, 2011.

[49] C.J. Newburn, Byoungro So, Zhenying Liu, M. McCool, A. Ghuloum, S.D. Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan 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*, pages 224–235, 2011.

[50] CUDA Nvidia. Programming guide, 2008.

[51] Travis E Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[52] Travis E. Oliphant. Python for Scientific Computing. *Computing in Science and Engineering*, 9:10–20, 2007.

[53] Jeffrey M Perkel. Programming: pick up python. *Nature*, 518(7537):125–126, 2015.

[54] SN Rao, EV Prasad, and NB Venkateswarlu. A critical performance study of memory mapping on multi-core processors: An experiment with k-means algorithm with large data mining data sets. *International Journal of Computer Applications IJCA*, 1(9):90–98, 2010.

[55] 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.

[56] 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.

[57] Mark Segal and Kurt Akeley. The OpenGL Grapichs System: A Specification (version 2.0). `https://www.opengl.org/documentation/specs/version2.0/glspec20.pdf`, 10 2004. [Online; accessed 12 March 2013].

[58] Kenneth Skovhede and Brian Vinter. NumCIL: Numeric operations in the Common Intermediate Language. *Journal of Next Generation Information Technology*, 4(1), 2013.

[59] 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.

[60] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

[61] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339, November 1990.

[62] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. *SIGARCH Comput. Archit. News*, 34(5):325–335, October 2006.

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

[64] Tijmen Tieleman. Gnumpy: an easy way to use gpu boards in python. 2010.

[65] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[66] 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.

[67] Guido van Rossum. Glue it all together with python. In *Workshop on Compositional Software Architectures, Workshop Report, Monterey, California*, 1998.

[68] 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.

[69] F.T. Winter. QDP-JIT/PTX: A QDP++ Implementation for CUDA-Enabled GPUs. *PoS*, LATTICE2013:042, 2014.

[70] W.Y. Yang, W. Cao, T.S. Chung, and J. Morris. *Applied numerical methods using MATLAB*. Wiley-Interscience, 2005.