UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

MSc in Computational Physics

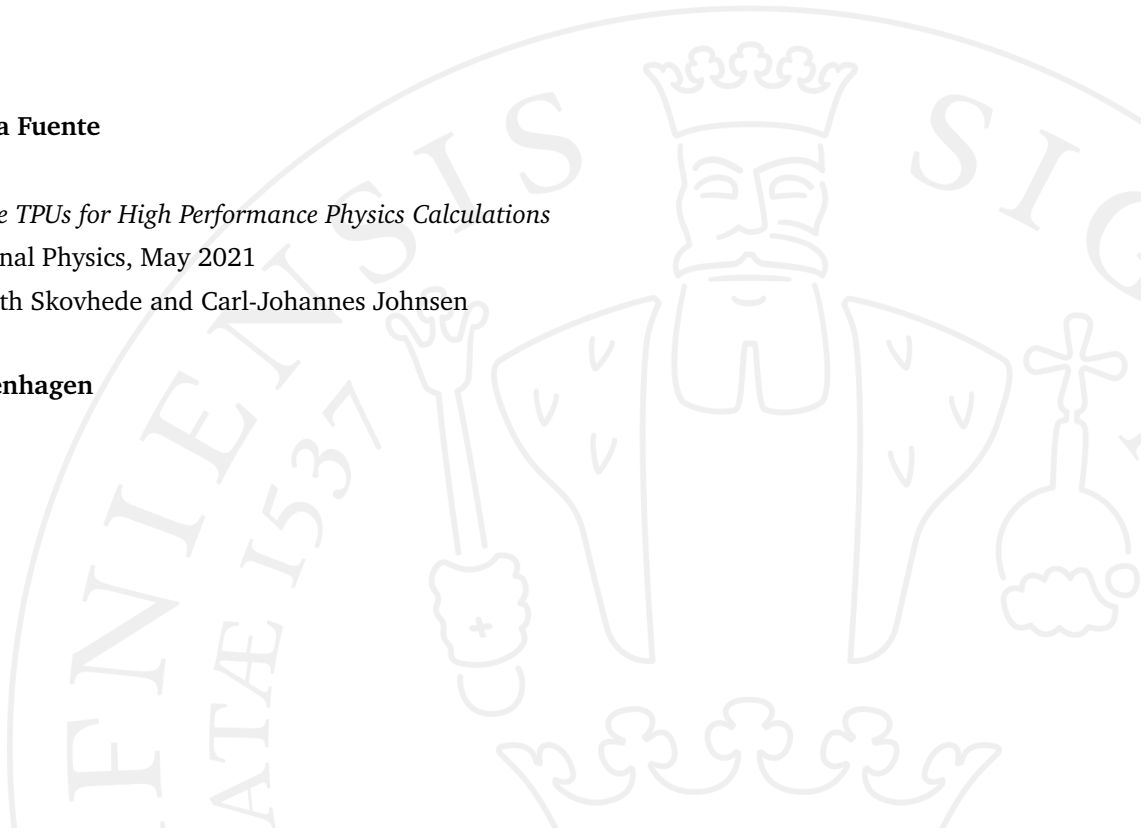
Evaluation of Google TPUs for High Performance Physics Calculations

Albert Alonso de la Fuente
aaf@di.ku.dk
Supervised by Kenneth Skovhede & Carl-Johannes Johnsen

May 2021

Albert Alonso de la Fuente
aaf@di.ku.dk
Evaluation of Google TPUs for High Performance Physics Calculations
MSc in Computational Physics, May 2021
Supervisors: Kenneth Skovhede and Carl-Johannes Johnsen

University of Copenhagen
Faculty of Science
Niels Bohr Institute
Blegdamsvej 17



Acknowledgements

I would like to express my gratitude to the *eScience* department, or at least the people I had the chance to meet during this chaotic year. Special thanks to Kenneth Skovehede and Carl Johnsen for their constant support and help, as well as to the Veros Friday meeting for being a constant remainder that working from home did not mean working alone. I would like to specially thank Dion Hafner for every useful comment and suggestion he has given me during this thesis.

I am grateful to the *JAX-on-TPU* team that granted me access to the new alpha TPU-VMs and replied to my queries with very insightful and extensive answers. On a similar note, I would also like to thank the TensorFlow Research Foundation for providing me with free access to TPUs and being extremely helpful whenever I made a mistake, such as expending 300$ on a single night by forgetting to turn off the machine. Without them this thesis would not have happened, literally.

Finally, I am grateful to my family, whose expertise on theoretical physics and complex mathematical abstractions have always been very insightful, and specially I want to thank Mireia, without whom I would not have finished this thesis.

Thank you.

Abstract

This thesis evaluates the use of novel AI chips as a solution for high performance large scale scientific simulations. By looking for fast and energy efficient accelerators, we assess the repercussions of porting physics calculations to Matrix Engines based accelerators, purposely designed to run efficiently the linear algebra found on Deep Learning workflows.

In our study, we focus on the use of Google's in-house Tensor Processing Units (TPU) as well as Google's machine learning research programming library JAX, due to it's versatility of backends and its similarity to the most used numerical python libraries. We present an alternative method to compute finite difference derivatives that leverages the matrix operation capabilities of TPUs outperforming by 2 times the performance of conventional vector approaches. Simulations whose main computing part mostly consists of matrix multiplications are found to be up to 3 times faster when used on a single core as opposed to their performance on a entire Graphical Processing Units.

We reproduce an implementation of the Ising Model developed on TensorFlow using our approach on JAX which indicates that the XLA compiler may performs better when used on TensorFlow graphs, despite JAX providing a more readable and familiar code.

Finally, we consider the viability of porting a general circulation model to make efficient use of TPUs while we outline the reasons to consider Matrix Engines as a viable scalable solution for certain type of physical simulations.

Contents

1 Introduction

Simulations based on physical phenomena are heavily used both in academia as well as industry and most of the times rely on classical numerical methods to find approximate finite solutions to continuous problems. Therefore, it is common to look for novel solutions that can allow these computations to be run faster and more efficiently.

Over the last decades, the High-Performance Computing (HPC) community has embraced the use of highly parallel accelerators, such as Graphic Processing Units (GPUs), to outperform conventional Central Processing Units (CPUs) by exploiting their massive parallel capabilities [1]. Despite their compelling performance improvement over more classical general purpose systems, GPUs still present certain inefficiencies to the scientific workflow that the community is actively trying to overcome [2]. In particular, one of the main issues of these inefficiencies is the large amount of electricity they consume, which makes data centers very expensive to maintain.

Several proposed solutions rely on the use of application-specialization hardware [3]. Despite not being a novel approach, since Field Programmable Gate Arrays (FPGAs) have been used effectively on scientific workloads for years [4], it does not mean that they are the best candidate to replace the current used processing units. For instance, FPGAs also require domain specific knowledge in order to be programmed efficiently and therefore do not present the commodity that current GPUs workloads do.

In recent years, the Artificial Intelligence (AI) field has seen a tremendous increase on popularity, on academia as well as on the industry [5]. The most prominent reason of its stunning performance, and hence, its demand, is due to their impressive results found using Deep Learning (DL) or Deep Neural Networks (DNN) [6]. DL workloads are characterized by a massive use of matrix operations during training, and the use of GPUs has also been proved to be more efficient than classical computing units [7, 8]. Nevertheless, due to the massive amount of data and computing resources that the Machine Learning (ML) workflows are exposed to, the ML community has found themselves on a

similar position as the scientific computing community, looking for novel solutions that can perform similarly to GPUs on a more power efficient way. This has also lead them to propose Application-Specific Integrated Circuits (ASIC) solutions rather successfully [9].

Since DL is the most resource intensive ML workflow, it is to expect that the most popular candidates for novel hardware are indeed targeting DL core computation. The most prominent calculation in DL processes is the General Matrix Multiplication (GEMM). A common approach in order to compute them, is to create Matrix Engines (ME) based on systolic arrays [10, 11]. Fortunately, GEMM is also a linear algebra operation commonly found in many scientific programs [12].

Due to their high cost of production, the proposed ASIC solutions tend to be general-purpose enough to cater to the diversity ML workflows might require. Additionally, they are also accompanied with high productivity array-based programming frameworks for high level programming languages such as Python/Numpy, Julia, R, etc. since the industry keeps increasing and data scientists tend to develop proficiency on those languages. Analogously, due to the vast amount of scientific libraries available on those high level languages, most physicist are also comfortable on these array-based programming frameworks. In this regard, the AI community is ahead of the scientific community as it still is common to port Matlab/Numpy scientific simulations to low level languages such as C/C++ or Fortran in order to gain speedups of orders of magnitudes. However, is it also common that these portings are inefficiently implemented and with the cost of large amount of programming time for the scientist [13]. Therefore, a scalable solution, such as the one usually found on ML libraries, where no porting to a low level languages is required but still achieving large speedups, would meant a more reliable, reproducible and fast development of scientific solutions. Hence, it begs the question: Can the scientific computing community get the advantage of using industry-backed new developments to perform their numerical simulations, or should the AI chips remain on the ML field?

The aim of this thesis is to give an insight on whether the use of novel AI chips has a place on large-scale long-running physics simulations. Focusing on the performance of partial differential equations (PDEs) solvers and stencils operations. It is important to indicate that our study will be used on the use of Google Tensor Processing Units (TPUs)[14] and the JAX programming library [15], which serves as a frontend to the high performance XLA compiler[16], but the general results and ideas are expected to be applicable to other ME hardware accelerators and ML libraries, as there is a tendency to have similar features overall.

The research will focus on the following questions in order to reach a conclusion:

- How difficult it is to make use of this novel hardware for non ML related calculation?

- What advantages and disadvantages does this new programming model has?
- How are long running simulations affected by the lower precision data types found on ML accelerators?
- Are novel AI chips the solution to scale physics prototypes?

In order to answer them, an evaluation on the performances of common numerical calculations will be carried, specifically in the finite difference method for PDE solvers. Additionally, in an attempt to see how extensible to other types of simulations the proposed methods are, an updated implementation of an Ising model will be made to compare with their existing performance results.

Thesis outline

The thesis structure is as follow.

On Chapter 2, an introduction to the current state on AI ASIC hardware accelerators will be done as well as brief contextualization of the hardware specifications of Google TPUs. Chapter 3 begins by examining the advantages and disadvantages found on JAX, the chose programming language, and ends with the current methodology to carry use and benchmark TPU programs. Chapter 4 will deal with the most common approach found on PDE solvers as well as some proposed methods to make a better utilization of the hardware. Each method will be accompanied with benchmarks and discussions of their strength and weakness. Chapter 5 will focus on validating results obtained on previous studies as well as comparing them to the results obtained with our proposed methods and tools. Since the frameworks and hardware used are still on alpha state and therefore are subject to drastic changes, having an older paper to compare with gives us the change to review improvement in performance of the compiler used over the last couple of years. Lastly and based on the observation from the results, I will evaluate the viability of porting an in-house ocean simulation to use TPUs efficiently, as well as provide a general guidance for evaluating new translations.

2 Hardware Accelerators

General processing units such as CPUs are great due to their vast versatility which allow them to perform the operations needed to run a wide range of applications. However, having to cater a large diversity of programs does presents some inefficiencies that are not ideal for simple repetitive workloads. One of the reasons for this lack of efficiency is due to performing the fetching and interpreting of instructions constantly, which may end up costing thousands of times more than the what is actually required to compute it [17].

These inefficiencies however have not supposed a large inconvenience for the industry as up until recently, in order to overcome the large overhead one could simple wait until the performance and efficiency CPUs increased in accordance to Moore's Law [18]. The persistent increase in performance has facilitated the development of additional units, such as the ones capable of performing Single Instruction Multiple Data (SIMD) instructions. The relevance of SIMD operations is that they can be used to perform vector operations to a memory block instead of repeating the same instruction, which has to be fetched, to each element individually. These units are the reason why array-based frameworks such as NumPy and Matlab can achieve execution times on the same order of magnitude as high performance low level compiled languages such as Fortran and C++. Nevertheless and despite discussion on whether Moore's Law is approaching its end of life [19], it is clear that the price increase of producing transistor-denser units makes it harder to depend on them. For that reason, engineers and system architects are actively looking into alternatives to the von Neumann architecture, the model that current CPUs are based on, with most of the interest and successful results appearing in domain specific acceleration options [20].

2.1 Domain Specific Accelerators

We define domain specific accelerators (DSA) or hardware accelerators as those physical computing engines specialized on performing specific tasks for a particular domain

efficiently. We will briefly introduce the three most common accelerators to establish the current situation and understand the reasoning behind the recent surge of these novel accelerators as proposed solutions to continue the steady improvement in computing performance of the recent decades.

2.1.1 Graphical Processing Units

Originally designed to render images on screen, Graphical Processing Units (GPUs) are specialized on performing element-wise floating point operations in order to represent 3D graphics on 2D surfaces. This made GPUs massive parallel computing units capable of performing basic operations on thousands of threads simultaneously. Currently, the number of threads on a single GPU can range from the hundreds to the ten thousands. Figure 2.1 shows a simplistic generic scheme of a GPU in comparison with the schematics of a CPU. While the conventional CPU has a low count of cores designed to excel at executing sequential operations on a wide range of tasks, GPUs present more simple computing units able to perform the same tasks on parallel, achieving massive throughput to the cost of flexibility.

Over time and with the releases of specialized toolkits such as CUDA [21], GPUs became general purpose processors which increase their use on scientific tasks. Due to the large market share GPUs currently have, it does not suppose any problem to get access to them at a relative low cost. Therefore, GPUs are a very convenient and powerful accelerator for applications where massive parallel tasks occur.

During the last decades, new algorithms have been implemented to run on GPUs efficiently [22, 23]. Thus, nowadays GPUs are being use extensively on physics simulations [24, 25]. Some reasons of it success are, other than its massive speedups over conventional hardware, its power efficiency in embarrassingly parallel applications and its relatively ease to use in comparison to other accelerators.

However, for all the simultaneous Arithmetic Logic Units (ALUs) computations, a GPU still has to access registers or shared memory to read and write intermediate results. This overhead can be reduced with existing techniques such as memory coalescing. However, the reality is that most programs will not go that far as efficient GPU programming requires high skills.

2.1.2 Field-Programmable Gate Arrays

A Field-Programmable Gate Array (FPGA) device can be programmed to remove the overhead usually found on general purpose architectures. FPGAs allow the programmer to,

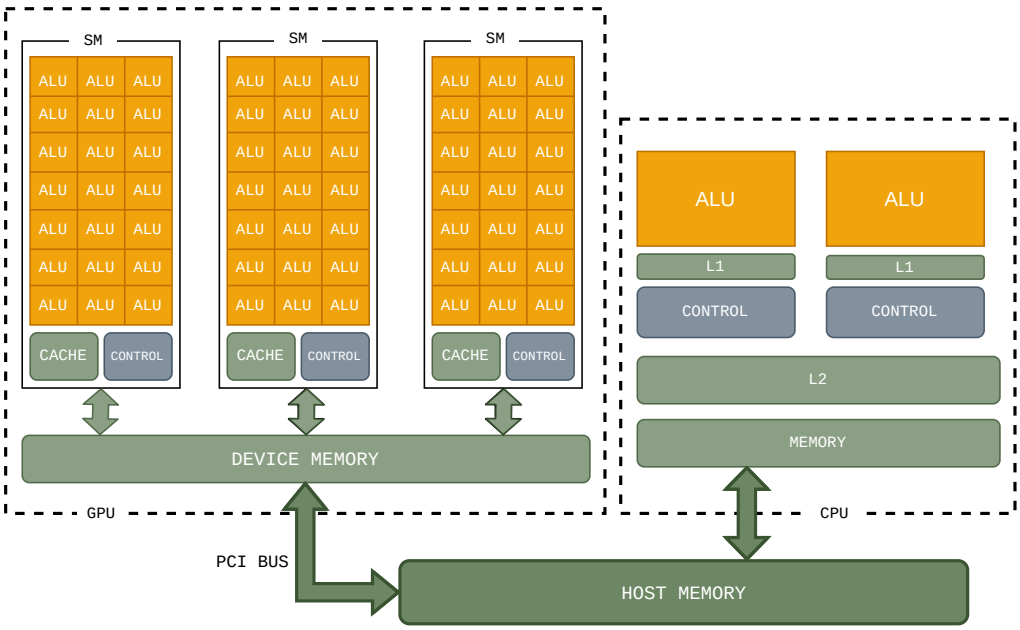


Figure 2.1.: Simplified block diagram of the internals of GPUs and CPUs.

from a very low level approach, control the physical connections on the integrated circuit itself through the use of dynamic gates between transistors.

FPGAs present certain advantage to regular processors, such that, a programmer with a high degree of knowledge, can design chips able to perform specific tasks really efficiently. In addition, because the circuit is mainly dynamically programmed instead of built during manufacturing, the use of the same chip can be completely changed on a moment without any hardware modifications. This versatility is useful on scenarios where a few different high performance workflows can be expected. Hence, it provides an option to run simulations with very low latency as well as low power consumption in fairly stables executions. For those reasons, FPGAs are currently being use to run several HPC physics simulations, such as computing quantum simulations [26] or molecular dynamics [27] among others.

Nevertheless, the main disadvantage in order to create an efficient FPGA to run your simulation is the need of extensive knowledge that most physicists do not have. Moreover, the unreasonably long compiling times and less than ideal working tools, make FPGAs not suitable for being the de facto accelerator for physics calculations. Instead, GPUs offer a lot more convince while being fairly less expensive.

2.1.3 Application-Specific Integrated Circuit

When a tasks becomes too frequent that even the versatility of an FPGAs becomes unnecessary and you want to get rid of their performance limitations, you can start looking

for Application-Specific Integrated Circuit (ASIC). Despite its high initial cost and poor programmability, ASIC accelerators provide the highest efficiency. Usually, engineers are able to remove most of the overhead as well as to optimize memory access patterns for the given task. This explains why, recently, large corporations are spending their sizable budgets on developing ASIC solutions to cater all the AI computations currently running on their data centers [14].

Therefore, we will focus our attention on ASIC hardware. Specifically, on those that make use of Matrix Engines (ME) as their core architecture. Some examples of Matrix Engines based accelerators are shown in Table 2.1. It is relevant to notice that most of the previous mentioned accelerators are marketed towards AI workloads design for high performance but also high energy efficiency. And even though they are intended to perform GEMM, most are flexible enough to be considered general purpose units, with an accelerator attached.

Accelerator	ME Size (PE)	TFlops/s	TDP (W)	ME Engine
IBM Power10	4x4	16.3	190	Matrix Accelerator
Google TPU v1(Chip)	256x256	92(int8)	40	Systolic Array
Google TPU v2(Chip)	128x128	46	280	Systolic Array
Google TPU v3(Chip)	128x128	123	450	Systolic Array
Huawei Ascend 910	16x16x16	320	310	Cube Engine
NVIDIA Tesla V100	4x4x4	112	300	Tensor Core
NVIDIA Tesla A100	4x4x4	312	250	Tensor Core

Table 2.1.: Overview of some of the most popular accelerators using Matrix Engines [28, 29].

2.2 Google Cloud TPUs

One of the first and more popular solutions for Deep Learning accelerator has been Google Cloud Tensor Processing Units (TPUs). Their performance-efficiency balance as well as their continuous record breaking reports on ML contests have given them a position of advantage over their rivals. Being the product of one of the largest companies AI companies in the world has also helped on their consolidation. Because of that, we have decided to based our study on their chip, since we expect it to remain on the edge of ME accelerators as well as to continue with a similar architecture for the years to come.

In this section, we will briefly introduce their main qualities together with what we consider some of their limitations. For that, we will use the third generation TPU currently available on Google Cloud. Despite some news regarding a fourth generation chip, we did not have access to them during this thesis and all the discussions are based on v3. We expect that most of the points will remain valid for several generations since the main focus seems to be on increasing performance and not structural changes.

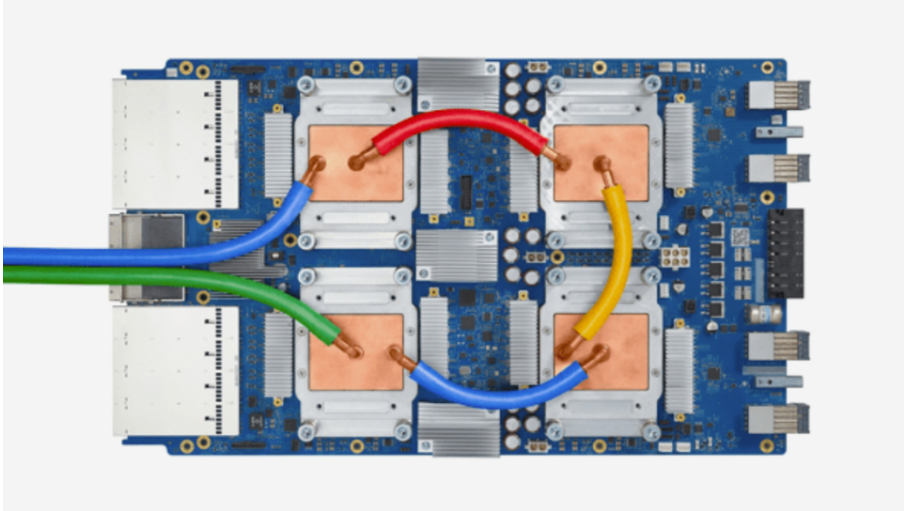


Figure 2.2.: Picture of a third generation Tensor Processing Unit.

Device Architecture

As seen in Figure 2.2, a TPU Unit consists of 4 TPU chips. On each of those chips, there are 2 TPU cores. Currently, there is no way to differentiate between cores inside a TPU unit and it can be considered that the communication between cores inside the same chip is as fast as the communication between cores of different chips from the same unit.

In contrast to GPUs, none of the cores share memory with each other. In a third generation TPU, a core has 16GB of High Bandwidth Memory (HBM). That is similar to what an entire NVIDIA V100 GPU has access to [1]. Hence, using the 8 cores of a single TPU Unit would grant you 128GB of HBM while using 32 128-bit buses to four short stacks of DRAM on each chip (2 cores).

Inside each TPUv3 core, there are two matrix multiplication units (MXU). The MXU consists of a 2D systolic array of 128×128 identical Processing Elements (PEs) that support floating point fused multiply-accumulate (FMAC) arithmetic. Thus, the MXU is capable of performing 16384 ALUs operations every clock cycle. Considering that its reported clock cycle is $940MHz$, the theoretical limit for a TPU chip can be computed as

$$940 \cdot 10^6 \frac{\text{cycles}}{s} \cdot 2 \frac{\text{cores}}{\text{chip}} \cdot 16384 \frac{\text{FMAC}}{\text{cycleMXU}} \cdot 2 \frac{\text{FLOPS}}{\text{FMA}} = 123\text{TFLOPS}/s$$

Additionally, each core has core sequencer fetcher which interprets the instructions from the software-managed Instructor Memory and executes scalar operations using a 4K scalar data memory and a 32 scalar registers of 32 bits. Moreover, since the interpreted instructions used are 322-bits long (VLIW), the sequential unit is able to perform 8 operation per

[1] NVIDIA also offers a 32GB of HBM configuration

fetching, such as 2 scalar, 2 vector ALU, vector load and store and queue data from the MXUs.

As can be seen in Figure 2.3, the TPU core also has a vector processing unit (VPU) to perform SIMD operations while also containing the vector memory of the chip, which can hold up to 32K 128×8 32-bit elements (16MiB), in addition the the vector registers than can store up to 4KiB with 128×8 dimensions.

The Transpose Permute Unit does 128×128 matrix transposes, reductions and permutations of the VPU lanes.

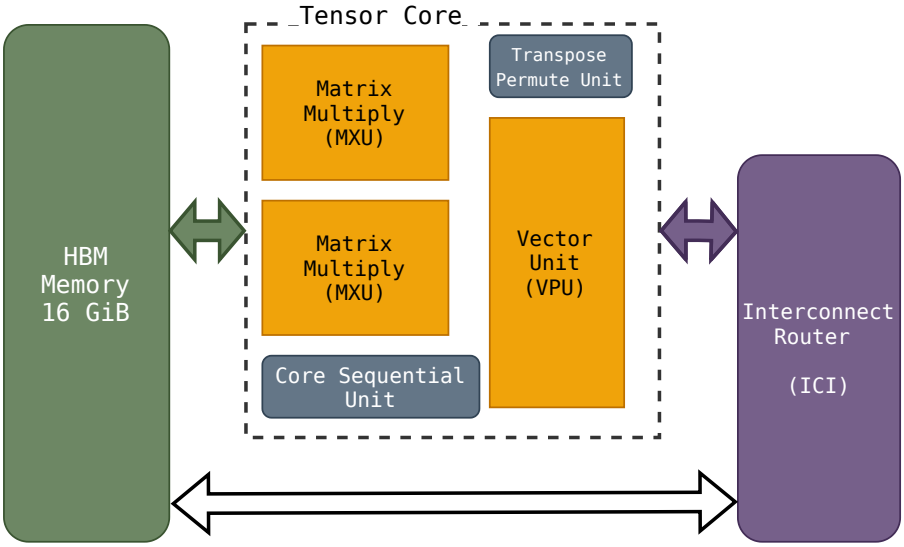


Figure 2.3.: Simplified scheme of the internal architecture of a Tensor core of a third generation TPU

Systolic Array

Since the MXU is the main computing power of a core, and it is optimized for MAC operations, the program should make use of it extensively in order to get the maximum performance out of the unit.

It is interesting to understand why exactly having a systolic array inside the accelerator allow it to perform dense matrix-matrix calculations efficiently.

As the systolic part of its names suggests, data flows through the chip at each clock cycle, similarly to blood flowing through the heart at each heartbeat. This means that the output from the multiplication of one PE goes as input to the following PE, where it is added to the other multiplication.

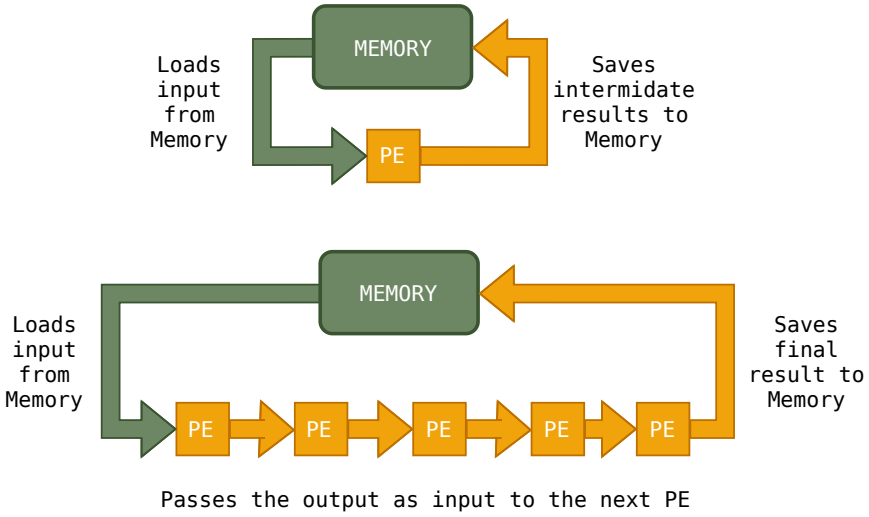


Figure 2.4.: Illustration of the idea behind systolic arrays in a simple scheme.

Therefore, it avoids the need of writing and reading intermediate results on the register or the shared memory. The MXU only needs to load the matrices once to feed them into the systolic array and save the final output. This flow of data can be seen on Figure 2.5.

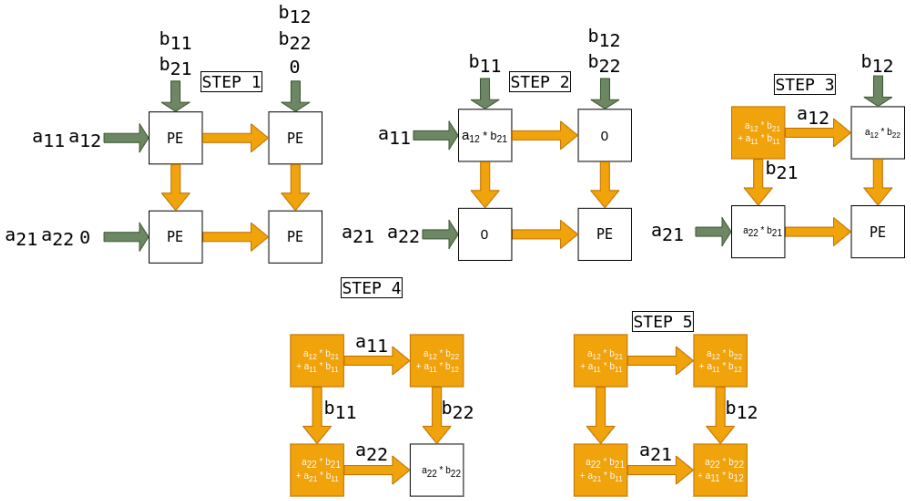


Figure 2.5.: Google own MXU scheme. Do a quick graph where it kind of follows the path of the value.

In contrast, GPUs without Tensor Cores will be able to perform operations simultaneously, which is a great speedup with respect to conventional CPUs, but will still be bottlenecked by their memory accesses.

Brain floating point precision

Computer systems have to save real numbers in bit representations. The precision of these representations are determined by the number of bits and the location of the decimal point. We will use the IEEE floating points definitions to talk about data types.

Since machine learning research has proved that using lower precision arithmetic does not degrade their prediction accuracy significantly, they can get away using lower precision values to reduce their model's memory requirements as well as to speedup their calculations [30]. The most precise numeric representation that a third generation TPU can operate with is single floating points (`float32`). Because physics simulations try to recreate real continuum phenomena, it is common to use double floating points (`float64`). Hence, it is still not clear if the precision requirements can make MXU not suitable for certain physics workloads. The reduction is more drastic however when we make use of the MXU inside the TPU. The MXU only works with brain-floating points (`bfloat16`). Namely, it will convert the input from `float32` to `bfloat16` to perform the multiplication and it will return an output of `float32` for the addition. At the moment of writing this thesis, a similar approach is used on NVIDIA Tensor Cores[31].

Some studies have shown that the use of mix precision for solving GEMM can be suitable outside of DL computations [32, 33]. Additionally, there has been a tendency to decrease precision on certain simulations in order to reuse the computational power to increase their range [34]. Nevertheless, in physic simulations, low precision can lead to increasing divergences on the results and should be studied on each application individually.

Another interesting point on the lower precision of TPUs is the use of brain floating points instead of the standard IEEE half-floating point (`float16`). Brain floating points (`bfloat16`) is a custom floating point format design to work on DL applications, where the exponent has the same range as `float32` while reducing the number of bits of the mantissa part. That means that the resolution of the decimals is further reduced to 7 bits instead of the 10 found on IEEE half-precision `float16`, thus, the precision is between two and three decimal digits. A comparison can be seen on Figure 2.6.

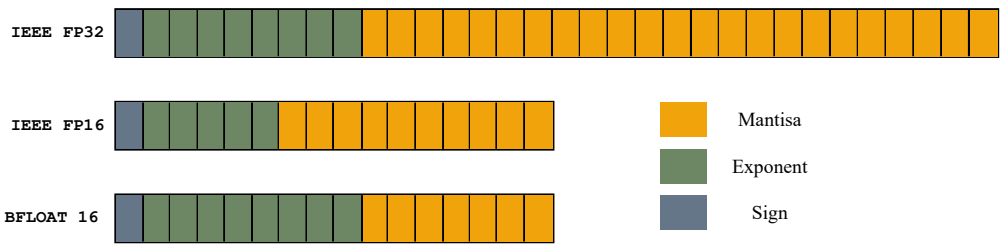


Figure 2.6.: Comparison between standard IEEE formats and Brain floating points.

Power efficiency

One of the original reasons for TPUs conceptualization was the need to find a more efficient accelerator to run machine learning algorithms on the data centers at Google. As a consequence, the first TPU would only consume, based on its TDP value, $75W$ per chip, which is considerably lower than any GPU. However, on the second generation, they were able to increase performance in such capacity, that even though the TDP increased to

$280W$, the throughput achieved made it still worth it. Similarly, when the third generation of TPUs was release, with almost three times the performance of the previous generations as seen in Table 2.1, which required an increase on its TDP to $450W$. Hence, we assume that the power consumption of an enteire TPU would be around $\sim 1800W$.

Looking at the original paper Jouppi *et al.* [14], it is stated that even though the TDP on a v1 TPU is $75W$, the energy consumption measures on busy workloads was $40W$. Even though this is expected, we will be assuming TDP value as actual consumption during this thesis as there is no way to measure actual energy consumption. As a consequence, it will make an unfair comparison as other devices or even newer generation of TPUs may not have such a large gap between TDP and actual consumption.

Since we know the TDP of a current generation TPU as well as its theoretical maximum performance, we expect that a TPU will be able to produce 27.33 GFLOPS$/W$.

TPU Pods

Lastly, one of the main advantages of TPUs is their high connectivity between devices, as in theory, it makes TPU highly scalable solutions. We use the term Pod to describe a collection of TPUs devices connected over network . For a third generation TPU, a Pod provides a maximum configuration of 256 devices for a total 2048 cores and 32 TiB of HBM memory, with its chips on a 16×32 2D torus form due to each chip containing 4 connections of 656 Gbits/s each as seen on Figure 2.7

The reason for having high speed connections between different devices is due to having physical connections between the devices themselves, without the need of going through the host. The interconnection of the devices are designed in 4 connections per chip, hence, it is to expect that the speed between further devices is worse than between near ones.

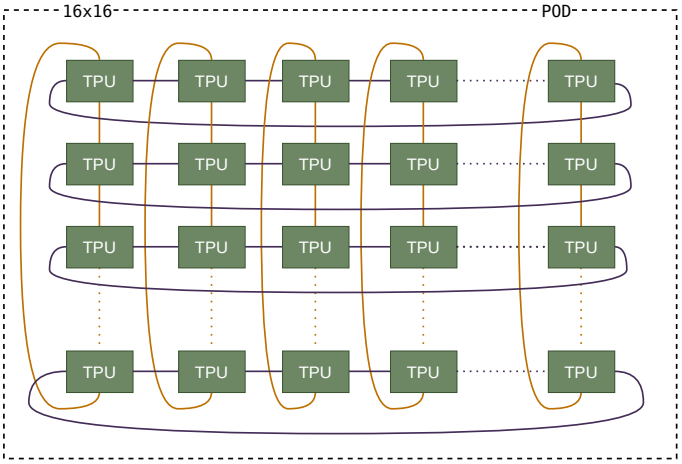


Figure 2.7.: TPU v3-512 slice with a 2D Torus Topology.

3 JAX and Benchmarking

With the purpose of motivating the transition of high efficiency languages into running large scale physical simulations, a high performance programming framework is needed. At the moment, there are several frontends to XLA, the compiler that allows code to be run on TPUs, such as TensorFlow, JAX, PyTorch, Nx and Julia. In this section, we justify the decision of using JAX as our chosen frontend. Additionally, the advantages and disadvantages of JAX are discussed along with the past and current state of TPUs in relation to working with JAX, describing the limitations experienced during this project.

This section it is not expected to be a tutorial on JAX, but to serve as an introduction to the basic features that will allow to understand the discussions and implementations that will appear during the remaining parts of the dissertation.

3.1 The XLA Compiler

XLA is a high performance linear algebra compiler targeted towards accelerating the operations found on machine learning workloads [16]. XLA is designed to improve calculation speed by removing inefficiencies in addition to reducing the memory usage of the program by searching and merging temporal allocations of memory.

Originally designed to work with TensorFlow graphs, XLA implements a set of operations that are mapped to common TensorFlow functions so that, if specified, the operations can be compiled into a High Level Operations (HLO IR) language that acts similar to a compiler intermediate representation (IR), which then can be given as an input to the XLA compiler.

Once XLA has been given the HLO graphs defined by the XLA operations, it compiles them into machine instructions that can adapt to different types of hardware architectures. For the sake of simplicity, we will only assume that those architectures are CPU, GPU and TPU. On the CPU case, XLA uses LLVM for the machine instructions, the GPU backend uses LLVM NVPTX on NVIDIA GPUs and in the case of TPUs, a proprietary backend with no public information available. Therefore, despite XLA allowing the use of custom calls

on CPUs and GPUs by making use of frameworks such as CUDA, no custom calls can be implemented on TPUs.

In order to provide the mentioned speedups, XLA performs a set of target-independent as well as target-dependent optimizations before the code is even run. For instance, XLA performs Common Subexpression Elimination (CSE) before sending the HLO code into the device, meaning that the compiler searches for identical expressions that yield similar results and reuses them as needed. Taking the snippet below as a showcase:

```
c = a/x + b
d = a/x - e
```

we see that the division `a/x` is performed twice on a short period of time. This would be computed twice on a normal Python/NumPy program, however, since XLA is aware of these inefficiencies, the previous snippet is translated into:

```
t = a/x
c = t + b
d = t - e
```

which reduces the computing time by dividing `a` over `x` only once.

However, the major boost in performance that XLA offers comes from fusing composable XLA operations. Lets take the case that, in some part of the program, we are performing an element wise addition of two vectors. The low level approach code would look something like this:

```
  for i in range(N):
    c[i] = a[i] + b[i]
```

But in the case that later on the program, the same vector c needs to be multiplied by another vector, we would also have something like

```
  for i in range(N):
    d[i] = c[i] * a[i]
```

When looked in terms of element wise operations, it is clear that having to loop through the array N twice is a waste of time, but this is not that obvious when a program uses vector operations such as

```
  c = a + b
  ...
  d = c * a
```

The power of XLA comes when it is able to identify these vector operations, since they are written in terms of XLA operations, and fuse them together so that the low level code performs something like this.

```
  for i in range(N):
    d[i] = (a[i] + b[i]) * a[i]
```

as the XLA graph allows it to see if any of this functions will be used prior to running them. In contrast to Python/NumPy programs, XLA is able to see what will be run and therefore, can cache recurring functions so that there is no need to interpret them again.

The second most important feature of XLA is its ability to manage memory, as the resulting compilation would, in theory, avoid redundant memory allocations by knowing whether a variable will be used further on the program.

With these features, XLA is capable of overcoming two of the major drawbacks that high level dynamically-typed languages currently face. Hence, the reason it is such an appealing solution for HPC programs.

XLA was originally designed to work with TensorFlow on Machine Learning tasks, but in the very recent years, some support for other frontends to XLA have been developed with the most prominent candidate of those being the JAX Python numerical library.

3.2 JAX: An accelerated NumPy Library

JAX is a high performance numerical computing Python library targeting machine learning research that serves as a NumPy-like frontend for the XLA compiler[15]. JAX is being used to develop high performance physics frameworks such as in the molecular dynamics field [35] and an ocean simulations [13] as it has proven to perform competitive performances while still making use of the flexibility Python provides. JAX main advantage over libraries such as NumPy comes from the use of a set of transformations functions that allow them to use the XLA compiler above common array-manipulating programs. These transformations are *Just In Time* compilation, automatic vectorization, automatic differentiation, and single-program multiple-data (SPMD) parallelism. From those, the ones we will be focusing are the JIT compilation and the parallel execution, as the other are more targeted towards machine learning programs.

JAX provides a simple and powerful API for writing accelerated numerical code, but working effectively in JAX sometimes requires some additional considerations.

3.2.1 Just-In-Time Compilation

The JIT compiler transformation is based on the idea of collecting the information from the operations inside the function as well as some information about the function inputs, so that it can be converted to proper input for the XLA compiler to perform its optimizations.

The process resembles to a TensorFlow Graph transformed to HLO that makes use of XLA predefined operations.

In order to convert the abstract NumPy-like code into XLA input, JAX uses JAX primitives. JAX has already implemented a set of JAX primitives that map to most of the XLA operations. By transforming the user code to those primitives, new JAX primitives need be defined. However, JAX allows the creation of primitives by using existing ones. In reality, most of the NumPy functions that JAX comes with are implemented by using other JAX primitives.

Once the function to run is implemented in terms of other JAX primitives, it becomes a traceable function that can make use of all the transformations that JAX has to offer. For instance, `jit` would take a traceable function and return a semantically identical function but lazily compiled by XLA for accelerators. In order to be used on different input values, JAX traceable functions define the input as abstract objects where information such as the data type or the array shape is passed to the compiler but not concrete values. This abstract information is passed the first time the function is called, hence it is to expect that the first run will take longer to run as it needs to be compiled. Adding the `jit` decorator to the function should be enough to compile it.

```
@jax.jit
def pitagoras(a, b, c):
  return jax.numpy.sqrt(a**2 + b**2 + c**2)
```

The compilation time increases for large complex functions where common Python control flows are used. However, using JAX specific approaches to substitute them usually solves the issue. For instance, if we were to create a function that performs a `for` loop, XLA would statically unroll it in order to optimize the function execution speed. Nevertheless, JAX provides the `jax.lax.fori_loop` primitive that avoids unrolling the loop but still returns traceable functions.

Similarly, defining conditions inside jitted functions yields unexpected errors as the compiled can not predict the exact type of the argument used in the condition, and thus compilation is usually generic enough to adapt to different arguments. By doing so, avoiding unnecessary recompilations that may results on very low performances. Hence, JAX supports the use of `jit` arguments such as `static_argnums` where the user can specify whether the argument type is going to change or not on different calls. If that is not the case but you still need a conditional evaluation inside the compiled function, the use of `jax.numpy.where` and `jax.lax.cond` is recommended.

3.2.2 Single-Program Multiple-Data

JAX allows for the same operation to be sent to different devices in order to be run on parallel. Commonly, this is used to run different batches while training Neural Network models, as their independence between each other makes them the perfect candidate for parallel runs. However, the API provided is relaxed enough that can be used to perform other type of calculation across XLA devices.

As most of the transformations proposed by JAX, the usage of `pmap` is fairly straight forward. Assuming you have 8 devices, as it would be the case for a TPU unit with 8 cores, you can run an operation on each device by creating by sending a sharded array as an input

```
>> x = jax.numpy.arange(8)
>> f = jax.pmap(lambda i: i+i)
>> f(x)
ShardedDeviceArray([ 0,  2,  4,  6,  8, 10, 12, 14], dtype=int32)
```

The above computation has been performed on 8 devices simultaneously, where each has received a single scalar from the original device and has performed the sum over just one value. Of course, this approach is readily extensible to larger dimensions. As the name of the resulting array indicates, after being sent to each device, the array is not collected on the initial core as one may expect. Instead, the host collects the requested values only to display from the XLA devices, but the arrays themselves remain on the distributed devices.

Additionally, JAX provides access to collective operations that allow for communication between devices without relying on passing through the host. Collective operations rely on the use of named axes to decide which information is shared between devices. The idea of named axes has been used as the base for the very experimental `xmap` transformation which will not be covered in any of the work done on this thesis but that early testings indicates that it will provide a powerful API to distributed code on JAX.

3.2.3 JAX on TPU

JAX gained support for running on TPUs on December of 2019. However, no significant information has been published with regards to it since then. That is due to the complexity of running JAX on TPUs with respect to using TensorFlow instead. In this section, the key instructions to make use of TPUs while using JAX are outlined, as well as an introduction of the new approach JAX has been developing on the recent months to improve their end user experience. Some performance guidelines are given specifically for TPUs.

Cloud TPUs provide high performance with relative ease to use. If the code works on other hardware, most likely works on a TPU as well. However, in order to avoid inefficiencies specific of TPUs some prior knowledge is necessary.

Arrays on TPUs are tiled, as described on Chapter 2, the vector registers are designed to store 128×8 32 bits elements. Hence, when using arrays that are not multiple of those dimensions, XLA will pad them with 0. Even though XLA does it so that the data can be manipulated efficiently, the reality is that we are adding null information that does not contribute in any useful manner to our program. Therefore, the best performance will always be achieve when no padding during the compilation is involved.

Regarding the lower precision data type discussed on Chapter 2 with regards to the MXU, XLA provides methods to avoid them, whilst to the cost of performance. With the purpose of achieving higher precision, JAX can simulate multiplication of larger floating-point resolutions by increasing the number of passes on the MXU. For instance, in order to achieve roughly 16 bits of precision on the significant, JAX can force the TPU to perform 3 consecutive passes on the MXU for each multiplication. Likewise, 32 bit precision (`float32`) can be achieved if the TPU performs $6\times$ the amount of passes of a single matrix multiplication. Clearly, having to repeat operations affects performance significantly and, despite being the option to compute more precise matrix multiplications for specific parts that may needed it, a lower precision requirement should be necessary if you want to run the code on TPU with decent performance.

Despite being provided on Google Colaboratory, in order to use TPU effectively, ones needs to use Google Engine Cloud. During half the time of this thesis, the approach to running code on TPUs was as follows:

1. From the host machine, the user needs to start a Google Cloud virtual machine VM.

2. Once started, one connects to the remote VM to start the defined TPU associated with the Google Account. Because TPUs are still used by TensorFlow and JAX has not had an stable release yet, a specific version should be stated when creating the TPU.

3. Once both the VM and the TPU is running (and adding cost to the bill), the IP of the TPU shown in the Google Cloud Console should be indicated at the start of the JAX program so that it can connect to it. That connection can only happen from the VM to the TPU inside Google Cloud and can not be used outside in places such as Colaboratory.

4. Then the code is compiled on the host machine and later send to the TPU hosts, where they send it to the TPU devices themselves and later return to the user VM.

Figure 3.1 shows a diagram of how code run through TPUs worked. Even if the VM was to be really close to the TPU hosts machines, all the code had to be sent back and forth over the network, making performance subpar compared to the raw power of TPUs. Luckily, the JAX on TPU researchers were aware of the limitation of this appeal and at the end of last year, announced a completely new workflow to deal with Cloud TPUs.

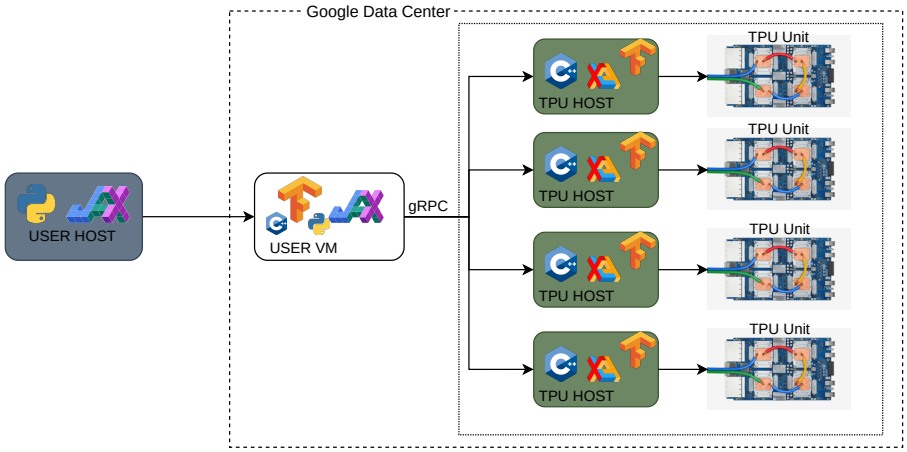


Figure 3.1.: Diagram of the current JAX-TPU Cloud Infrastructure.

Although still in early stage, the researches at Google Brain gave us access to the new proposed mechanism. In contrast to the old one, a direct connection to each TPU host now is possible, reducing all the latency previously experienced. Moreover, there are no workarounds that use TensorFlow backend in order to use the Cloud TPUs. In removing the hassle of using Google Cloud VMs and the drawbacks of the previous connections, JAX researches have been able to provide a very easy and clean experience that makes using TPUs no harder than connecting to a typical remote desktop. Even though the porting is not finished at still yields unexpected errors from time to time, I strongly believe that the new access to Cloud TPUs is an improvement and removes most of the concerts that dealing with TPUs on JAX for daily operations may have.

The new system also provides an interesting solution to dealing with multiple TPU units, as it requires and independent `ssh` connection to each in order to be used as seen in Figure 3.2. JAX considers each TPU to be run independent processes in parallel, which is useful when dealing with I/O tasks or data management. However, it requires certain tools that allow for broadcasting commands over multiple remotes to be run on the same time.

In case of `pmap`, each process (or host in the default configuration), should still call the distributed function with the sharded array only taking into account the local devices. Local devices are all of those XLA devices connected to the same host (or process), such

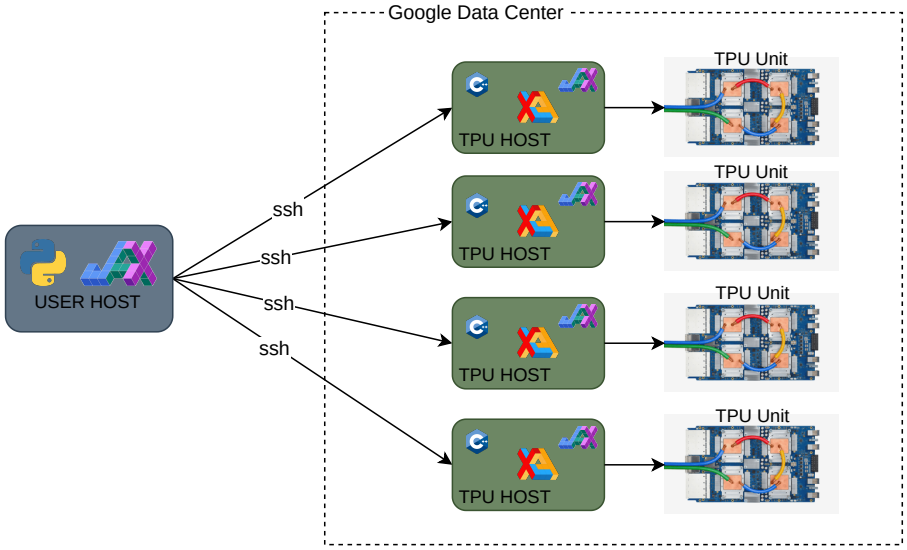


Figure 3.2.: Diagram of the new alpha JAX-TPU Cloud Infrastructure.

that a single TPU host would have 8 XLA local devices, despite being part of a larger TPU Pod. This independence between hosts allows them to run different Python/JAX code, which end up in a performance boost. However, they are not completely separated from one another as XLA collective operations such as `gather`, `sum` and `permute` are still computed over all the running devices. Hence, if a single program is to be run, the only hassle involved is that the python script, designed to work on a single TPU, has to be run simultaneously on each host.

3.3 Performance Analysis

Profiling is used to find bottlenecks that allow us to get the biggest performance boost with the least amount of work by understanding the parts of the program that take the most time to run. Therefore, if we were to find efficient approaches to speedup physics calculations, a working profiler would be an essential tool.

Despite the mention of profiling in JAX's official documentation and the instructions to use TensorBoard, a TensorFlow visualization toolkit, on Google Cloud Docs [36], in addition to plenty of opened issues on the GitHub repository and countless attempts to make it work, it was not possible to get a usable profiling on the current system. Without a profiling tool, we were stuck to *hand* measurement the performance of single functions by making use of microbenchmarking techniques such as the `timeit` and `time` libraries. An important note on microbenchmarking JAX functions is that they need to implement a blocker, since JAX uses asynchronous dispatch for the functions and therefore, using `%timeit` may yield the dispatch time instead of the calculation time. For instance, if we were to measure the performance of a dot product:

```
>> %timeit jax.numpy.dot(x, x)
100 loops, best of 3: 37.8 us per loop
```

However, by indicating jax to wait until the computation has been completed to run again, we obtain a different results

```
>> %timeit jax.numpy.dot(x, x).block_until_ready()
100 loops, best of 3: 16.23 ms per loop
```

Because once a function gets compiled, the operations are translated to machine instructions, and despite XLA allowing for a detailed description on those instructions on other architectures, those were not available to see when working on TPUs. Thus, there was no possible manner of seeing what optimizations were actually happening other than by performing small changes to the code and evaluating their repercussion on different conditions to understand their relevance on the approach. Therefore, something that may haven taken minutes using a profiling tool, became a task which duration could vary from hours to days.

In the memory management analysis, the only way to know whether XLA was optimizing properly certain operations was to predict the memory usage in a NumPy workload and check if the system would crash when going above the theoretical limit.

For a brief moment at the end of 2020, the profiler was able to store tracings of the XLA operations that run on TPUs. However, when visualizing them on the Profiler, the only information displayed was `regions` of the code which were the result of fused operations, as seen on Figure 3.3. In conclusion, there was no possible mapping to be done between the tracings and the code as a single `region` could come from several fused operations. Furthermore, the profiling stopped working briefly after, due to incompatibilities of the TPU version with the current TensorFlow and TensorBoard versions, as changes in the saving mechanics were being implemented.

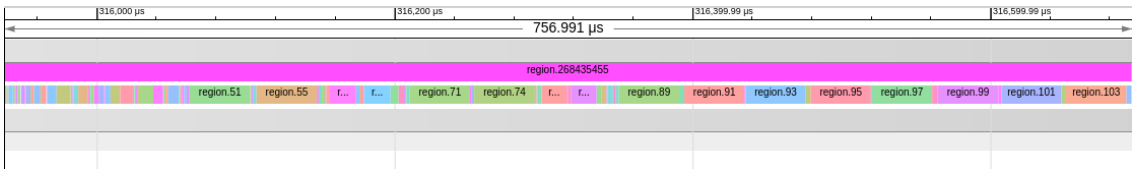


Figure 3.3.: First snapshot of a simulation being caught on the profiler.

When the new alpha system was introduced, our hopes on having a working profiler raised again, and with almost half the thesis time remaining, we could still manage to get a steady boost on the productiveness of evaluating code. However, that was not the case and, even though they stated that it was being worked on, the profiler on the new system was completely missing up until a few weeks before the submission of this dissertation.

Even though the brief amount of remaining time does not allow us to make use of the profiling tools, it serves to give us a glimpse at how the profiler on TPUs works and how useful could have been in performing the evaluations. An example on what the tracings look like now can be seen on Figure 3.4, where a lot for information on what operations are actually being computed is displayed. Additionally, the new profiler includes a memory analyzer as well.

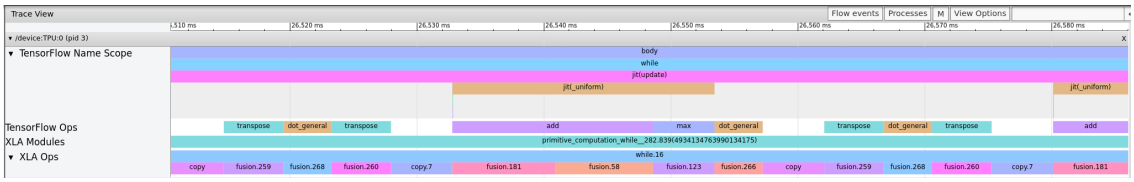


Figure 3.4.: Snapshot of a simulation being caught on the newly release profiler for alpha TPUs.

4 Partial Differential Equations

4.1 Introduction

Physic problems are commonly based on the description of physical phenomena in order to understand and predict its behavior. Usually, the resulting description can be expressed as changes on their variables, and the way to formulate those changes is with the use of differential equations. When the physical law or equation depicts changes with regards to variables on systems that depend on two or more independent variables, partial derivatives are used. Therefore, these studies will lead to the formulation of partial differential equations (PDEs) in order to describe them.

A wide range of PDEs is used in every aspect of physics and thus, one expects to frequently encounter them when doing calculations on fields such as fluid dynamics, quantum mechanics or electrodynamics among others. One of the most common cases on physics, and therefore one which will have a substantial impact on our evaluation are the two dimensional second order equations, whose general expression is as follows:

$$a\frac{\partial^2 U(x,y)}{\partial x^2} + b\frac{\partial^2 U(x,y)}{\partial x \partial y} + c\frac{\partial^2 U(x,y)}{\partial y^2} + d\frac{\partial U(x,y)}{\partial x} + e\frac{\partial U(x,y)}{\partial y} + fU(x,y) + g = 0 \quad (4.1)$$

Considering that we are studying whether TPUs are a straightforward drop in replacement for existing simulations, the finite difference method (FDM), one of the most common and versatile numerical method to solve PDEs, will be analyzed. The FDM is based on the assumption that derivatives respect one or more of the variables in the system are approximated by the difference between consecutive elements, also called difference coefficient. Hence, as proved on Appendix A, we can express first order partial derivatives as

$$\frac{\partial U}{\partial x} \simeq \frac{U(x+\Delta x, y) - U(x-\Delta x, y)}{\Delta x} \quad (4.2)$$

and second order partial derivatives

$$\frac{\partial^2 U}{\partial x^2} \simeq \frac{U(x+\Delta x, y) - 2U(x,y) + U(x-\Delta x, y)}{\Delta x^2} \tag{4.3}$$

where Δx indicates the distance between two consecutive values of x. In order to simplify notation, from now on we will use h to describe spatial distances and we will consider all the spatial variables to be equally separated. Thus, (4.2) and (4.3) will look like

$$\frac{\partial U}{\partial x} \simeq \frac{U(x+h, y) - U(x-h, y)}{h} \tag{4.4}$$

$$\frac{\partial^2 U}{\partial x^2} \simeq \frac{U(x+h, y) - 2U(x,y) + U(x-h, y)}{h^2} \tag{4.5}$$

and analogously for the y-axis

$$\frac{\partial U}{\partial y} \simeq \frac{U(x, y+h) - U(x, y-h)}{h} \tag{4.6}$$

$$\frac{\partial^2 U}{\partial y^2} \simeq \frac{U(x, y+h) - 2U(x,y) + U(x, y-h)}{h^2} \tag{4.7}$$

Research and studies regarding the accuracy of the finite difference method have been extensively done at this point. Therefore, we will not focus on the error between numerical simulations and exact solutions, but instead, study the accuracy difference between TPUs and GPUs.

Moreover, it is common to work on grid points when solving PDEs. The following expression are the discrete grid points representation of (4.4) and (4.5) respectively.

$$hD_x u_{i,j} = u_{i+h,j} - u_{i-h,j} \tag{4.8}$$

$$h^2 D_{xx} u_{i,j} = u_{i+h,j} - 2u_{i,j} + u_{i-h,j} \tag{4.9}$$

where D_x and D_{xx} indicate the first and second order derivative while i, j indicate the coordinates on the grid.

In addition to being one of the most common problems regarding physical simulations, FDMs are not considered to be efficient when performed using matrix multiplications. Meaning that, if we were to achieve similar performance on TPU than on other non-matrix based accelerators, it would indicate that TPUs will not suffer a penalization on methods that do not use matrix multiplication explicitly.

4.2 Benchmarks

In order to obtain general results that can be applied to a wide range of simulations, we have used three distinct, well known numerical simulations that can be easily reproduced if needed. Before explaining the different implementations and the results obtained, we will briefly introduce each benchmark and their main characteristics. All the simulations have been implemented using pure Python/JAX and even though not all of them will be used for every study case, they will serve to provide a general sense of how each method performs on simulations of varying complexities.

For the sake of simplicity, we will assume Dirichlet boundary conditions on all the benchmarks whenever possible. However, if in order to use another boundary condition on any of the proposed method it ought to be explained, we will clarify it on the corresponding method section.

We will run the benchmarks on conventional hardware (CPU), the prominent current accelerator (GPU) and in different configurations of the TPU; a TPU core, a TPU Chip (2 Cores) and a TPU Unit (4 Chips). The technical information about the used devices can be found on Appendix B. The benchmarks are also implemented differently when only one XLA device is in used, so that the exchange and message passing part is not considered. Those cases are on CPU, a single GPU and a TPU Core. On the other hand, the benchmarks run on a TPU Chip and a TPU Unit have been specially written to be very scalable, meaning that excluding the boundaries exchanges and the collectives operations, each core will run the same operations as in the single case. We will talk about their connectivity on each implementation.

4.2.1 Heat Diffusion

The first and most simple benchmark we will be implementing is the Heat Diffusion benchmark. It simulates the heat transfer on a surface represented on a 2D grid. We will consider the Heat Equation in a steady state, thus, invariant on time. We will also assume no internal generation of heat either, so that the PDE to solve is as simple as possible and can be expressed as the Laplace equation.

$$\Delta T(x,y) = \nabla^2 T(x,y) = \frac{\partial^2 T(x,y)}{\partial x^2} + \frac{\partial^2 T(x,y)}{\partial y^2} = 0 \tag{4.10}$$

with $T(x,y)$ being the Temperature distribution on the 2D surface. Since the steady heat diffusion problem can not be solved with an explicit scheme, we will use an iterate

solver. For the sake of simplicity and ease on parallelization, the Jacobi solver is used until numerical convergence is achieved, despite the convergence rate fo the Jacobi method being embarrassingly slow and the existence of faster convergence methods such as the Successive Over-Relaxation (SOR) iterative solver or the Gauss-Seidel method. Nevertheless, these more efficient methods do not translate as well as the Jacobi solver does to vector operations.

The Jacobi method does not always converge, but it is guaranteed to converged under conditions that are satisfied for this simulation. On its discrete form, (4.10) can be expressed as

$$\frac{t_{i+1,j} - 2t_{i,j} + t_{i-1,j}}{\Delta x^2} + \frac{t_{i,j+1} - 2t_{i,j} + t_{i,j-1}}{\Delta y^2} = 0 \tag{4.11}$$

where i, j are the coordinates on the grid. Hence, on its implicit form and given the Jacobi formula, the equation we will iterate will be

$$t_{i,j}^{n+1} = \frac{1}{k}\left(\frac{t_{i+1,j}^{n} + t_{i-1,j}^{n}}{\Delta x^2} + \frac{t_{i,j+1}^{n} + t_{i,j-1}^{n}}{\Delta y^2}\right) \tag{4.12}$$

where k is given by $k = 2\frac{\Delta x^2 + \Delta y^2}{\Delta x^2 * \Delta y^2}$. Applying the assumptions stated before, we will use h to indicate the separation between grid points on both axis, meaning $\Delta x = \Delta y = h$. Thus, $k = 4/h^2$. This change simplifies the calculation since the h^2 cancel each other on (4.12) and we end up with the following stencil operation also visualized in Figure 4.1.

$$t_{i,j}^{n+1} = \frac{1}{4}(t_{i+1,j}^{n} + t_{i-1,j}^{n} + t_{i,j+1}^{n} + t_{i,j-1}^{n})$$

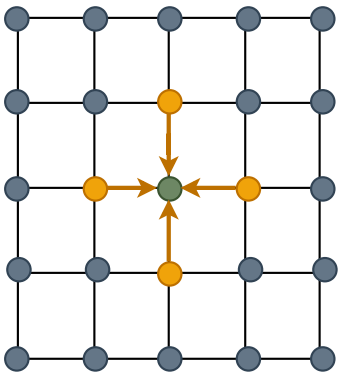
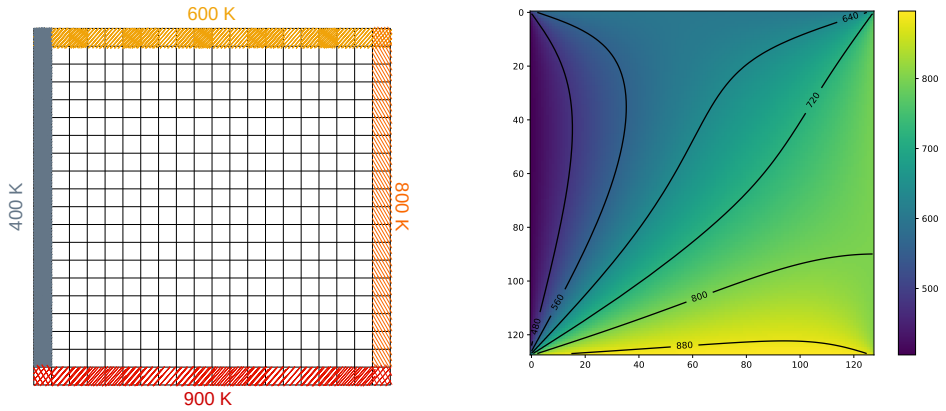


Figure 4.1.: Illustration of the discrete calculation of the Jacobi method, where the adjacent grid points (*yellow*) give the value to the midpoint (*green*).

In the case of our Boundary conditions, we assume to have Dirichlet conditions with the following values. A visual representation of this conditions, as well as the final converged solution can be seen in Figure 4.2.

$$T(0, y) = 400K \qquad T(x, 0) = 600K$$
$$T(L_x, 0) = 800K \qquad T(x, L_y) = 900K$$



(a) Sketch of the boundary conditions. White grid points have a Temperature value of $T = 0K$.

(b) Result of the solved heat equation once it has converged.

Figure 4.2.: Initial and final state of the Heat Diffusion Benchmark.

Finally, a quick overview of the implemented algorithm can be seen in Algorithm 1. It is clear that the most compute intensive part of the algorithm is found on the update of the grid points, i.e. the mean of the nearest neighbors values. Therefore, it is the operation we will focus during our study.

We expect that any result found with this simulation can be applicable to stencil based simulations where its compute-intensive part is also found on using the near neighbours on a grid or lattice. We will test this expectation on Chapter 5 where the Ising Model is implemented.

4.2.2 Wave Equation

The following hyperbolic PDE is used to simulate classical waves propagation and it is known as the Wave Equation.

$$\frac{\partial^2 U(x, y)}{\partial t^2} = c^2 \Delta U(x, y) \tag{4.13}$$

$U(x, y)$ being the displacement on the z axis and c a coefficient that can vary depending on the problem.

Algorithm 1: Jacobi Solver for the Heat Diffusion Problem

Input: The original state of the temperature t_0 and the convergence tolerance ϵ
Result: The converged temperature t distribution of the 2D grid
Initialize the temperature of the 2D grid and set a valid δ.

$$t \leftarrow t_0 \quad \delta \leftarrow \epsilon + 1$$

while $\delta > \epsilon$ **do**

Update the grid position as shown in (4.12)

$$t_{new} \leftarrow 0.25 \times (t_{north} + t_{south} + t_{east} + t_{west})$$

Compute the difference between consecutive states

$$\delta \leftarrow \sum_{i,j}^{N} |t_{new} - t|$$

Replace the grid with the new found state.

$$t \leftarrow t_{new}$$

end
return t

On this simulation, it is clear that the most computing intensive part are the second spatial derivatives found on the Laplacian. The Laplacian is useful when describing the flux or gradient flow of a function. Therefore, it is one of the most used PDEs on physics. Because of that, we expect the results obtained with this benchmark to have a large impact on the assessment of the viability of TPUs on physics workloads. As in the previous benchmark, we assume a fixed Dirichlet type boundary condition, even though in this case we consider them to be zero. However, we expect the analysis to be readily extended to other boundaries conditions.

$$U(0, y) = 0 \qquad U(x, 0) = 0$$
$$U(L_x, 0) = 0 \qquad U(x, L_y) = 0$$

Since we will use the Leapfrog integration method to obtain stable results, the complexity of the simulation will be larger than the used on the steady heat diffusion problem. We will be passing and updating the velocity as well as the position of the wave. Moreover, the simulation is very susceptible to its initial conditions. Therefore, we will use a smooth Gaussian with null velocity as its initial state as seen in Figure 4.3.

A summary of the sequence of steps on our implementation can be seen in Algorithm 2.

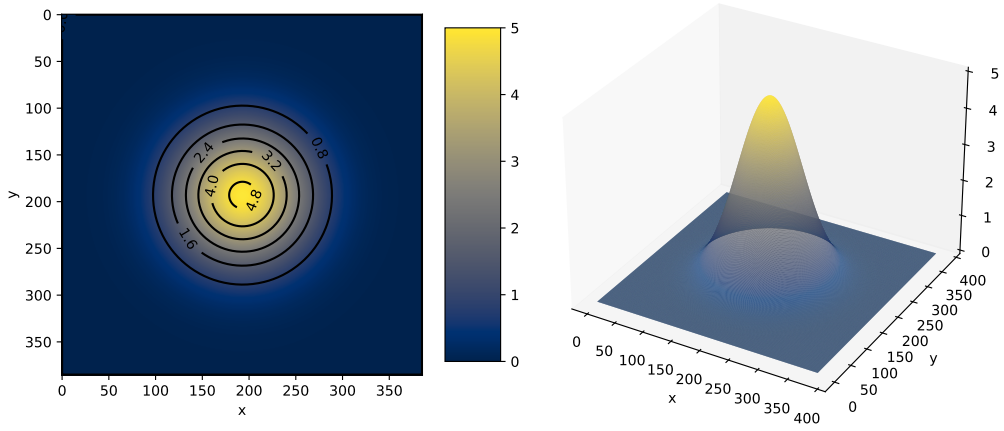


Figure 4.3.: Initial conditions for the the wave position in the form of a smooth Gaussian.

Algorithm 2: Simulation of Waves motion

Input: Gaussian parameters of the initial state μ , σ as well as integration steps Δt and h.

Result: Height of the waves on a 2D surface grid after $K\Delta t$ time.

Initialize the 2D grid positions with a smooth Gaussian and null velocities

$$u^0 \leftarrow \mathcal{N}(\mu, \sigma) \quad v \leftarrow 0$$

for $n \leftarrow 0$ **to** K **do**

Compute the acceleration with the Laplacian from the positions

$$a_{i,j} \leftarrow c^2 \Delta u_{i,j} \leftarrow \frac{-4 * u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j-1} + ui, j + 1}{h^2} \tag{4.14}$$

Update the velocity state

$$v_{i,j}^{n+1} \leftarrow v_{i,j}^{n} + a_{i,j}\Delta t \tag{4.15}$$

Update the position of the grid

$$u_{i,j}^{n+1} \leftarrow u_{i,j}^{n} + v_{i,j}^{n+1}\Delta t \tag{4.16}$$

end

return (u,v)

4.2.3 Shallow Water

The last benchmark is a simulation of the water motion after a drop has been placed on the middle of the 2D surface when the vertical distance is significantly smaller than the horizontal one. These motions follow the Shallow Water Equations. These equations

are derived from the Navier Stoke Equations when integrating over its depth as per the conditions just mentioned. These equations are used to describe situations where there the bottom of the water affects its motions, such as lakes, puddles and coasts. The latter case having a lot of relevance for the study of phenomena such as Tsunamis as SWE simulations are the foundation of any Tsunami warning system[37].

We assume a lake of uniform water density. For simplicity, Coriolis effects and nonlinear terms are also ignored. Lastly, we also assume a flat bottom ($h(x, y) \equiv h$). With these conditions, the Shallow Water Equation can be expressed as

$$\frac{\partial u(x,y)}{\partial t} = -g\frac{\partial \eta(x,y)}{\partial x} \tag{4.17}$$

$$\frac{\partial v(x,y)}{\partial t} = -g\frac{\partial \eta(x,y)}{\partial y} \tag{4.18}$$

$$\frac{\partial \eta(x,y)}{\partial t} = -h(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}) \tag{4.19}$$

where u and v are the velocity in the x and y direction respectively, η is the elevation of the surface on the z-axis and g is the gravity force.

In contrast to the previous simulations, this benchmarks computes the first derivatives on a successive manner. Moreover, to demonstrate different implementation of the methods, we will use the forward finite difference scheme to approximate these equations as follows, also shown in Appendix A.

$$u_{i,j}^{n+1} = u_{i,j}^{n} - \Delta t g\frac{\eta_{i,j+1}^{n} - \eta_{i,j}^{n}}{\Delta x} \tag{4.20}$$

$$v_{i,j}^{n+1} = v_{i,j}^{n} - \Delta t g\frac{\eta_{i+1,j}^{n} - \eta_{i,j}^{n}}{\Delta y} \tag{4.21}$$

$$\eta_{i,j}^{n+1} = n_{i,j} - \Delta t g h\frac{u_{i,j-1}^{n+1} - u_{i,j}^{n+1}}{\Delta x} - \Delta t h\frac{v_{i,j}^{n+1} - v_{i,j-1}^{n+1}}{\Delta y} \tag{4.22}$$

The model will be based on the Arkakawa C-grid, where a numerical grid is used in which the components of velocity are found between adjacent grid points.

Similarly to the WE benchmark, we start the system with a small perturbation on a flat surface in a Gaussian form as seen below.

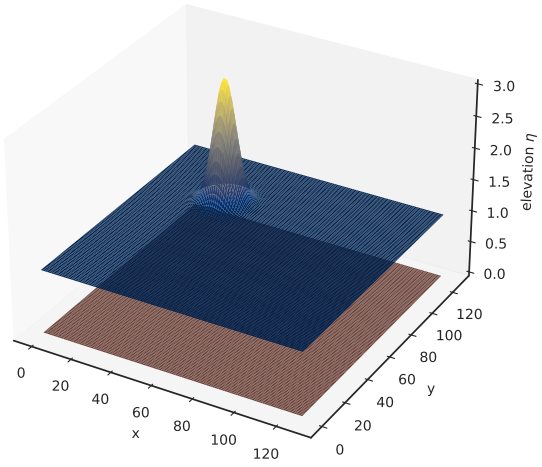


Figure 4.4.: Initial conditions for the Shallow Water simulation. Notice how the horizontal axes x and y are larger than the highest point of the Gaussian. The brown area below represents the sea floor.

4.3 Vector Operations

The initial approach we want to measure is the one that is probably already implemented on many physics simulations. As discussed on Chapter 3, the main reason that the Python scientific computing community is thriving nowadays is due to the use of array-based programming languages. In consequence, we will consider it to be the general current approach and the one we will compare the other methods with. But before that, we need to know how well it actually performs on TPUs.

4.3.1 Implementation

Vector programming is based on the idea of applying the same bitwise operation to whole chunks of memory at the same time. Thus, circumventing the expected low performance of dynamic loops on languages such as Python, where the computer can not predict what instructions will need. These slice operations shine the most when replacing element-wise operations usually done inside nested loops.

For instance, a basic Laplacian calculation, or any stencil computation for that matter, would typically be implemented by iterating over all the elements of the grid and storing the values on another grid with the same dimension. We will avoid mentioning padding

when not relevant to simplify the discussions. Consider the following snippet of a laplacian calculation in compiled language such as C.

```
for (int i=1; i<N-1; i++)
  for (int j=1; j<M-1; j++)
    lap[i][j]=-4*u[i][j]+u[i+1][j]+u[i-1][j]+u[i][j+1]+u[i][j-1]
```

it clearly resembles the notation used on (4.3). Likewise, if that code was to be run with the same nested loop approach on Python, as shown below, the expected performance would be horrible.

```
for i, j in itertools.product(range(1, N-1), range(1, M-1)):
    lap[i,j]=-4*u[i,j]+u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1]
```

As previously stated, this is due to Python runtime not knowing what is inside the loop and hence, not being able to prepare for it as a compiler would. By contrast, a vector approach to this same calculation would be express as:

```
lap = -4*u[1:-1,1:-1]+u[1:-1,2:]+u[-2:,1:-1]+u[:2,1:-1]+u[1:-1,:-2]
```

without the use of any loop. Regardless of the confusion caused by the slicing, we see that Python/NumPy allow for one-linears operations that would take 2 or more nested loops in C.

Despite the performance benefits of doing array operations being widely known by now, Table 4.1 shows the execution times of the previous snippets.

C	Python	NumPy	JAX with JIT (Compilation)
0.92 ± 0.04 ms	1220 ± 40 ms	4.13 ± 0.02 ms	0.32 ± 0.11 ms (254 ms)

Table 4.1.: Comparison between normal and array based computations on the same CPU (Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz). The table shows the timings of a Laplacian calculation on a grid of 1000×1000 random floating points values on the [0,100] range.

Looking at the results, we see that NumPy and JAX are in the same order of magnitude as C performance wise, while native Python performs way worse, which is why we see on these novel tensor units the opportunity to leverage high efficient array-based languages for HPC workloads. We expect this to be known to everyone currently making physics simulations on these languages. Thus, we assume the vector approach to be the current standard and the one to use as baseline.

Because index slicing can be confusing for people not used to work on array operations, Figure 4.5 illustrates the discussed Laplacian calculation. It also shows how simple and efficient vector operations really are and why they are so appealing to work with.

Python/NumPy uses views, i.e. references to the part of the array, instead of creating copies that would fill up memory, when working with slices. This is indeed a highly effective

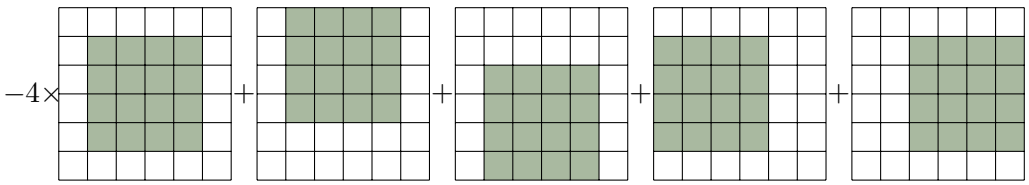


Figure 4.5.: Graphical representation of the slices used to compute the Laplacian as vectors without the use of loops. The green shade correspond to the subsection, or *view* of the original grid.

method to work with vector operations. However, as stated on Chapter 3, JAX arrays are immutable outside the JIT context. Therefore, if we do not want to store 5 copies of our $N \times N$ array every time step and create a memory bottleneck in the process, we need to make use of the XLA compiler. We need to indicate that the grid is going to be overridden by the returning value and that its memory address can be reuse. Hence, we ought to be thoughtful on using the correct JIT parameters such as `donate_argnums` and `static_argnums` when implementing our code.

The discussion of the Laplacian calculation can trivially be extended to the stencil operations on the heat diffusion problem, as well as the first derivatives on the Shallow Water benchmark. The solver function of the Heat Equation implementation can be seen on Listing 4.1. In spite of using some of JAX main functions discussed on Chapter 3 such as `jax.jit` and `lax.while_loop`, the implementation should look familiar to any Matlab/NumPy user.

```
1  from jax import ops, lax, jit, partial
2  import jax.numpy as jnp
3
4  def stencil_op(x):
5      north  = x[1:-1, :-2]
6      south  = x[1:-1, 2:]
7      east   = x[:-2, 1:-1]
8      west   = x[2:, 1:-1]
9      return 0.25 * (north + south + east + west)
10
11 def solver_step(x_prev):
12     x = x.at[1:-1,1:-1].set(stencil_op(x_prev))
13     delta = jnp.sum(jnp.abs(x - x_prev))
14     return x, delta
15
16 @partial(jit, donate_argnums=0)
17 def solve(x, epsilon):
18     cond = lambda s: s[1] > epsilon
19     loop_func = lambda s: solver_step(s[0])
20     return lax.while_loop(cond, loop_func, (x, epsilon+1))
```

Listing 4.1: Vector implementation of the Heat Equation solver

One caveat of using JIT however is that we can not know how it is going to handle the memory allocation. Using the recently released memory profiler on TPU, we can visually check that the memory footprint of the `solve` function shown is $3\times$ the amount of the original grid.

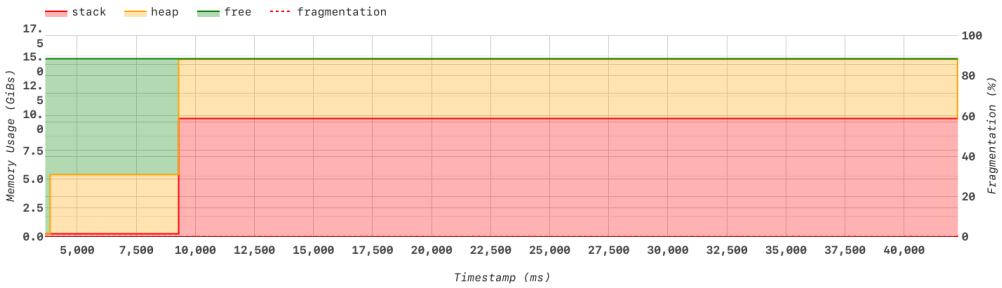


Figure 4.6.: Memory profiler snapshot of the heat diffusion simulation. The initial conditions array represent the initial heap allocation while the jump at $\sim 9s$ indicates when the `solver` functions is actually run. The stack area indicates the temporal memory that XLA uses to perform the operations where as the heap is the memory allocated to the user defined initial grid.

Multi-Core Implementation

For the multi-core case, as seen on Chapter 3, we utilize the tools provided by JAX that allow for high-speed message passing between XLA devices without having to go through the host. On the current method, all three benchmarks have a fairly similar approach.

Because we are implementing operations that only rely on nearest neighbors, all three benchmarks will deal with the same communication issues. Due to the way slicing is intended to work, we need to use $N+2 \times N+2$ arrays, that way, when we want to get the element $i+1$, we can use the slice `[:-2,...]`, but the last element would be missing its element-wise operation. That is why we need the padding. Nevertheless, for our case, the padding is perfect, as we can perform a halo exchange before each calculation so that the values are propagated across devices. Figure 4.7 shows how the halo exchange between multiple devices works. We consider the boundary rows/columns to be ghosts, meaning that they will only hold the information that is being passed from other devices or set as the fixed boundary conditions, but the stencil operation will not alter them.

We create a set of functions that would allows us to adapt to any core topology. To do that, we use `CollectivePermute` XLA operation, as it allow send and receive data across devices. Considering that we do not have control on how the physical TPU cores are distributed, we create a logical layout on our code. Even though one could implement its own method to choose a good locality of the devices so that the code matches the logical layout, the connection between cores inside the same TPU Unit is indistinguishable. For that reason, we have decided not to dwell on it and just provide generic functions that would perform

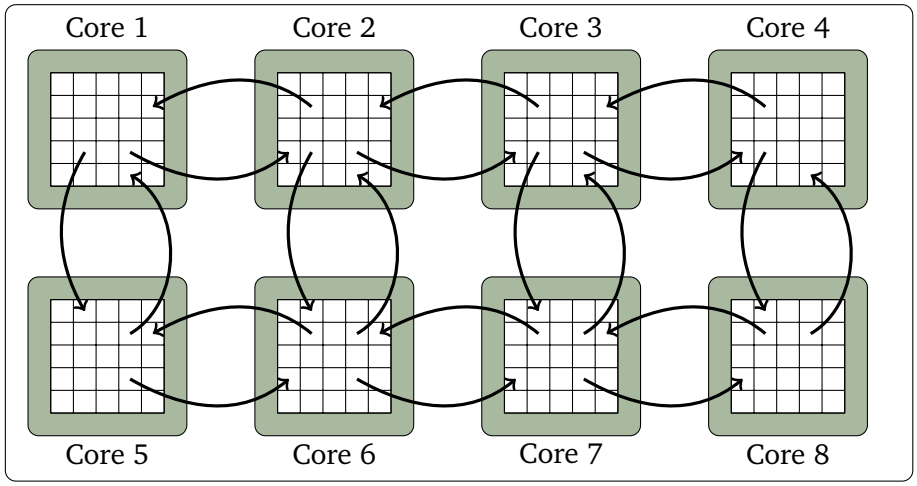


Figure 4.7.: Illustration of the halo exchange of each grid with their logical neighbors cores. This figure shows a 2×4 topology.

```
from jax import lax
from itertools import product

def send_up(boundary, nrows, ncols, axis_name='x'):
    pair = lambda i,j: ((i+1) * ncols + j, i * ncols + j)
    goup = [pair(i,j)
            for i,j in product(range(nrows-1),range(ncols))]
    return lax.ppermute(boundary, perm=goup, axis_name=axis_name)

def send_left(boundary, nrows, ncols, axis_name='x'):
    pair = lambda i,j: (i * ncols + j + 1,  i * ncols + j)
    goleft = [pair(i,j)
              for i,j in product(range(nrows),range(ncols-1))]
    return ppermute(boundary, perm=goup, axis_name=axis_name)
```

Listing 4.2: Helper functions to do array permutation between Cores (or XLA devices). These permutations consider no periodic boundaries.

this operations in every topology. For instance, Listing 4.2 shows the implementation of sending the top row to the core above on `send_up` and the left column to the core on the left on `send_left`. Those implementation are thought to be used for Dirichlet conditions. The devices on the boundaries will receive an empty array instead. If, for instance, we would like to have periodic conditions, we would make the last element of each axis to form a pair with the first element on the left. Assuming the layout of Figure 4.7, we would need to add $(5, 8)$ and $(1, 4)$ to the `pair` list on `send_left`.

On our implementation of the Laplacian, we do not want to exchange the boundaries of the grid, but the rows and columns next to the padding, the `step` function includes the `exchange_boundaries` before performing the Laplacian calculation. It is important that we call the `exchange_boundaries` function inside the JIT context, not only because we need it to be run on the accelerator but to avoid extra copies of the array `x` on every index update.

```python
from jax.ops import index

def exchange_boundaries(x, nrows, ncols, axis_name='x'):
    top = send_down(x[-2,...], nrows, ncols, axis_name)
    bottom = send_up(x[1,...], nrows, ncols, axis_name)
    left = send_right(x[...,-2], nrows, ncols, axis_name)
    right = send_left(x[...,1], nrows, ncols, axis_name)

    x = x.at[0,...].set(new_top)
    x = x.at[-1,...].set(new_bottom)
    x = x.at[...,0].set(new_left)
    x = x.at[...,-1].set(new_right)
    return x
```

Listing 4.3: Helper function to exchange the halo boundaries between cores.

Of course, it should be said that the exchanging of boundaries varies between simulations. Despite having the same core concept–i.e. exchange ghost cells before using them on the slicing, simulations such as the shallow water have no need for exchanging both sides, since the calculation is between $i+1$ and i or $j+1$ and j.

4.3.2 Results

We start by simulating different grid sizes to compare how they perform on different hardware. Figure 4.8 shows the speedup obtained when using a GPU, a TPU Core, a TPU Chip (2 Cores) and a whole TPU Unit (4 Chips) with respect to a CPU (AMD EPYC 7501 @ 2GHz). We see that the TPU Unit achieves a speedup of $350\times$, vastly outperforming the maximum speedup found on the GPU by $1.8\times$.

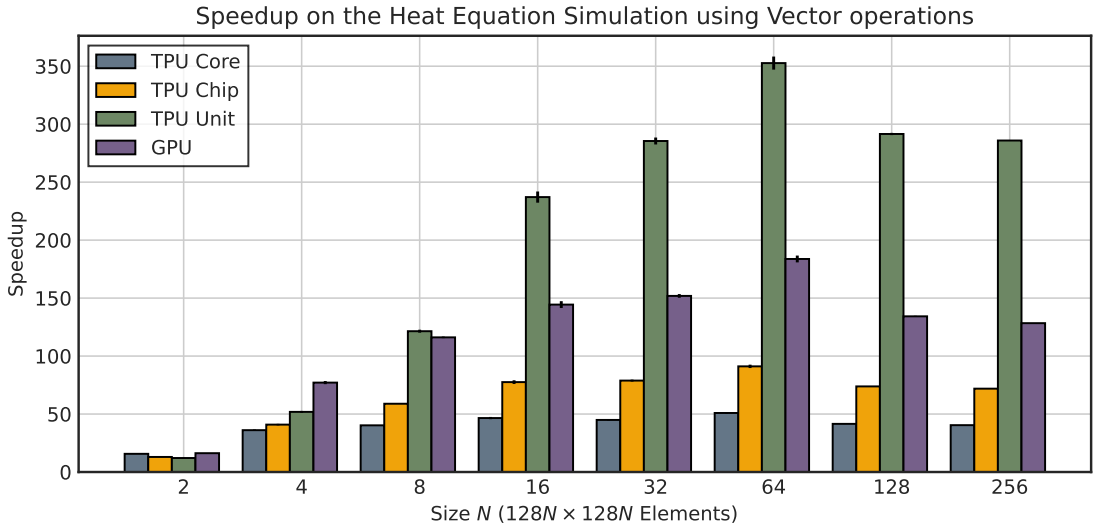


Figure 4.8.: Speedups on the running time on different accelerators of the Wave Equation Benchmark using Vector/Slice Operations.

However, it is not fair to compare the raw power of a TPU Unit against a single GPU, since a GPU resembles more to a single TPU Chip in terms of specifications as it can be seen on the Table B.1 from Appendix B. By looking at the results of the TPU Chip, it is clear that a GPU does yield faster iteration times with a fairly consistent $1.8\times$ speedup regardless of the problem size.

In order to get a clearer picture, we proceed to take into account the energy consumption of each device as seen in Figure 4.9. As it was not possible to get access to Google's internal energy consumption metrics, we are going to assume the specified TDP as its power usage, and because only the TDP of a Chip has been disclosed, we will also assume that it scales linearly with the number of Chips/Cores. For this approach to be reasonable, we will assume that in order for the chip to be using that amount of power, it needs to be at a maximum usage of its HBM. If the amount of operations and data handling is large enough, we can expect that it would be similar to the real application a TPU is prepared to handle. It is important to state that because we are not using all the processing units inside the TPU Core for this method, the results should be considered a very rough estimate, as we can not expect to reach the TDP usage.

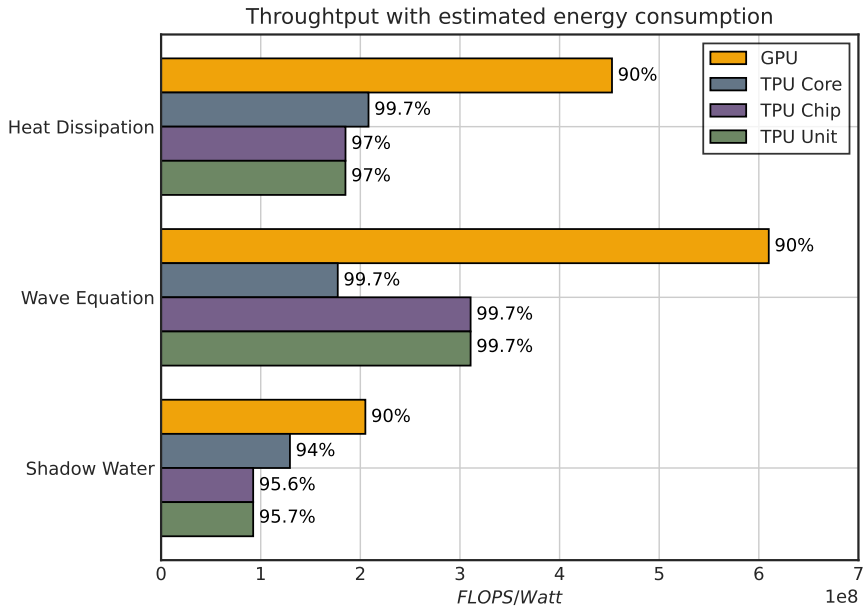


Figure 4.9.: Performance/Efficiency comparison between the different devices on the vector method. The percentage value shown in each bar is the memory usage of the device.

Looking at the values from Figure 4.9, it is clear that TPUs are no match to GPUs on vector operations. Not even considering that TPUs were designed with efficiency in mind. Of course, that is not a surprise. As explained in Section 2, the most powerful unit inside a TPU core is not its VPU, but the MXU. Thus, saying that the GPU, an accelerator designed specifically to perform vector operations, is faster than an misutilised TPU is expected. Regardless of it, using a whole TPU unit does yield an impressive performance, despite

not using its most powerful feature. Meaning the VPU does not hold back on performance either and can handle vector operations rather efficiently.

Another aspect to evaluate when comparing TPUs and GPU is their economic factor. Due to only being available on Google Cloud at the moment, we can use their current pricing to evaluate their performance on terms of amount of raw power a dollar can get you. Assuming that Google is still making profit on both scenarios, the pricing can be a seen as a more realistic measurement of their maintenance cost. Hence, if we only focus on the options with a price tag and we compute the number of FLOPS per dollar spent, we obtain Figure 4.10. The results now are much closer, despite the GPU still outperforming the whole Unit once normalized.

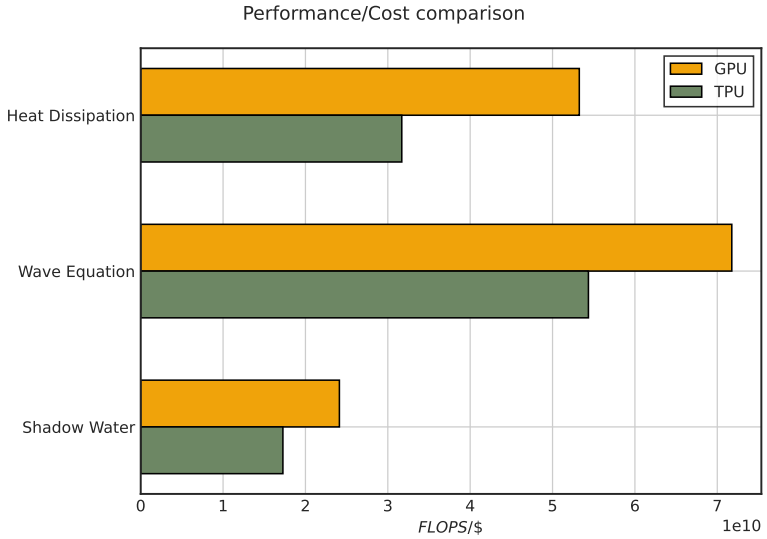


Figure 4.10.: Performance/Cost comparison between a GPU and a TPU Unit

By looking at the resulting charts, it should be clear that when doing vector operations, GPUs should be the target accelerator. However, it also should be stated that we are comparing an auxiliary unit of the TPU to the main GPU feature, and yet we still get a decent performance for the money. Regardless, a TPU is not the best solution to run vectorized calculations in terms of Efficiency. Therefore, we need to look into methods that can leverage the MXUs on each TPU Core.

4.4 Matrix Multiplication

In order to make us of the systolic arrays inside each core, we need to reformulate the finite difference method. Since only matrix multiplications will be fed to the MXU, the most obvious and straightforward solution is to express it in terms of a system of equations. Luckily, the finite difference method can be trivially expressed as one.

4.4.1 Implementation

The second derivative shown in (4.3) can be rewritten into the following system of equations:

$$\begin{aligned} h^2 D_{xx} u_1 &= -2u_1 + u_2 \\ h^2 D_{xx} u_2 &= u_1 - 2u_2 + u_3 \\ &\vdots \\ h^2 D_{xx} u_{N-1} &= u_{N-2} - 2u_{N-1} + u_N \\ h^2 D_{xx} u_N &= u_{N-1} - 2u_N \end{aligned}$$

Hence, it can be expressed on the matrix form $Ax = b$ as seen on (4.9).

$$\begin{pmatrix} -2 & 1 & 0 & \dots & 0 & 0 & 0 \\ 1 & -2 & 1 & \dots & 0 & 0 & 0 \\ & \vdots & & \ddots & & \vdots & \\ 0 & 0 & 0 & \dots & 1 & -2 & 1 \\ 0 & 0 & 0 & \dots & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix} = \begin{pmatrix} D_{xx} u_1 \\ D_{xx} u_2 \\ \vdots \\ D_{xx} u_{N-1} \\ D_{xx} u_N \end{pmatrix} \tag{4.23}$$

Similarly, we can apply the same formulation to first derivatives (4.8)

$$\begin{aligned} h D_x u_1 &= -u_2 \\ h D_x u_2 &= u_1 - u_3 \\ &\vdots \\ h D_x u_{N-1} &= u_{N-2} - u_N \\ h D_x u_N &= u_{N-1} \end{aligned} \qquad A_{D_x} = \begin{pmatrix} 0 & -1 & 0 & \dots & 0 & 0 & 0 \\ 1 & 0 & -1 & \dots & 0 & 0 & 0 \\ & \vdots & & \ddots & & \vdots & \\ 0 & 0 & 0 & \dots & 1 & 0 & -1 \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 \end{pmatrix}$$

Despite being used extensible in linear algebra, matrix multiplications are considered a fairly expensive method to run on conventional computing systems and are usually avoided if possible. However, because TPUs are designed specifically to compute them without any overhead, it begs the question of whether they will yield competitive results not found on other accelerators.

As in the previous section, we take a look at the 2 dimensional Laplacian computation. Using matrix multiplications, we solve the whole system in two consecutive operations by using (4.23) for each axis. We can see an example on Listing 4.4, where the laplacian function has been rewritten to use only matrix multiplications. Consequently, we have an implementation of the finite difference method that mostly uses the MXU and can be expressed in a fairly simple manner.

```
import jax
import jax.numpy as jnp

  A = jnp.eye(N, k=-1) - 2 * jnp.eye(N, k=0) + jnp.eye(N, k=1)

  @jax.jit
  def laplacian(x, A, h):
    L = jnp.dot(x, A) + jnp.dot(A, x)
    return L / h**2
```

Listing 4.4: Laplacian implementation in terms of matrix multiplications.

This reformulation gets a bit trickier on the Shallow Water simulation, where we are doing the forward difference scheme. If, in order to save memory, we were only to define a single matrix A , we would do it on the following way

$$A_{D_x} = \begin{pmatrix} -1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 1 & -1 & 0 & \dots & 0 & 0 & 0 \\ & \vdots & & \ddots & & \vdots & \\ 0 & 0 & 0 & \dots & 1 & -1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 & -1 \end{pmatrix}_{NxN} \tag{4.24}$$

In contrast to the Laplacian and stencil matrices, $A^T \neq A$. Hence, in order to use the same matrix A for all the operations, we need to transpose it and change its sign. The resulting snippet of shows how the implementation would work

```
import jax.numpy as jnp

A =  jnp.eye(nx, ny, k=-1) - jnp.eye(nx, ny, k=0)

def integrate_step(state, A, depth, g, dt, dx):
    e, u, v = state
    u = u - dt / dx * g * jnp.dot(e, A)
    v = v - dt / dx * g * jnp.dot(A.T, e)
    e = e - dt / dx  * depth * (jnp.dot(u,-A.T) + jnp.dot(-A,v))
    return e, u, v
```

Listing 4.5: Shallow water integration step using matrix multiplications

As previously stated, the Laplacian computation using vector operations uses twice the amount of memory required as it can make use of references to avoid redundancy. That is not the case for our matrix multiplication. We need A in order to perform our computation.

We know A to be a tridiagonal matrix, which becomes a sparser matrix the more elements the grid contains. Therefore, we will be storing a $N \times N$ matrix, mostly filled with zeros in order to perform the calculation. Unfortunately for this method, TPUs systolic array will only perform dense matrix multiplications, which means that all the operations that involve sparse matrices and multiplying element by 0s are going to be wasted resources.

Another limitation that this approach presents is that we are using As with square shape in order for the matrix multiplication to work. By definition, the first component on the multiplication should have the same number of columns than rows has the second one. It does not need them to be squared, but because we want the operation to be performed on all the elements of the matrix, we require them to be square as well. This leaves us with a constraint that only square grids per device can be solved with by this approach.

Boundary Conditions

We believe it is important to add a note on how the implementation is affected by a different boundary condition. Even though we have assumed Dirichlet conditions, the boundary conditions would also affect the matrix A or the RHS of (4.23) and not only the grid as it is may be the case in the vectorize method from Section 4.3.

For instance, if instead of Dirichlet condition we were to have a Periodic system, we would be using

$$A_{\text{periodic}} = \begin{pmatrix} -2 & 1 & 0 & \dots & 0 & 0 & 1 \\ 1 & -2 & 1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & -2 & 1 \\ 1 & 0 & 0 & \dots & 0 & 1 & -2 \end{pmatrix}_{N \times N} \tag{4.25}$$

These changes only affect when dealing with a single device however, as the message passing of the boundaries between devices can not be done using the matrix.

Multi-core Implementation

When scaling to more cores, we can reuse the functions from the vector method described in Section 4.3 as we are going to be exchanging only boundaries again. However, by the nature of matrix multiplications, we will not be loosing two elements on each axis after the operations. Therefore, we would like to avoid having to call padding every integration step.

Since we still need to update the boundaries, the elements on the boundaries are going to be missing the value of one of their neighbours after doing the multiplication. Thus, we need to include the value afterwards. We do it by using `jax.ops.index_add` to modify the already calculated value.

```
import jax.numpy as jnp

def integrate_step(...):
    u, v = state

    # Send boundaries
    bottom = send_up(u[0,:], nrows, ncols)
    top = send_down(u[-1,:], nrows, ncols)
    left = send_right(u[:,-1], nrows, ncols)
    right = send_left(u[:,0], nrows, ncols)

    a = c ** 2 * laplacian(u, A, h)

    # Add the adjusted values
    a = a.at[0,:].add(top/h**2)
    a = a.at[-1,:].add(bottom/h**2)
    a = a.at[:,0].add(left/h**2)
    a = a.at[:,0].add(right/h**2)

    v = v + a * dt
    u = u + v * dt
    return u, v
```

Listing 4.6: Water Equation integration step adapted for message passing.

which should be fairly familiar to anyone who has had to work with the Message-Passing-Interface Protocol.

4.4.2 Results

Comparing the speed of applying this method similarly to what we did with the vectorized operations we obtain the following results, shown in Figure 4.11. In this case, we use the speedup with respect to the GPU (NVIDIA RTX 2080Ti) instead of the CPU used previously. By now we should be aware of the massive improvements we get from using accelerators. As stated above, this method only works for square lattices, hence in our distributed approach, we do not repreduce the same system, as a (N, N) square can not be split into 8 square matrices. Instead, we operate on squares matrices of size (L, L)that, combined, contain the same number of elements as the (N, N) original square. To do so, we apply the relation $N = \sqrt{P}L$, where P indicates the number of XLA devices.

Regarding the chart, we see that matrix multiplications on the TPUs core perform better than on the GPU. Not only better, but we manage to get up to $\sim 3\times$ times faster with a single core and up to $\sim 400\times$ speedup when using the whole unit, as the split also removed

a lot of the sparsity of the array. As we were expecting, these results confirm us that matrix multiplications hold the key to use TPUs efficiently, as their performance can be matched with GPUs.

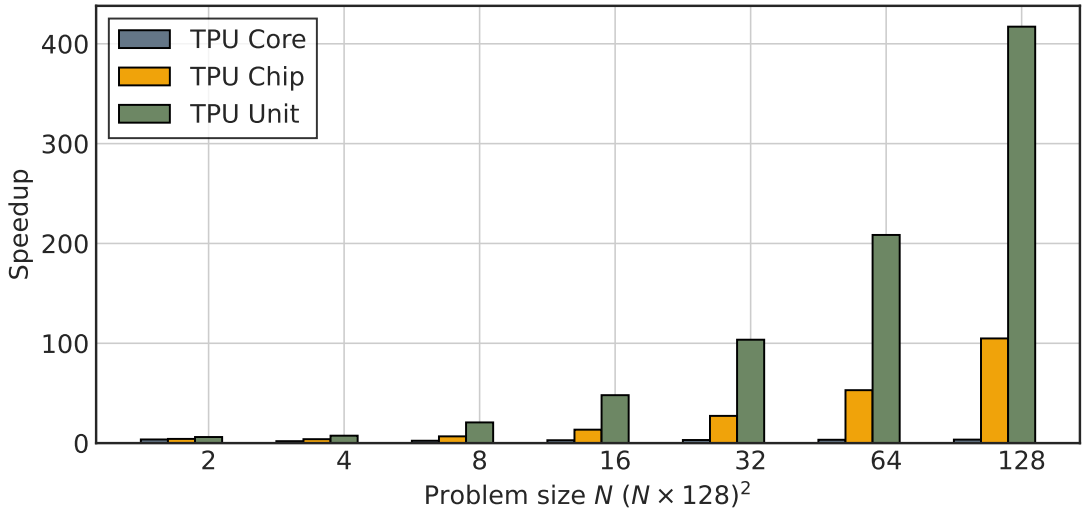


Figure 4.11.: Speedup graph of different amounts of Cores in comparison with a GPU run. All the devices are using the same matrix implementation.

Despite the good results of the TPU with respect to the GPU, when comparing the values from the ones obtained on the vectorized approach, we observe that after the $(16 \times 128, 16 \times 128) = (2048, 2048)$ grid size we stop getting any speedup and instead, it decreases considerably, making the vector approach more appropriate for larger sizes. That result is actually expected. As we have discussed, A is only getting sparser the more elements we have. Therefore, the sparsity of A makes the method useless for large scale simulations.

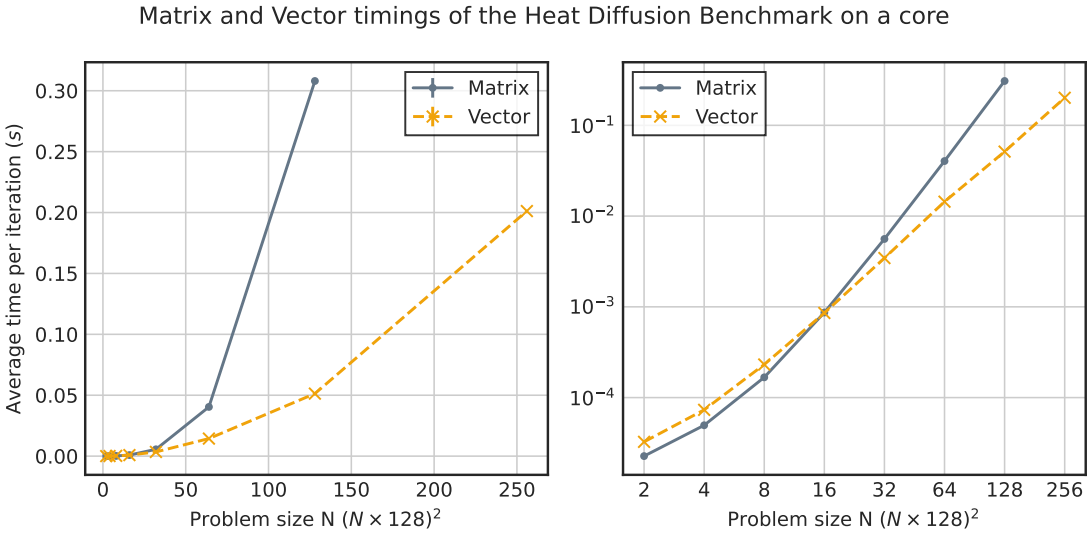


Figure 4.12.: Performance of the Heat Diffusion Problem using matrix multiplications as we increase the problem size.

4.5 Tiles Matrix

In spite of the speedups observed with small sizes with respect to their vector counterpart, it is clear that using massive sparse matrices is not a feasible solution for large-scale simulations. Due to the locality of the FDM and some properties of matrix multiplications, we may find a way to avoid it.

4.5.1 Implementation

As stated in Chapters 2 and 3, the current generation of MXUs found inside each TPU core has 128 PE and the XLA compiler will pad the arrays with 0 up until a multiple of 128 in order to use it. Therefore, we expect that doing matrix multiplications with matrices smaller than 128×128 is not going to be very efficient. However, if we do a matrix product for arrays multiple of 128, the systolic array is going to be used optimally. Since distant grid points do not depend on each other when computing the partial derivative through the finite method, we are in an ideal position to split the grid in smaller grids of size $n \times n$ with n being a multiple of 128. Therefore, creating a grid of tiles,where each can then be computed independently using a $n \times n$ matrix A as seen in Figure 4.13.

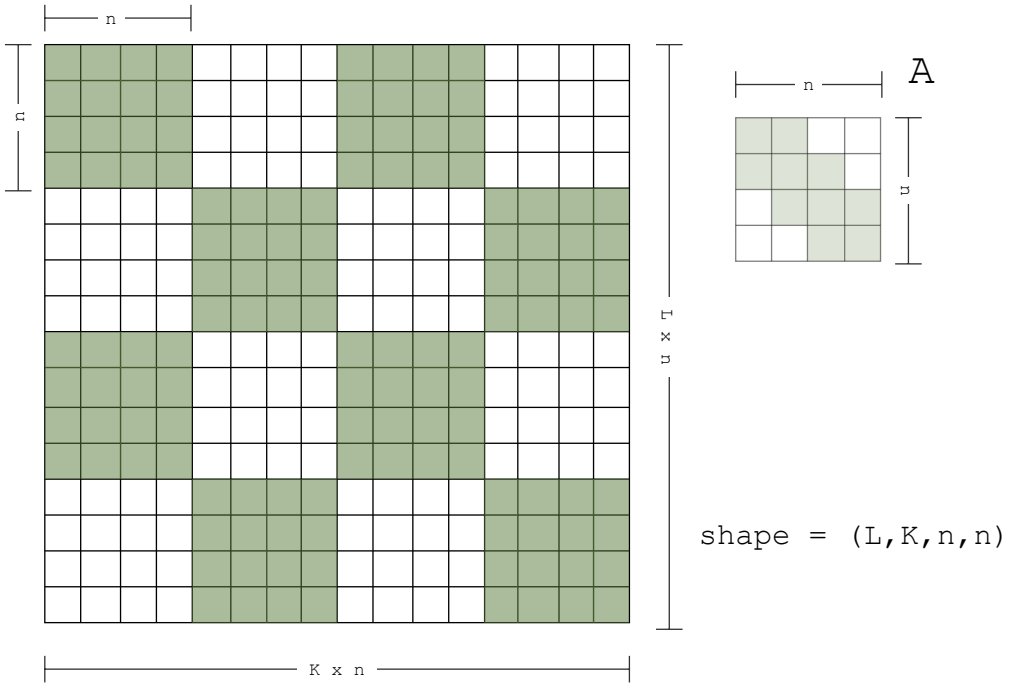


Figure 4.13.: Illustration of the slip of the grid into mini-grids. The size of the stencil matrix is also added to show the memory savings of this approach.

An obvious drawback from this approach is that now we need to update boundaries between tiles similarly to what we have done on the distributed approach of the simple matrix multiplication method. Nevertheless, we expect that cost of boundary exchange in

elements of the same core will be outperformed by the boost of not dealing with massive sparse matrices.

The change on our implementation does convey some tricks that needed to be worth mentioning. It should be clear that our working grid will become a 4 dimensional array, of size $L \times K \times n \times n$, where $L = N/n$ and $K = M/n$ being N and M being the size of the original grid. We also need to be sure that the reshape function works as we need to. The correct split into tiles can be obtained using the following code

```
def tile_split(x, n):
    L, K = x.shape
    return x.reshape(L//n, n, K//n, n).swapaxes(1, 2)

def merge_tiles(x, N, M)
    return x.swapaxes(1, 2).reshape(N, M)
```

Listing 4.7: Auxiliary functions to transform correctly a grid into tiles and vice versa.

We add the exchange on inner core boundaries to the `stencil_op` as seen in Listing 4.8. We also describe our matrix multiplication as

$$D_{xx}u_{i,j,k,q} = \sum_{l=0}^{n} u_{i,j,k,l} \cdot A_{l,q} \tag{4.26}$$

```
import jax.numpy as jnp
from jax import ops

# size of x is (L,K,n,n)
A = jnp.eye(n, k=-1) + jnp.eye(n, k=1)

def stencil_op(x, A):
    c = jnp.matmul(A, x) + jnp.matmul(x, A)
    c = c.at[1:,:,0,:].add(x[:-1,:,-1,:])
    c = c.at[:-1,:,-1,:].add(x[1:,:, 0,:])
    c = c.at[:,1:,:, 0].add(x[:,:-1,:,-1])
    c = c.at[:,:-1,:,-1].add(x[:,1:,:, 0])
    return 0.25 * c
```

Listing 4.8: Step function on the heat diffusion problem for the tiles approach.

The implementation above assumes Dirichlet conditions. Some difference to the exchange boundaries should be applied for periodic conditions, similarly to a multi-core approach.

With this implementation, the restriction of a square matrix gets lifted. Albeit N_x and N_y should be multiples of n in order to get the split correctly, but $N_x \neq N_y$ is possible.

Multi-core Implementation

Regarding the multi-core implementation, we are in a very similar situation to the case on Section 4.4, where we will add the boundary values after the stencil operations has been carried. The only main difference been that we are now using 4D arrays, so for instance, when exchanging the top row, we will need to send the top row of all the mini-grids on the top row of the large grid. For instance, a small modification should be done to Listing 4.9 such as

```
import jax.numpy as jnp
from jax import ops

def integrate_step(...):
    # Send boundaries
    bottom = send_up(u[0,:,0,:], nrows, ncols)
    top = send_down(u[-1,:,-1,:], nrows, ncols)
    left = send_right(u[:,-1,:,-1], nrows, ncols)
    right = send_left(u[:,0,:,0], nrows, ncols)

    a = c ** 2 * laplacian(u, A, h)

    # Add the adjusted values
    a = a.at[0,:,0,:].add(top/h**2)
    a = a.at[-1,:,-1,:].add(bottom/h**2)
    a = a.at[:,-1,:,-1].add(left/h**2)
    a = a.at[:,0,:,0].add(right/h**2)

    v = v + a * dt
    u = u + v * dt
    return u, v
```

Listing 4.9: Water Equation integration step adapted for message passing.

We see that indexing starts to get a bit confusing as we increase the dimensions of the array.

4.5.2 Results

Before comparing the results with the other methods, we need to decide on which size of tiles would be optimal for our solvers. Rationally, the larger the size the more sparse the matrix is going to be, so we want to remain on the lower side of the sizes. Due to the physical limitation of 128 PE, n should not be under 128. Therefore, we proceed to do a grid search on different values of n for different problem sizes. Table 4.2 shows the speedups of the three benchmarks for the new tiles implementation with respect to the vector implementation.

Looking at Table 4.2, we see that we are able to achieve a solid 20% speedup with respect to the vector approach. The values oscillate quite a bit and that may due to the sample size.

Grid \ Tile	$(1 \times 128)^2$	$(2 \times 128)^2$	$(4 \times 128)^2$	$(8 \times 128)^2$	$(16 \times 128)^2$	$(32 \times 128)^2$	$(64 \times 128)^2$
	Heat Diffusion						
$(2 \times 128)^2$	0.84	**1.43**	-	-	-	-	-
$(4 \times 128)^2$	0.78	1.04	**1.44** -	-	-	-	
$(8 \times 128)^2$	0.71	0.96	1.12	**1.20**	-	-	-
$(16 \times 128)^2$	0.68	0.93	1.18	**1.21**	0.89	-	-
$(32 \times 128)^2$	0.69	0.90	**1.21**	1.19	0.82	0.54	-
$(64 \times 128)^2$	0.72	0.93	1.23	**1.24**	0.86	0.53	0.31
$(128 \times 128)^2$	0.64	0.83	1.10	**1.18**	0.76	0.47	0.26
$(256 \times 128)^2$	0.63	0.81	1.07	**1.15**	0.74	0.47	0.26
	Wave Equation						
$(2 \times 128)^2$	0.89	**1.89**	-	-	-	-	-
$(4 \times 128)^2$	0.73	1.06	**1.57**	-	-	-	-
$(8 \times 128)^2$	0.59	0.84	1.09	**1.16**	-	-	-
$(16 \times 128)^2$	0.53	0.75	0.97	**1.15**	0.75	-	-
$(32 \times 128)^2$	0.51	0.66	0.93	**0.99**	0.68	0.40	-
$(64 \times 128)^2$	0.69	0.91	1.20	**1.27**	0.92	0.56	0.30
$(128 \times 128)^2$	0.69	0.91	1.20	**1.27**	0.92	0.58	0.32
	Shallow Water						
$(2 \times 128)^2$	1.16	**2.18**	-	-	-	-	
$(4 \times 128)^2$	1.11	1.51	**1.93**	-	-		
$(8 \times 128)^2$	0.86	1.18	**1.47**	1.20	-	-	
$(16 \times 128)^2$	0.95	1.29	**1.65**	1.49	-	-	
$(32 \times 128)^2$	0.88	0.94	1.20	**1.62**	1.52	-	-
$(64 \times 128)^2$	1.15	1.47	**1.91**	1.86	1.07	0.58	0.27
$(128 \times 128)^2$	0.96	1.52	**1.96**	1.91	1.10	0.57	0.27

Table 4.2.: Speedups of the tiles approach with respect to the vector approach for the three benchmarks.

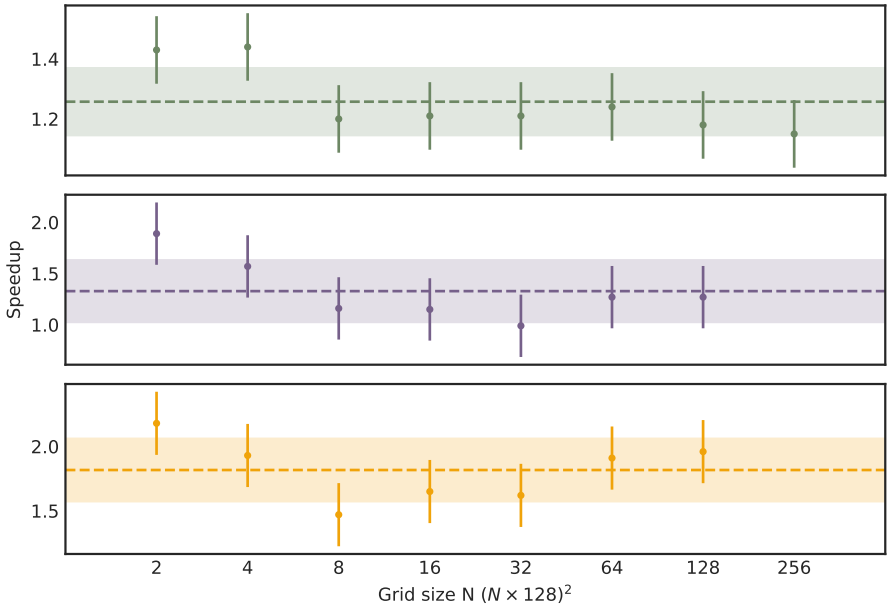


Figure 4.14.: Observations of the speedups for different problem sizes.

The reason behind the different results when the grid size is $(32 \times 128)^2$ are still unknown, but the fact that ig happens in the three simulations begs the question if something strange is happening on the device. Due to times constrains, each simulation has only been run 50 times in order to get the the final mean time. Regardless, the point that tiles larger than $\simeq 1000$ start to decrease the performance of the core should remain.

Figure 4.14 shows us that the speedup factors are not related to the problem size.

4.6 Direct Solution by Tensor Product

One of the reasons of the surge of AI accelerators is that matrix multiplications have not been the most efficient method to implement on conventional hardware, despite the impressive work done on libraries such as BLAS, which is the backbone of most the the GEMM operations in high level libraries such as NumPy. However, a TPU performs the best when dealing with matrix multiplications as seen from previous benchmarks. Those results gives us the opportunity to study methods that rely on the use of tensor products to obtain exact solutions to PDEs, which would leverage the use of the MXU without having to perform any custom transformation. Therefore, we will briefly introduce the method described in Lynch *et al.* [38], which we will compare its performance on solving the heat dissipation problem to the results from the previous iterative methods, as well as with the current hardware similarly to what have been doing during this whole section.

4.6.1 Implementation

The main idea of the method is to obtain a system than can be expressed on the following form

$$(I \otimes A + B \otimes I)u = g \tag{4.27}$$

where A and B are $N \times N$ matrices and $\otimes$ indicates the Kronecker Product.

The Kronecker product, also referred as tensor product, of two matrices $A \otimes B$ returns a $NM \times NM$, where A is of order $N \times N$ and B is of order $M \times M$. For instance, if we define $a_{i,j}$ as the elements of the matrix A, the resulting tensor product of $A \otimes B$ would be

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & a_{13}B & \cdots & a_{1N}B \\ a_{21}B & a_{22}B & a_{23}B & \cdots & a_{2N}B \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{N1}B & a_{N2}B & a_{N3}B & \cdots & a_{NN}B \end{pmatrix}$$

Thus, the method relies on being able to express the PDE to study in the form of (4.27) so that it can be directly solved. To do so, the inverted matrix $(I \otimes A + B \otimes I)^{-1}$ is obtained by using the tensor product properties as well as their eigenvectors and eigenvalues.

We will start by assuming that we have a system described in the form of (4.27). We will also assume A and B to be non-singular matrices and to have distinct eigenvalues. This conditions should be reevaluated when applying the method to an specific PDE. Hence, we can expect the eigenvectors matrices of A and B to exist and fulfil the following relations

$$P^{-1}AP = \lambda(A) \tag{4.28}$$

$$Q^{-1}BQ = \lambda(B) \tag{4.29}$$

where P and Q are the matrices with the eigenvectors of A and B respectively.

Therefore, by using the properties of the tensor product, we can express (4.27) in terms of the eigenvectors and eigenvalues of A and B. To do so, we apply the previous relations to the LHS so that we have the following expression

$$P^{-1} \otimes Q^{-1}(I \otimes A + B \otimes I)P \otimes Q = I \otimes \lambda(A) + \lambda(B) \otimes I \tag{4.30}$$

thus, using the elemental properties of tensor products we can express the inverse as

$$(I \otimes A + B \otimes I)^{-1} = P \otimes Q(I \otimes \lambda(A) + \lambda(B) \otimes I)^{-1} P^{-1} \otimes Q^{-1} \tag{4.31}$$

Consequently, obtaining the solution of u from (4.27) as

$$u = P \otimes Q(I \otimes \lambda(A) + \lambda(B) \otimes I)^{-1} P^{-1} \otimes Q^{-1} g \tag{4.32}$$

where g contains the information regarding the initial conditions as well as the boundary conditions. In order to make it more clear, we can rewrite (4.32) in terms of the matrix elements

$$u_{i,j} = \sum_{\alpha=1}^{N} p_{i,\alpha} \sum_{\beta=1}^{M} q_{j,\beta} \frac{1}{\lambda_\alpha(A) + \lambda_\beta(B)} \sum_{n=1}^{N} p'_{\alpha,n} \sum_{m=1}^{M} q'_{\beta,m} g_{n,m} \tag{4.33}$$

which is nothing more than 4 consecutive matrix multiplications as seen below

$$R = r_{n,\beta} = \sum_{m=1}^{M} q'_{\beta,m} g_{n,m} = (Q^{-1} \cdot G^T)^T \tag{4.34}$$

$$S = s_{\alpha,\beta} = \frac{1}{\lambda_\alpha(A) + \lambda_\beta(B)} \sum_{n=1}^{N} p'_{\alpha,n} r_{n,\beta} = \Lambda \odot (P^{-1} \cdot R) \tag{4.35}$$

$$T = t_{\alpha,j} = \sum_{\beta=1}^{M} q_{j,\beta} s_{\alpha,\beta} = (Q \cdot S^T)^T \tag{4.36}$$

$$U = u_{i,j} = \sum_{\alpha=1}^{N} p_{i,\alpha} t_{\alpha,j} = P \cdot T \tag{4.37}$$

where U is the exact solution to the PDE, similar to the approximation we would obtain on iterative methods such as the already discussed Jacobi iterative solver. The resolution of this method is trivially implemented as it is shown in a generic solver on Listing 4.10.

```
import jax.numpy as jnp
from jax import partial, jit
# N x N broadcasted inverse eigenvalues matrix
EIVAL = 1 / (eivalA_broadcasted + eivalB_broadcasted)

@partial(jit, donate_argnums=0)
def solve(G, Q, P, EIVAL, IQ, IP):
    R = jnp.dot(IQ, G.T)
    S = EIVAL * jnp.dot(IP, R.T)
    T = jnp.dot(Q, S.T)
    return jnp.dot(P, T.T)
```

Listing 4.10: Solver of PDEs using tensor products.

With the method described, we will focus on our heat dissipation benchmark, as it is the only one that deals with convergence instead of time evolution. Since it is a steady state problem that follows a Laplacian equation, the method can be simplified as follows

For starters, the PDE to solve is of the Laplacian form

$$D_{xx}u_{i,j} + D_{yy}u_{i,j} = 0 \tag{4.38}$$

which, due to the independence of each D operator in regards to the other axis, it is readily rewritten as

$$(I \otimes A + A \otimes I)u = g \tag{4.39}$$

where g contains the initial state with the Dirichlet Boundary conditions. Therefore, we are in the case where $B = A$, hence $Q = P$ and $\lambda(A) = \lambda(B)$.

Since the second order differential matrix with Dirichlet conditions is a fairly common scenario, we can obtain the eigenvectors from the literature [38].

$$P = Q = \sqrt{\frac{2}{N+1}} \sin\left(\frac{i \cdot j \cdot \pi}{N+1}\right) \tag{4.40}$$

for i and j with values $i, j = 1, ..., N$. A visualization of the pattern of P can be seen on Figure 4.15. From the equation, as well as from the figures, we see that despite being normalized, the eigenvectors of A are not sparse. Hence, we will be working with dense matrices despite the sparse initial condition. The eigenvalues are found by using (4.28).

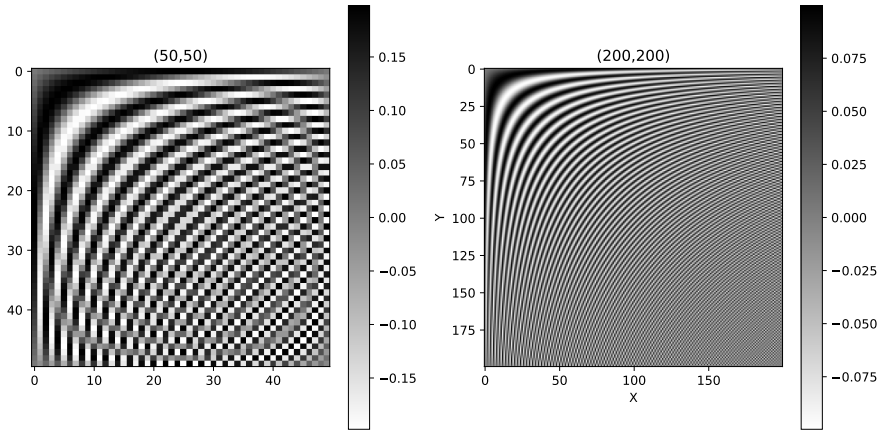


Figure 4.15.: Visualization of the values and pattern of the Matrix P for different sizes for Dirichlet Boundary conditions.

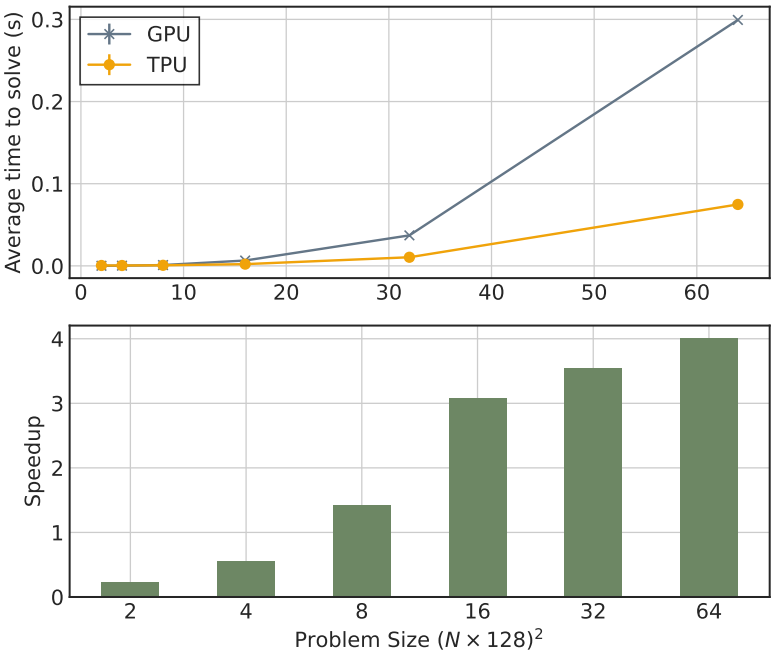


Figure 4.16.: Performance comparison of the tensor product solver using a GPU and a TPU core. Both devices performing the same implementation of the tensor solver.

4.6.2 Results

A simple timing comparing the solver on both a TPU core and a GPU yields the results shown at Figure 4.16. Despite the GPU performing better than a single TPU core for smaller sizes, at $N^2 = (8 \times 128)^2$ the TPU starts to outperform the GPU. It should be clearly stated that the comparison is between a single core and the whole GPU, which the performance of TPUs doing matrix multiplication all that impressive.

These results serve as an example to confirm what was already expected from the matrix multiplication method results. A TPU will vastly outperform a GPU in terms of scalability. The larger the matrix (or grid), the better will the TPU perform in comparison to the GPU. Nevertheless, the memory footprint of this method is something to take into consideration when looking into extending it to large scale problems, as it would seem that it uses at least $8\times$ the amount of memory needed to store the surface grid, as seen in Figure 4.17.

Despite not being a fair comparison to do, Table 4.3 shows the times in takes for the solver to converge.

Because the tensor products returns the exact solution as opposed to the iterative method, the idea is that most of the times, if there is a method that uses matrix operations, it will likely outperform those that need transformations in order to use the MXU.

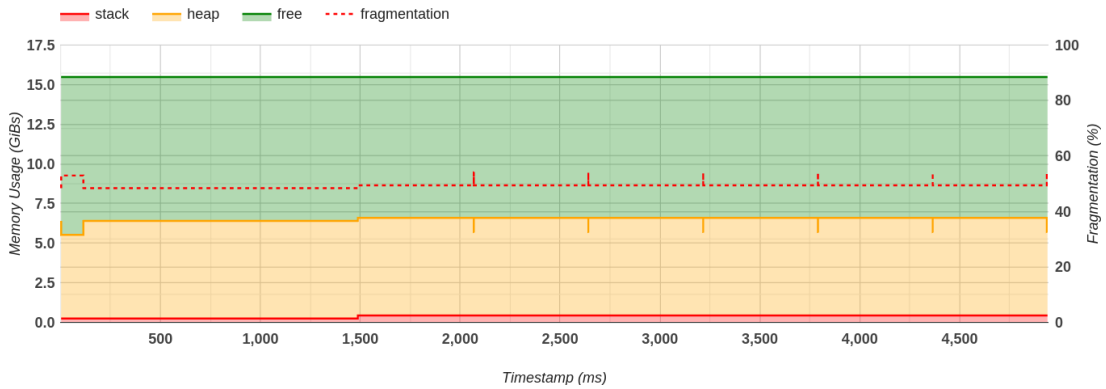


Figure 4.17.: Memory profiler of the Direct Solution using Tensor Products for a grid of size $(120 \times 128)^2$. The profiler display 7 consecutive solvers which explains the red spikes in the fragmentation. The original grid would have uses 0.9GiB, instead, there is a report peak memory usage of 12.75GiB in the profiler, despite not being shown in the graph. Hence, memory management of this method can not be considered optimal

	GPU Vector	TPU Product
Convergence Time	$1023s$	$0.28s$

Table 4.3.: Comparison between the time it takes the vector iterative method to obtain a close result to the direct solution by tensor product on a TPU core. The size of the problem is of $N^2 = (100 \times 128)^2$

As described in the original paper, the model should not present any instability for larger number of elements in the grid. During our testing and studying of the implementation, no instability for larger number of grid elements was found either. Despite its performance, the model does need a rather idealized scenario so it does not have the versatility that the FDM has.

4.7 Discussion

After finding a method capable of providing a reasonable speedup with respect to conventional methods, we proceed to summarize and evaluate all the aspects of it.

4.7.1 Performance and Efficiency

To summarize, a comparison of the performance of all the methods is done as shown in Table 4.4. The results show how effective the tiles method is compare to the other two for all three of the simulations. From the three benchmarks proposed, the tiles approach continues to yield the best results on the Shallow Water solver when using every core of the TPU.

Another remarkable observation to make is that the difference in Memory usage does not seem to be dependent on the method used. This does goes against our intuition. However, as stated in Chapter 3, once a function is jitted, XLA is the one in charge of memory management. And judging by these results, it would appear to be that XLA is highly dependent on the order function are written. Hence, some more consideration should be added to prioritize memory utilization instead of readability. Ideally, XLA should be able to optimize even inefficient written code as long as all the functions used are traceable. However, it does seem to perform optimization for some of the simulations. Nonetheless, I believe that XLA may not be as powerful as it can be yet and the way of programming needs to be careful planned to not hog memory.

In order to compute the energy consumption on each case, we use the TDP value, which indicates the maximum Power ($W = J/s$) can expect on heavy workloads, and we multiplying by the time it took the simulation to run. Hence, we end up with a rough metric that can tell us how much energy does each method consumes.

Method	Execution Time (s)	Performance $(iter/s)$	Energy Consm. (kJ)	Mem. usage (GiB)	Cloud Cost ($)
Heat Diffusion					
Vectors	51.72 ± 0.02	19.3	93.096	24	0.1264
Matrix	319.4 ± 0.6	3.13	574.989	18.08	0.7808
Tiles	43.77 ± 0.01	22.84	78.801	16.96	0.1070
Wave Equation					
Vectors	60.50 ± 0.03	16.53	108.903	34.16	0.1479
Matrix	311.27 ± 0.1	3.12	560.293	36	0.7609
Tiles	40.99 ± 0.02	24.39	73.790	32	0.1002
Shallow Water					
Vectors	98.46 ± 0.06	10.16	177.225	40.2	0.2406
Matrix	623.21 ± 0.2	1.60	1121.773	38	1.523
Tiles	50.93 ± 0.01	19.36	91.677	32	0.1245

Table 4.4.: Comparison results of the three methods on the three simulations on a $(131072 \times 16384) = 2.19 \cdot 10^9$ grid all using a single TPU unit (8 cores). In the heat diffusion simulation, we consider 1000 iterations instead of waiting for convergence. Likewise, we use 1000 iteration steps on the rest of time forward benchmarks.

With the tile method being the clear winner, when considering the correct tile size, it is compared to the most performant method on GPU. Because we are comparing 8 TPU Cores against a single GPU, the performance metrics are also normalized per power usage and cloud pricing. Since our GPU is not listed on Google Cloud, there is no price tag for it. Based on its half floating point performance, a similar price to the V100 has been assumed despite being an overestimation, since the latter has yield better performance across the board.

	TPU			GPU		
Grid Size/Core N^2	GFLOPS	GFLOPS/$	GFLOPs/W	GFLOPs	GFLOPS/$	GFLOPs/W
	Heat Diffusion					
$(16 \times 128)^2$	360.1	40.91	0.20	122.2	47.94	0.48
$(32 \times 128)^2$	390.4	44.35	0.21	131.6	51.62	0.52
$(64 \times 128)^2$	390.8	44.40	0.22	134.4	52.72	0.54
$(128 \times 128)^2$	392.3	44.58	0.22	135.2	53.04	0.54
	Wave Equation					
$(16 \times 128)^2$	519.1	58.99	0.29	229.1	89.84	0.92
$(32 \times 128)^2$	539.8	61.35	0.30	251.2	98.50	1.00
$(64 \times 128)^2$	526.1	59.78	0.29	258.2	101.26	1.03
$(128 \times 128)^2$	523.9	59.53	0.29	250.7	98.31	1.00
	Shallow Water					
$(16 \times 128)^2$	369.7	42.02	0.21	128.3	50.32	0.51
$(32 \times 128)^2$	383.1	43.53	0.21	135.5	53.15	0.54
$(64 \times 128)^2$	381.5	43.36	0.21	137.6	53.97	0.55
$(128 \times 128)^2$	379.5	43.12	0.21	138.2	54.18	0.55

Table 4.5.: Comparison between the best performant method on TPU against the most performance GPU method.

Based on the final results, it is clear that computing finite difference will not be more efficient nor economic to run on TPU instead of GPU for any size or any simulation. However, we did not test the scaling performances on multiple GPUs and those could yield interesting results if not they were not capable of scaling as well as TPUs.

4.7.2 Accuracy

One important aspect about the use of TPUs is their limitation on the data precision, as the standard precision for physical simulations is usually double floating points (`float64`). Because JAX allows to modify the default data format from 32 to 64 on CPUs and GPUs, we will use a simulation on GPU using the full 64 bits to see how the different data formats make the simulation diverge.

Despite not seeing any alteration on the simulations by performing a visual inspections on the animated outputs, we measure the Mean Square Error of all the grid points on different stages of the simulation to see if the error of the values remain fairly similar over time or instead, they present a divergent behavior that may result in wrong predictions for long running simulations.

By looking at Figure 4.18, we observes some divergences when using the TPU, and specially the MXU, on its default configuration. Despite not being visible, on a 10,000 iterations simulation, the difference between the double floating points simulation and the brain

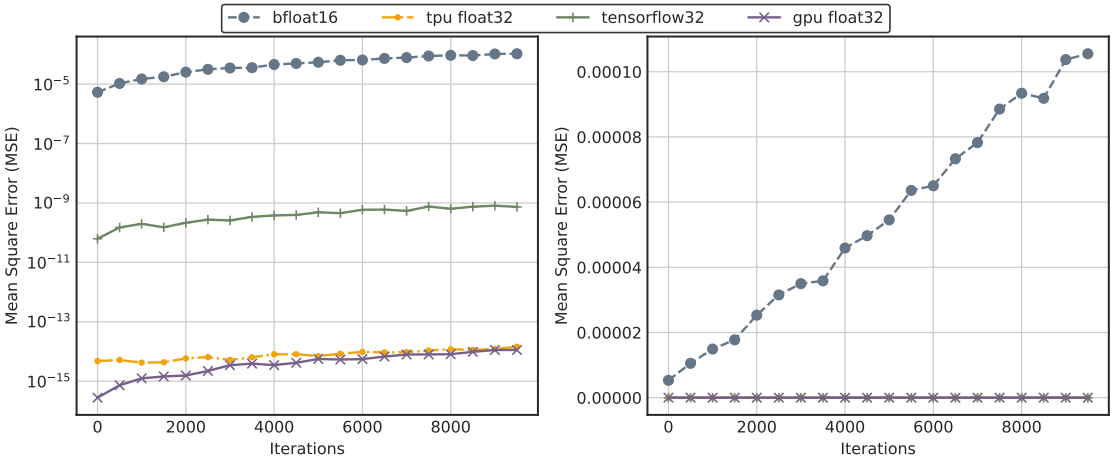


Figure 4.18.: Accuracy comparison between the height of the waves of the Wave Equation benchmark as the iterations on the simulation go on. The MSE are with the values obtained from a GPU run using double floating points data type.

floating paints have increased linearly. Regardless, assuming the worst case scenario where the MXU is used on all the iterations, it does still yield good results, with only $\epsilon \approx 10^{-5}$ as the mean square error. For simulations that use large values, this divergence would have less noticeable and the limitation of having a lower range of values as it would be the case for `float16` would have been inferior. Nevertheless, based on this evidence, we argue that the impacts of using lower precision would be noticeable in long simulations, and that using lower precision would only be worth it for simulations with number of on the hundred of thousands order of magnitude if precision was necessary, which would still include plenty of common physics calculations. Hence, the speedup provided from using lower precision exceeds the possible divergence on the results respect the real values, but, should be avoided if a high fidelity is required for very long simulations, and using more passes on the MXU yields worst performance.

4.7.3 Scalability

Scalability is a big selling point on TPUs. Due to the independence of batch operations, ML workflows can easily distribute the workload into hundreds of different devices and still not deal with communication issues.

However, because we need to exchange boundaries in every time step, it begs the question if using multiple TPUs will yield the same linear scalability observed on a single unit. Thus, Figure 4.19 shows the strong scaling performance of TPUs on using the tiles method. The observed scalability on using 2 TPUs Unit is linear, with a very small deviation when using 16 cores, which is to expect considering that in order to be able to perform the tiles, there's need to be at least $(8 \times 128)^2$ grid points. And at that scale, the computation happens so fast that the TPUs end up having to wait on the exchange.

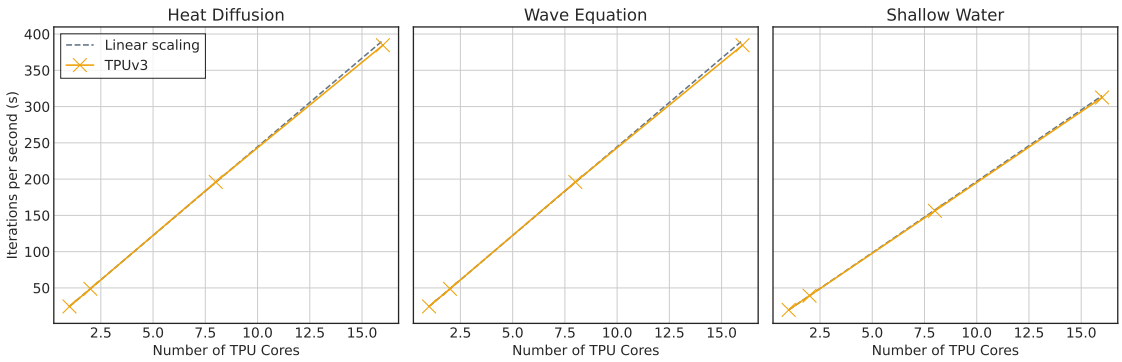


Figure 4.19.: Strong scaling of the Tiles algorithm of a $(128 \times 128)^2$ grid. By using tiles of $m = 1024^2$ as they proved the best performance, we can only scale up to 2 TPUs.

Looking at the weak scaling performance on Table 4.6, we see that the time to perform 100 iterations is the same regardless of the number of cores, which confirms that TPUs scalability is one of the strong points to use. Additionally, it proves that our implementation has no issues scaling linearly.

# TPU Cores	Grid Size	Heat Diffusion (s)	Wave Equation(s)	Shallow water(s)
1	$[128, 128] \times 128$	4.37	4.09	5.08
2	$[256, 128] \times 128$	4.38	4.09	5.09
8	$[1024, 128] \times 128$	4.37	4.10	5.09
10	$[1280, 128] \times 128$	4.39	4.10	5.10
12	$[1536, 128] \times 128$	4.38	4.10	5.09
16	$[2048, 128] \times 128$	4.37	4.09	5.10
18	$[2304, 128] \times 128$	4.40	4.10	5.12
20	$[2560, 128] \times 128$	4.39	4.11	5.10
24	$[3072, 128] \times 128$	4.39	4.12	5.11

Table 4.6.: Weal scaling performance of the new implementation. The table shows the times each run has taken to perform 100 iterations.

5 Complex Systems: Ising Model

In this section we take a look at the only previous study, to our knowledge, based on the viability of Google TPUs in HPC (Yang *et al.* [39]), where it is displayed the performance of TPUs running a large scale Ising Model. As the authors motivations resemble our own, we analyze their approach and adapt it to this thesis findings. Mainly, the aspects and issues that we would like to touch upon on this section are:

- **TensorFlow:** They published their paper before TensorFlow 2.0 was released, hence their implementation is written in TensorFlow 1. As Google does not recommend using Tensorflow 1 anymore, it is likely that their expectations on making TensorFlow 1 a numerical methods framework will not be met. Even though porting from TensorFlow 1 to its next version should not require extensive work, it is our opinion that TensorFlow graph model alienates physicists whom are accustomed to array syntax. Regardless, JAX has access to the same XLA operations in a more friendly approach and I believe that showing their algorithm implemented on JAX may help people interested in porting or writing stochastic simulations on TPUs.

- **Tile size:** On their implementation, they also realize that in order avoid sparsity, the grid needs to be split into smaller grids. However, they reason that because 128 is the smallest size possible on the physical systolic array, it is an effective subgrid size. That assumption contradicts the results seen on Chapter 4 for the proposed tiled approach, where $m = 128$ tiles are found to be up to twice less effective than a regular vectorized approach, regardless of not using the MXU at all. However, larger tiles do report higher performance, thus, we analyze how tile size changes their implementation performance.

- **Wrong TDP value:** Because their paper came before most of the information regarding TPU was available and despite the authors working at Google, they estimated the TDP of a v3 chip to be $200W$, less than half the real reported value of $450W$ [40]. Hence, a correction to their estimated power consumption values is done in order to compare it to our results.

- **XLA advances:** Lastly, we use their paper as a reference to quantify if there has been an improvement on the XLA compilers when dealing with TPUs. From personal experience during these months working on this project, it has seemed that XLA, and JAX for that matter, have improved their performance drastically. Therefore, we will use this to formally quantify that feeling.

5.1 The Ising Model

Before we implement their algorithm, we briefly introduce the physics behind it. By doing so, we expect to show that the reasoning of its results can be extendable to other stochastic simulations that may resemble the Ising Model.

The Ising model is an example of a lattice model where one variable is located at each site of a regular grid and the state of the variables is determined by a function. In the Ising model case, that function is the Ising Hamiltonian.

$$\mathcal{H} = -J \sum_{\langle ij \rangle} s_i s_j + h \sum_i s_i \tag{5.1}$$

Where $< ij >$ denotes that a sum is to be carried out over all nearest-neighbor pairs of sites i and j, and J is the coupling between these neighboring sites. It is assumed that long range interactions are neglected due to nearest neighbor interactions being dominant in the Hamiltonian. The quantity h represents an external magnetic field which interacts with the magnetic moment of each spin s_i and it is assumed null for this simulation. Such models have been successful in the description of critical phenomena, magnetism, models for high-temperature superconductivity and phase diagrams, disordered and non-equilibrium systems, etc.

Despite the 1 dimensional exact solution being relatively easy to reproduce which proves that there is no phase transition at 1D [41], finding the solution on 2 dimensions is not a trivial task as proved by [42]. For the 3D Ising Model no analytical solution has been found yet. Therefore, numerical simulations provide an alternative and in some cases, the only way, to study these systems behavior, thus its relevance in computational physics.

The Metropolis Algorithm

The Ising model is implemented using a Monte Carlo approach which would require picking random number of states, flipping them and performing their measurements, weighting them by their Boltzmann factor. However, this would require a large number

of measurements. The Metropolis algorithm provides a method on which, based on their Boltzmann factors, the flipping of a state is chosen.

Going into more detail into the implementation of the classical metropolis alghoritm, we have the following steps:

1. Generate a 2D lattice of size $N \times N$ where every cell is a lattice site with spin value $s_i = \pm 1$. A matrix with all initial values set to 1 is used.

2. For the model to work we need a large amount of sites $N \to \infty$. Therefore, we assume periodic boundaries.

3. A single lattice site is selected and flipped. If the change in energy ΔE is negative $\Delta E < 0$, the flip is accepted and another random point is chosen. Otherwise, if the change in energy is positive $\Delta E > 0$, the flip is accepted with probability $\exp(-\Delta E / \kappa_B T)$.

From the description above, is it clear that the Metropolis Algorithm is a sequential algorithm that can not utilize the potential of parallel accelerators. Nonetheless, an implementation that make use of the nearest neighbors interaction to perform the flips in a parallel manner exists, as it is the case of the checkerboard algorithm. [25].

5.2 TPU Implementation

As we have previously stated, their implementation resembles the tiled matrix approach discussed on this thesis. However, due to the nature of their simulation, there are a fair share of differences that we need to mention.

They based their implementation on the checkerboard algorithm just mentioned, which allows for the flipping of all non-interactive spins in the same step in parallel. The checkerboard algorithm assigns each spin a color, white or black similar to the checkerboard like pattern seen in Figure 5.1.

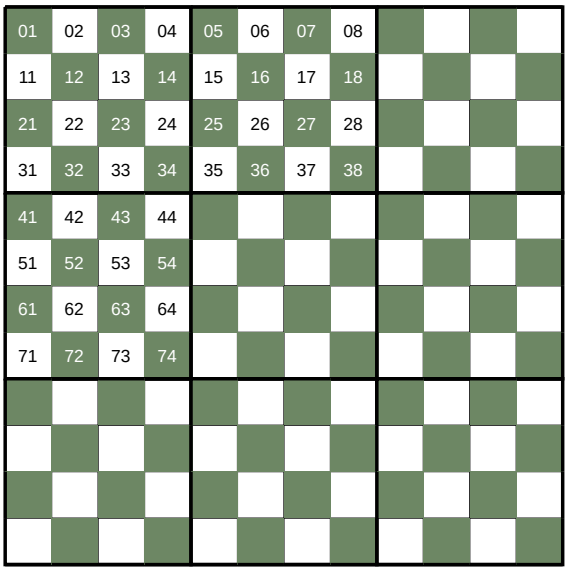


Figure 5.1.: Coloring distribution of the spin in the lattice. Green has used for aesthetics reasons and should be consider black on when referred to it. The number display on the initial cells are the subindexes i, j of s_{ij}

In doing so, they are able to flip all the spins of the same color using the other color value to evaluate the energy difference of each flipping. Since the computer intensive part is the sum of neighbors, as in our iterative solver, they apply the same matrix K.

$$K = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ 1 & 0 & 1 & \dots & 0 & 0 & 0 \\ & \vdots & & \ddots & & \vdots & \\ 0 & 0 & 0 & \dots & 1 & 0 & 1 \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 \end{pmatrix}_{128\times128} \qquad M = \begin{pmatrix} 1 & 0 & 1 & \dots & 0 & 1 & 0 \\ 0 & 1 & 0 & \dots & 1 & 0 & 1 \\ & \vdots & & \ddots & & \vdots & \\ 1 & 0 & 1 & \dots & 1 & 0 & 1 \\ 0 & 1 & 0 & \dots & 0 & 1 & 0 \end{pmatrix}_{128\times128} \tag{5.2}$$

To do it, a boolean mask M is applied so that the flip values only affect the spins of the same color. Another difference between the implementations is that they assume and infinite periodic system, meaning that the 2D lattice is in reality a torus. Therefore, once the matrix multiplication has been performed the boundary values are corrected to include the values from the neighboring subgrids. To show how simple this implementation can be, the code is shown in Listing 5.1.

This approach, however, presents certain redundancies that are observed on the original paper as well. In order to avoid them when computing on the whole lattice, they propose a new algorithm. According to their experiments, this newer approach yields a $3\times$ speedup over the previous method. Thus, we implemented it as well in order to test if this relation still exists.

```python
def ising_half_step(key, x, mask, K, beta):
    key1, key2 = jax.random.split(key, 2)
    prob = jax.random.uniform(key1, mask.shape)

    nn = jnp.matmul(x, K) + jnp.matmul(K, x)
    nn = nn.at[:,:,0,:].add(jnp.roll(x[:,:,-1,:], 1, 0))
    nn = nn.at[:,:,-1,:].add(jnp.roll(x[:,:,0,:], -1, 0))
    nn = nn.at[:,:,:,0].add(jnp.roll(x[:,:,:,-1], 1, 1))
    nn = nn.at[:,:,:,-1].add(jnp.roll(x[:,:,:,0], -1, 1))

    acceptance_ratio = jnp.exp(-2 * beta * x * nn)
    flips = (prob < acceptance_ratio) * mask
    x = x - 2 * x * flips
    return key2, x, 1 - mask
```

Listing 5.1: Naive implementation of the Ising Model with the checkerboard algorithm.

The new approach is based on removing the multiplication of the mask, as it means having to compute the sum over nearest neighbors as well as the acceptance ratio for half the grid that does not need it. Instead, each subgrid is divided in more subgrids with the spins of each rearranged to be in sub-blocks of the same color as seen in Figure 5.2.

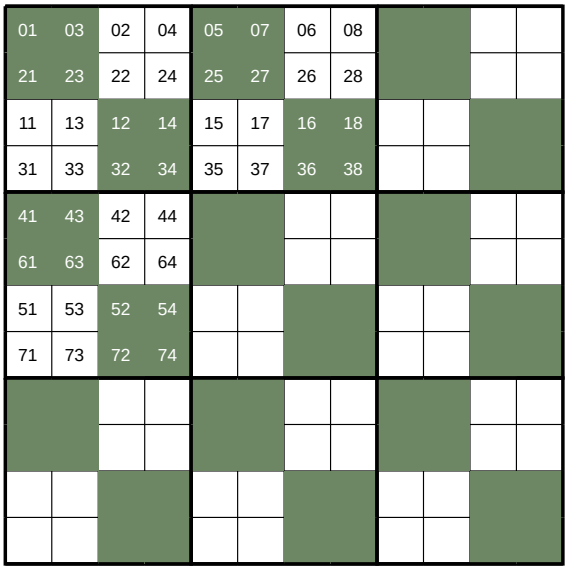


Figure 5.2.: Optimized rearrangement of the spins so that all the spins with the same color inside the sub-lattices are together.

Because of the new distribution, a new kernel, similar to the one of the forward finite difference scheme, is used.

$$\hat{K} = \begin{pmatrix} 1 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & 1 & \dots & 0 & 0 & 0 \\ & \vdots & & \ddots & & \vdots & \\ 0 & 0 & 0 & \dots & 0 & 1 & 1 \\ 0 & 0 & 0 & \dots & 0 & 0 & 1 \end{pmatrix}_{128\times128} \tag{5.3}$$

```python
def ising_optim(key, s,  black, K, beta):
    key1, key2 = jax.random.split(key, 2)
    probs0 = jax.random.uniform(key1, s[0].shape)
    probs1 = jax.random.uniform(key2, s[0].shape)

    idx1 = jnp.where(black, 0, 1)
    idx2 = jnp.where(black, 3, 2)

    nn0, nn1 = lax.cond(black,  nn_black, nn_white, (s,K))

    flips0 = probs0 < jnp.exp(-2*beta*nn0*s[idx1])
    flips1 = probs1 < jnp.exp(-2*beta*nn1*s[idx2])

    s = ops.index_add(s, ops.index[idx1], -2*flips0*s[idx1])
    s = ops.index_add(s, ops.index[idx2], -2*flips1*s[idx2])

    return key, s, black^True
```

Listing 5.2: Optim implementation of the Ising Model with the checkerboard algorithm.

```python
def nn_black(s):
    (s00, s01, s10, s11), K = s
    nn0 = jnp.matmul(s01, K) + jnp.matmul(K.T, s10)
    nn0.at[:,:,0,:].add(jnp.roll(s10[:,:,-1,:], 1, 0))
    nn0.at[:,:,:,0].add(jnp.roll(s01[:,:,:,-1], 1, 1))

    nn1 = jnp.matmul(K, s01) + jnp.matmul(s10, K.T)
    nn1.at[:,:,-1,:].add(jnp.roll(s10[:,:,0,:], -1, 0))
    nn1.at[:,:,:,-1].add(jnp.roll(s10[:,:,:,0], -1, 1))

    return nn0, nn1
```

Listing 5.3: Nearest Neigbours sum for the black color

Therefore, we define the black spins as s_{00} and s_{11} whereas the white spins are s_{01} and s_{10}. The near neighbors sum is then computed as

$$
\begin{aligned}
nn_{00} &= s_{01} \cdot \hat{K} + \hat{K}^T \cdot s_{10} \\
nn_{11} &= \hat{K} \cdot s_0 1 + s_{10} \cdot \hat{K}^T \\
nn_{01} &= s_{00} \cdot \hat{K^T} + \hat{K}^T \cdot s_{11} \\
nn_{00} &= s_{00} \cdot \hat{K} + \hat{K} \cdot s_{11}
\end{aligned}
$$

with the boundaries corrected using the same approach as before. The implementation, in spite of the added complexity, can still be written in a few lines of code as seen in Listing 5.2

and the near neighbor calculations as seen on `nn_black` on Listing 5.3.

5.3 Results

We begin by performing both algorithms to check whether they get such a different performance.

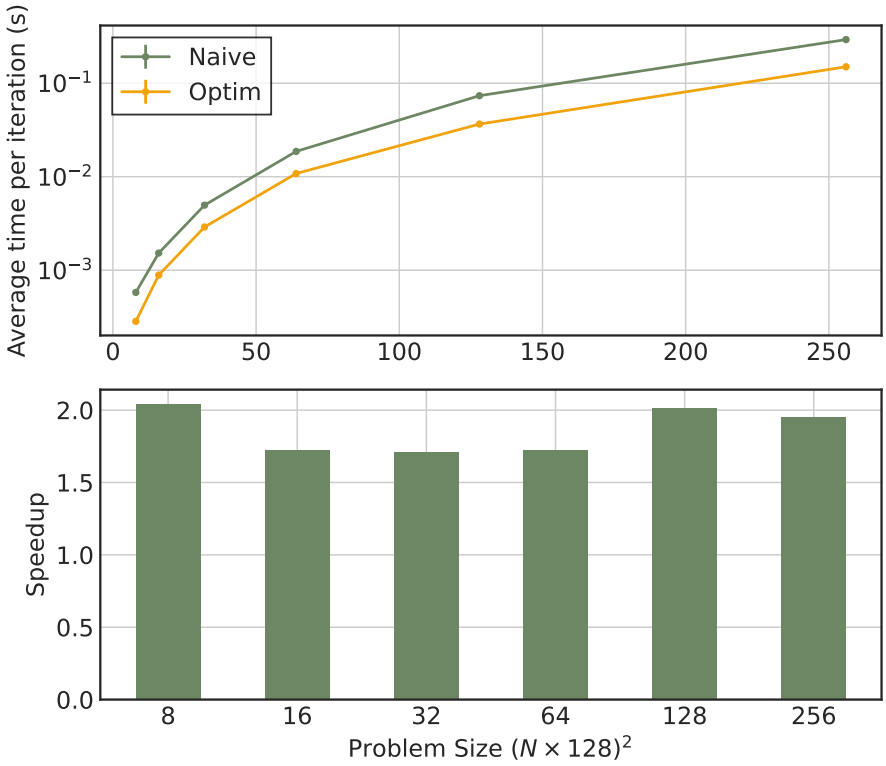


Figure 5.3.: Timing comparison between both algorithms to run the ising model.

The speedup observed is of $2\times$ in contrast to their reported $3\times$. However, the quality of the implementation can also be a factor to consider. Regardless of quality, it is clear that their optimized approach does yield better results than the naive approach. Additionally, it does seem that the speedup is constant regardless of the problem size, which aligns with the expectations as you are reducing to half the computations.

In order to compare our implementation with the ones from the previous study, we perform the same simulations with the same lattice sizes and compare their throughput as well as their efficiency. Their Corrected TDP has been multiplied by the correction faction 2.25 so that we consider a TDP of $250W$ instead of their assumed $100W$.

As seen on Table 5.1, the values for our implementation are considerably lower than the ones reported on the paper. The reason for this discrepancy is still unknown. Some alterations to the implementation were done without any successful result. However, since the profiling tool has just been released, we can analyze an iteration to see if there is an specific reason for it to perform that poorly.

	Reference		Our Implementation		
lattice size n^2	Original (flips/ns)	Corrected TDP (nJ/flip)	m=128 (flips/ns)	m=1280 (flips/ns)	Efficiency (nJ/flip)
$(20 \times 128)^2$	8.1920	27.4658	3.615	2.1786	60.4757
$(40 \times 128)^2$	9.36323	24.03248	3.668	2.6737	56.7058
$(80 \times 128)^2$	12.3362	18.23895	4.1101	3.2852	54.7425
$(160 \times 128)^2$	12.8266	17.50417	4.1600	3.6654	54.0872
$(320 \times 128)^2$	12.9056	17.43435	-	-	-
$(640 \times 128)^2$	12.8783	17.47125	-	-	-
NVIDIA Tesla V100	11.3704	21.9869			

Table 5.1.: Performance comparison between the original paper figures at the ones obtained on our implementation.

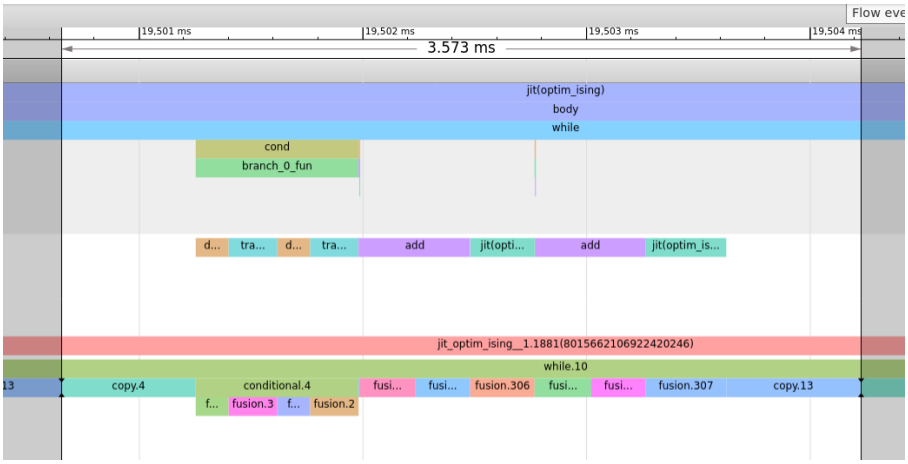


Figure 5.4.: Screenshot of the trace viewer of the optimized Ising method

Figure 5.4 shows the profiler view of a single iteration from the optimized algorithm. The initial observation from the trace viewer is that the dot product does not represent the largest computation on a step. Instead, the functions related to the element wise operations and random generation are the ones taking the most amount step time, alongside copying the array. These results had not been able to be observed when trying to improve the performance of the algorithm and gives us an idea on how to adapt it to perform better. Aside of helpful insight on the operations each time step, it also shows a time per iteration lower than the reported timing of our solution. Different timing techniques may explain the performance difference between our implementation and the reported ones in the paper, as in our case we are considering a timing on the whole simulation to later average it with the number of steps whereas the paper's authors only time the iteration. For instance, using the provided time in the profiler of $t = 3.573ms$ per iteration on a $n^2 = (40 \times 128)^2$ lattice, 7.3368 flips/ns is obtained, which yields twice the previous reported value and a similar, although lower, performance to the reported one on the original paper.

5.4 Discussion

After comparing our implementation with the one provided on the paper, we see that the JAX code is not able to match the performance of the TensorFlow graph. It is hard to point whether this is due to TensorFlow being a more mature language and having better XLA optimizations than the current state of JAX, or if the reason for the worst performance is due to the implementation. Regardless, the readability of the JAX code is being far better than the one of the TensorFlow approach.

From the results of profiling the function, we observe that our MXU utilization corresponds to the 20% of the step time, in contrast to their reported 60%. Additionally, there are two copying of arrays that use 32% of the time and that we were not aware up until seeing the profiler. However, the most informative part of the results corresponds to the time it takes for the pseudo-random generator to generate the uniform probabilities. Whereas in the original paper this tasks, performed using TensorFlow `tf.uniform` function, takes 12% of the time, on our implementation using JAX's own pseudo-random number generators it takes up to 35% of the running time.

These values need more studying but have served us to realize how informative and useful a profiler really is.

6 Porting of existing simulations

Porting Veros, an ocean simulation currently being developed by Team Ocean at the Niels Bohr Institute, to run efficiently on Google Cloud TPUs was originally the main motivation behind this thesis. Therefore, I have spent the large part of it trying to make Veros run efficiently on them. Due to the challenges usually found on new technologies, such as the lack of tools, documentation and unstable implementations, in addition to the complexity of Veros itself, I could not manage to achieve any significant result. Instead, we decided that it would be better to take a step back and perform a study on the solution of PDEs in a simpler scenario where there was more control over the variables involved. In doing so, providing some guidance for future works before attempting to port an established simulation such as Veros without any feedback on whether the changes made sense or not.

In this section, the results obtained from the proposed approach are used to evaluate the porting of Veros to TPUs. Its main features and core computations are assessed in order to reach a conclusion on whether a porting is a good idea. Moreover, I will include my suggestions on how I would approach some of Veros calculations using the experience gained after doing this thesis.

Based on my experience, I will discuss some guidelines to consider when thinking about supporting simulations to use Google Cloud TPU clusters as well as to express where I believe the proposed algorithm for solving PDEs may be more appropriately used.

6.1 Veros: The versatile ocean simulation

6.1.1 Introduction

Veros is an ocean simulation completely written in pure Python developed at the Niels Bohr Institute by the Team Ocean research group[13]. Veros is a porting of the highly used PyOM2 library with its FORTRAN backend replaced by pure Python functions. Henceforth, Veros tries to encompass the efforts of creating a HPC ecosystem in a high level language

such as Python, by providing a high performant general circulation model that is extremely extensible, accessible, massively parallelizable and generally, easy to use.

Veros implementation is not a direct translation from FORTRAN to Python as that would yield very slow performances and it would not be a viable solution. Instead, it is based on the observations of vectorized approaches discussed on Chapter 4, where it is showed that vectorized operations could reach similar performances to low level sequential code. Similarly to the PDE benchmarks, Veros numerical solutions are calculated using finite differences. Therefore, the assessment of Veros should resemble the approaches studied on Chapter 4 and the resulting study will be based on whether it is feasible to apply the proposed method to existing Veros functions.

Veros is intended to work on as many devices as possible and it is able to do so by making use of the flexibility and abstraction that Python provides. Consequently, one consideration to have is whether the alterations required to run them on the TPUs are not disruptive enough for the code that will run on the other devices.

Lastly, the main motivation behind Veros aligns perfectly with the one for surge of novel AI chips on which both target high speedups by making use of tensor operations and abstractions instead of relying on, prone to errors, index based approaches of low level languages. Therefore, it is to expect that a device called Tensor Processing Unit, may be the ideal candidate for a tensor based simulation such as Veros.

6.1.2 Assessment for TPUs

Luckily for this assessment, some benchmarks of Veros had already been implemented in order to compare different backends, which lead to the decision of adding support for JAX as it outperformed most of the other libraries on CPU as well as on GPU[43]. In addition to its high performance and numpy-like syntax, JAX also provided the support for multiple architectures as stated in Chapter 3, which made it great fit with Veros motivations of being hardware agnostic. In fact, having Veros support JAX was the main catalyst behind choosing Cloud TPUs to study its efficiency on running numerical simulations. The provided benchmarks encapsulate the most common operations found on Veros but without the complexity of a full fledged ocean model. Therefore, they are the ones that I used during my attempts to improve Veros performance.

Equation of State

The first benchmarks in the suite is the calculation of the Equation Of State with the full 48-terms based on Gibbs sea water EOS[44].

```
t179 = ct * (v28 + t177)
t185 = v35 * ct
t187 = ct * (v34 + t185)
t189 = ct * (v33 + t187)
t199 = t13 * t20
t217 = 2.0 * t117 * t199 - t110 * t92
t234 = v21 + ct * (v22 + t169) +
                sa * (v26 + ct * (v27 + t179) + v36 *
                sa + t82 * (v31 + ct * (v32 + t189)))
                + t217 * t20
```

Listing 6.1: Lines from the Equation of State benchmarks

The benchmark itself consists on thousands of array element wise floating point operations and a snippet of some of those is shown at Listing 6.1 where the variables `v#` and `t#` are 3 or 4 dimensional arrays.

Since there is no feasible way to perform these operations making use of the MXU, e.g. transforming it into `dot` operations, our only hope was that the XLA compiler on TPU would fuse them and *maybe* use the MXU of the resulting merged operations. However, in the metrics on the Google Cloud Console, running this benchmark on the initial TPUv2 yield a 0% usage of the MXU.

At some point, the profiler managed to store tracings of the TPU calculations and I could visualize them on Tensorboard as briefly discussed on Chapter 3. Regardless of this, the resulting profiling can be seen on Figure 6.1 where it indeed confirms us that XLA merged most of the operations into a single one, indicated as `region_2`.

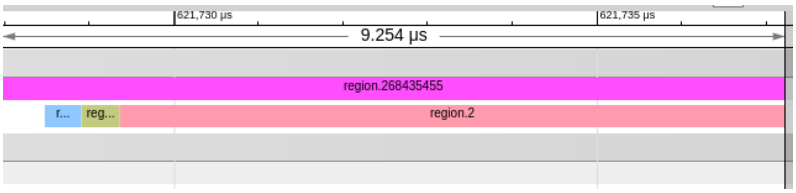


Figure 6.1.: Profiling screenshot of The Equation of State benchmark using TPUv2 on the Tensorboard profiler.

In conclusion, there was no too much to do on this case. Element-wise array operations are going to be handled by the VPU, which as seen on the results of 4.8 it does perform reasonable fine. Be that as it may, a GPU is the best hardware for these operations and in order to justify the use of the VPU for these calculations is if a most computational intensive task makes use of the MXU instead and the speedup of it overcomes the lower performance of the this part of the simulation.

```
from jax.ops import index_update, index
import jax.numpy as jnp
...
dTdy = index_update(
    dTdy, index[:, :-1, :], maskV[:, :-1, :] * \
    (temp[:, 1:, :, tau] - temp[:, :-1, :, tau]) \
    / dyu[jnp.newaxis, :-1, jnp.newaxis]
)
dSdy = index_update(dSdy, index[:, :-1, :], maskV[:, :-1, :] * \
    (salt[:, 1:, :, tau] - salt[:, :-1, :, tau]) \
    / dyu[jnp.newaxis, :-1, jnp.newaxis]
)
...
```

Listing 6.2: Calculation of the gradients at the north face of T cells.

Isoneutral Mixing

Assuming that the calculation of the Equation of State can not be optimized, the focus went to the operations in the Isoneutral mixing benchmark. This benchmark simulates the mixing in the ocean which takes place along surfaces of constant neutral density. Therefore, Veros needs to find the locations of these neutral density surface at every time step. To do so, it uses the finite difference scheme to compute derivatives and since it is the most costly operation in an ocean model, it should be clear why the study on PDE solvers was so important.

One example of the operations found on this benchmark can be seen on Listing 6.2 Despite the initial confusion of slices and indexes, this is a prime example on how to properly use vector operations to compute finite differences. The immutability of JAX's `DeviceArrays` does not help on the readability factor. In summary, there is a derivation on the y-axis of two 4D arrays (`temp`,`salt`) with a 4th axis representing the time layer fixed on τ and a not uniform spacing between elements, as seen in `dyu` and its broadcasting. Additionally, a boolean mask is applied so that the derivative is only computed on parts of the grid.

Regardless, the only part here that seems that could be transform into MXU friendly code is the difference between consecutive values similarly to the approach on the shallow water benchmark.

The implementation of these differences are rather trickier than in the 2 dimensional case, as `matmul` will operate on the last dimensions of the array. That constrain has a an easy fix on the 2D case, where you can just change the order of the operands when multiplying. However, that is not the case when dealing with higher dimensions $D > 2D$ since in order to use it we would need to apply a transformation to the array which even thought Tensorflow says it is essentially free [45], a performance hit was indeed noticed.

The only way to avoid having to transpose the axes of the matrix is to make use of the Einstein summation conventions, such that this:

```
c = jax.numpy.matmul(temp.transpose(1,2,3,0), A.T)
```

becomes this:

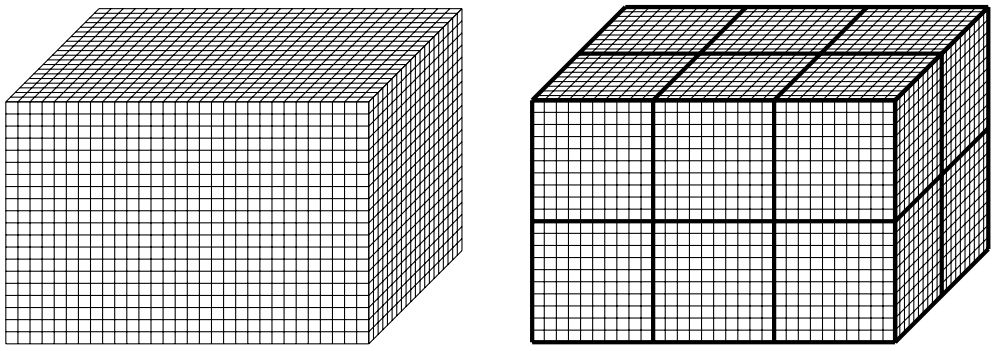
```
c = jax.numpy.einsum('ijkl,mi->jkml', temp, A)
```

As a proof of concept, the benchmarks on Table 6.1 show how transposing starts to affect performance and how `einsum` seems to be less affected by it. Regardless, a `matmul` operation as the ones done in the 2D benchmarks is included to show how not transposing does yield better results

JAX OPS	Execution time
`A[1:,:,:]-A[:-1,:,:]`	35.9 ms
`matmul(A.transpose(2,1,0), K.T)`	20.4 ms
`einsum('ijl, ki -> jkl', A, K)`	15.4 ms
`einsum('ijl, lk -> ikj', A, K)`	12.2 ms
`matmul(A, K.T)`	7.15 ms

Table 6.1.: Timing comparison between methods to perform a difference between arrays of size $(8 \times 128)^2$ run on a single TPUv3 core.

As this thesis has proved, using a single matrix does not yield effective results for grid per core larger than $(8 \times 128)^2$ elements, thus, a split in tiles should also be applied here if you need to get some speedup over the regular vector method. However, because the derivatives are carried in the three dimensions, the tiles should also be applied on all of them. Assuming that the time dimensions does not exists on the Listing 6.2 case, Figure 6.2 shows how the grids with the parameters, such as `temp` and `salt` should be distributed before computing the derivatives.



(a) Current 3D array **(b)** 6D array using tiles

Figure 6.2.: Illustration of the tile split in 3D

Once the split has been done, we double the amount of dimensions of each array, thus considering tiles of size $m = 1024^2$, accessing the element `temp[1025,1,4106,tau]` would

become `temp[1,3,0,1,1,10,tau]` and so forth. It is easy to see how this method may become unfeasible to work with when dealing with higher dimensions, which would still require some transposing tinkering in order to perform simple finite difference methods. Because reshaping in every calculation does involve adding a large overhead, I would propose two different approaches to take in this case:

- **Abstraction:** Veros already deals with a lot of its nuisances using the abstraction capabilities that Python provides. Hence, one could add an abstraction layer so that there was a translation between the the index operations of a 4D array to the 7D array without the end user noticing it. In doing so, providing a clean user experience but still getting the performance boost of splitting the arrays into tiles. This method would still need to be compatible with the rest of operations found on Veros.

- **Constrain size:** The other approach would be to limit the grid size per core. As the performance hit of using sparse matrix usually appears after $N^2 \sim (8 \times 128)^2$, avoiding reaching those limits would not require using the tiles method and still gain the performance boost. However, this method would rely exclusively in increasing the number of devices to create a larger grid size and it would not be applicable to other architectures.

Of course both methods would need an abstraction layer to perform the derivatives using `einsum` when transposing was needed, but since Veros does already provide with such abstraction, the change here would not be as disrupted as the proposed above.

In conclusion, despite being possible, improving the performance by $\sim 20\%$ may not be worth it, considering all the required changes.

Turbulent Kinetic Energy

The last benchmark deals with the parameterization of turbulence in order to quantify its effects on large-scale flows without the need of explicitly calculating it.

Other than finite difference operations similar to those on the isoneutral mixing benchmarks, this computation involves a tridiagional matrix solver based on the Thomas Algorithm. The code implementation can be seen in Listing 6.3

One thing that is not highlighted here is that `a`, `b`, `c` are matrices of matrices, therefore it is not as the Thomas algorithm can be replaced with a more parallel method such as the Augmented Block Cimmino Distributed (ABCD) method that allows the solver to use matrix operations. Instead, the focus was on replacing the operations inside the `compute_primes` to make use of the MXU, but no approach was found to report good results. In addition

```
@jax.jit
def solve_tridiag(a, b, c, d):

    def compute_primes(last_primes, x):
        last_cp, last_dp = last_primes
        a, b, c, d = x
        cp = c / (b - a * last_cp)
        dp = (d - a * last_dp) / (b - a * last_cp)
        new_primes = jnp.stack((cp, dp))
        return new_primes, new_primes

    diags_stacked = jnp.stack(
        [arr.transpose((2, 0, 1)) for arr in (a, b, c, d)],
        axis=1
    )
    _, primes = jax.lax.scan(compute_primes, jnp.zeros((2, *a.shape
    [:-1])), diags_stacked)

    def backsubstitution(last_x, x):
        cp, dp = x
        new_x = dp - cp * last_x
        return new_x, new_x

    _, sol = jax.lax.scan(backsubstitution, jnp.zeros(a.shape[:-1])
    , primes[::-1])
    return sol[::-1].transpose((1, 2, 0))
```

Listing 6.3: Tridiagonal matrix solver with diagonals a b c and RHS vector d

to the main computation, the transposing of the matrices in order to use `jax.lax.scan` was also a redundant operation and despite having a minor impact on performance on TPUs due to the dedicated transposing unit, it was something that I could fix by adapting `jax.lax.scan` to work on arbitrary axes. Hence, a patch was submitted to upstream[1] which as of the writing of this thesis has not been merged yet.

In all fairness, the lack of profiler did not allow to know whether the tridiagonal solver was a bottleneck on TPU. The profiling obtained on the brief period where it was possible to store tracings did not indicate what operation was actually taking the most time, but the assumption was that `region.27` on Figure 6.3 corresponded to the `solve_tridiagonal` function. In contrast to the profiler of the Equation of State, the TKE operations do not seem to be as easily fused. This can be argued as being due to the current inefficiency of the XLA compiler to optimize across operations other than consecutive pure algebraic operations.

To conclude and regardless of my best efforts, no use of the MXU could be achieved for the tridiagonal solver and therefore there was no method to run it on a TPU core efficiency.

[1] https://github.com/google/jax/pull/4591

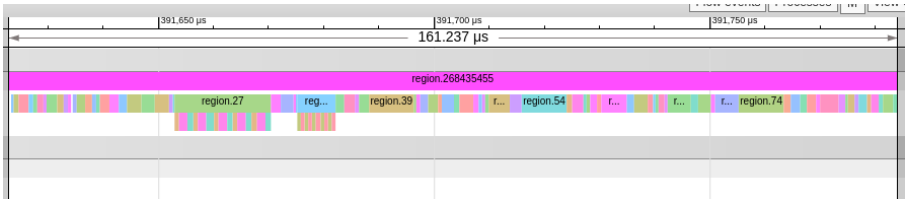


Figure 6.3.: Early profiler screenshot on TPUv2 of the Thermal Kinetic Energy benchmark

6.2 General Guidelines on Good Candidates

The results of the assessment point out that in order to achieve efficiency on TPUs, the speedups would need to happen on the derivatives calculation and those speedups should be enough to cover the reduced performance of the element wise operations. Moreover, those speedups would only be achieved by increasing the complexity of the model. With all of that in mind, I do not believe that Veros, despite all its merits, may be a good candidate for using TPUs.

Henceforth it begs the question of what type of simulations could be good candidates to be run on Cloud TPUs. Below there is an outline of several conditions that would favor the porting to a TPU:

- **Extensive matrix multiplications use:** From the results of the tensor product PDE solver, a simulation where matrix multiplication is already required would be ideal for TPUs as no work around would need to be found. The performance difference between performing the dot product on TPU instead of GPU are clear, and as long as the simulation does not require high-precision results, it would greatly benefit from the use of systolic arrays and HBM. For instance, models based on derivations using finite elements may find on TPU a good candidate to speedup their calculations.

- **2D Lattices / Operations:** As seen on the assessment of Veros, dealing with higher dimensions may be an important factor to consider as the manipulation of arrays into tiles gets confusing quickly since tiling involves doubling the number of dimensions.

- **Highly parallelizable with repetitive tasks:** One of the reason why Veros may not be right for TPUs despite having a method to optimize the finite difference is its versatility. If instead of a general circulation model that targets a wide range of applications, a more focused solver that deals with less amount of variety on its operations may find a possible way to make use of the MXU for operations with a small space locality. Even if the operation does not yield great speedups, the larger the number of iterations, the best it will perform at the end. Additionally and despite the low results yield on the Ising Model, JAX take on distributed scalable programs works perfectly with the idea of having each device generate their own

arrays and share the computed metrics on the collective, and due to the impressive high connectivity of TPUs, there is no bottleneck found for these collective operations as you could find in more traditional clusters.

- **Mixing Machine learning with numerical computations:** In recent years there has been an increasing interest on getting the advantage of using Deep Learning to simulate fluid dynamics [46, 47]. The approach tends to rely on an mix usage of classical numerical simulations and deep learning interference. For instance Obiols-Sales *et al.* [46] uses numerical simulations to warm-up the system, then feed it into a trained neural network and finally performs the classical numerical iterations as refinement as seen in Figure 6.4. In contrast Kochkov *et al.* [47] performs the numerical solver on a lower resolution and uses DL to interpolate the values and achieve higher resolutions. Both methods ML parts would do perfect usage of the MXU, however it does not seems that there is any usage when performing the conventional solvers. Henceforth, this type of applications in addition to the techniques described on this thesis would manage to utilize all the available power that a TPU can offer.

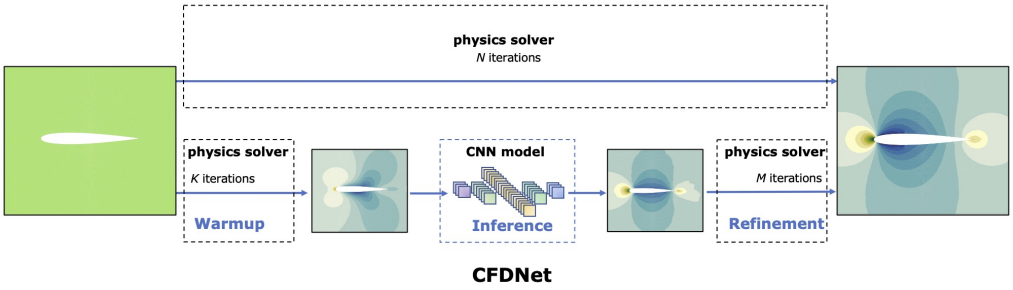


Figure 6.4.: Comparison of the traditional physics solver simulation with CFDNet. CFDNet integrates the domain-specific physics solver for warmup, followed by the neural network for inferring the steady state, and the final iterative refinement stage to correct the solution of the CNN and satisfy the convergence constraints.

Those are some of the examples where I believe TPUs would be the right candidate to be used. Nevertheless, all the cases where some modifications were needed to make proper use of the MXU would not perform as well on other hardware.

7 Conclusion

The aim of this thesis was to evaluate whether Google Cloud Tensor Processing Units could be a good candidate to motivate the transition of high efficiency programming languages into large scale High Performance Computing. In spite of the initial limitations of the tools, the constant issues found at the beginning and the lack of physical access to the devices, we have managed to gain some insight on how TPUs perform when doing physics calculations.

A new method for performing finite differences derivatives, which can also be applied to similar stencil operations, has been proposed. Using three different general benchmarks of partial derivatives equations, we have seen that our method is capable of achieving up to a $2\times$ speedup, e.g. 100% faster, on a single TPU with respect to the conventional vector approach. In spite of using lower precision floating points formats, no visual discrepancy has been found on any of the simulations preformed. Regardless, choosing to simulate them on lower precision for a large amount of time could lead to noticeable divergences, as its linear relation suggests. However, these discrepancies are less significant when working with larger numbers due to the use of brain-floating points instead of standard IEEE half floating points.

When comparing our chosen framework with a previous study that implemented an Ising model targeting TPUs, we have found that despite the good results obtained on the partial differential equation solvers, the outcome of the method developed in this thesis does not yield similar good results as the reported ones with TensorFlow. Nonetheless, due to the recent release of profiling tools, we believe that this performance gap can be reduced as indicated by some preliminary analysis.

Finally, with the information gathered throughout this project, we make the assessment that an existing ocean simulation, Veros, would not benefit from a porting to TPUs at this moment, but that there are a wide range of physical simulations that would take advantage of the TPUs capabilities.

Overall, we have proved that by applying transformations to existing algorithms we are able to make them perform more efficiently on matrix engines. By doing so, opening the

door to the creation of new approaches targeting matrix multiplications as their main computational operation, to perform non dot related calculations. As the usage of JAX and TPUs becomes easier, we believe that now is the moment to continue evaluating different types of calculations that would clearly define whether the computational physics community should embrace the use of matrix engines for general physics calculations.

7.1 Future Work

We have seen the importance of matrix multiplication when using TPUs across the thesis. Because Veros was largely dependent on the finite difference method, we based our research on them. However, there are other calculations that rely on matrix multiplications and do not require complex transformations in order ot make use of the specialized hardware. If there was more time remaining, the next chapter of this thesis would probably be the study of finite element methods, which I believe may yield impressive performance results when used on TPU clusters.

Another interesting continuation of this thesis would be to study the effects of our results for a single large scale long running simulation. As the profiling and tracing tools has proven to ease the study of this implementations greatly as well as the new JAX on TPU workflow, complex function would not require the inconvenience to map *by hand* where the bottlenecks may be found and the speed of the porting would be massively reduced. Therefore, I believe that on the current state of things, now is a good starting point for an exploration on applying the proposed methods on practical simulations.

As mentioned at the end of Chapter 6, the surge of hybrid numerical simulations that also rely on Deep Learning inference creates the perfect scenario for the proposed methods which, even though they can not compete with current GPUs, they will yield better performance that conventional methods if they already are expected to be run on Cloud TPUs.

8 Bibliography

[1] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. “A Survey of General-Purpose Computation on Graphics Hardware”. In: *Computer Graphics Forum* 26.1 (2007), pp. 80–113.

[2] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. “Motivation for and Evaluation of the First Tensor Processing Unit”. In: *IEEE Micro* 38.3 (May 2018), pp. 10–19.

[3] John Shalf. “The future of computing beyond Moore’s Law”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378.2166 (Mar. 6, 2020), p. 20190061.

[4] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. “Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs”. In: *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Nov. 2016, pp. 409–420.

[5] Shaveta Dargan, Munish Kumar, Maruthi Rohit Ayyagari, and Gulshan Kumar. “A Survey of Deep Learning and Its Applications: A New Paradigm to Machine Learning”. In: *Arch Computat Methods Eng* 27.4 (Sept. 1, 2020), pp. 1071–1092.

[6] Xiaoxuan Liu, Livia Faes, Aditya U Kale, *et al.* “A comparison of deep learning performance against health-care professionals in detecting diseases from medical imaging: a systematic review and meta-analysis”. In: *The Lancet Digital Health* 1.6 (Oct. 1, 2019), e271–e297.

[7] D. Steinkraus, I. Buck, and P.Y. Simard. “Using GPUs for machine learning algorithms”. In: *Eighth International Conference on Document Analysis and Recognition (ICDAR’05)*. Eighth International Conference on Document Analysis and Recognition (ICDAR’05). Aug. 2005, 1115–1120 Vol. 2.

[8] Kyoung-Su Oh and Keechul Jung. “GPU implementation of neural networks”. In: *Pattern Recognition* 37.6 (June 1, 2004), pp. 1311–1314.

[9] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. “Accelerating Deep Convolutional Neural Networks Using Specialized Hardware”. In: (), p. 4.

[10] *Volta: Performance and Programmability*. URL: https://www.computer.org/csdl/magazine/mi/2018/02/mmi2018020042/13rRUEgs2yF (visited on May 6, 2021).

[11] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. "DaVinci: A Scalable Architecture for Neural Network Computing". In: *2019 IEEE Hot Chips 31 Symposium (HCS)*. 2019 IEEE Hot Chips 31 Symposium (HCS). Aug. 2019, pp. 1–44.

[12] Constantinos Evangelinos and Chris N Hill. "Cloud Computing for parallel Scientific HPC Applications: Feasibility of running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2." In: (), p. 6.

[13] Dion Häfner, René Løwe Jacobsen, Carsten Eden, Mads R. B. Kristensen, Markus Jochum, Roman Nuterman, and Brian Vinter. "Veros v0.1 – a fast and versatile ocean simulator in pure Python". In: *Geoscientific Model Development* 11.8 (Aug. 16, 2018), pp. 3299–3312.

[14] Norman P Jouppi, Cliff Young, Nishant Patil, *et al.* "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: (), p. 17.

[15] James Bradbury, Roy Frostig, Peter Hawkins, *et al. google/jax*. Version 0.2.5. 2018.

[16] *XLA: Optimizing Compiler for Machine Learning*. TensorFlow. URL: https://www.tensorflow.org/xla (visited on May 6, 2021).

[17] Evangelos Vasilakis. "An instruction level energy characterization of arm processors". In: *Foundation of Research and Technology Hellas, Inst. of Computer Science, Tech. Rep. FORTH-ICS/TR-450* (2015).

[18] Gordon E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35.

[19] Thomas N. Theis and H.-S. Philip Wong. "The End of Moore's Law: A New Beginning for Information Technology". In: *Computing in Science Engineering* 19.2 (Mar. 2017), pp. 41–50.

[20] John M. Shalf and Robert Leland. "Computing beyond moore's law". In: *Computer* 48.12 (2015), pp. 14–23.

[21] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[22] Martin Burtscher and Keshav Pingali. "An efficient CUDA implementation of the tree-based barnes hut n-body algorithm". In: *GPU computing Gems Emerald edition*. Elsevier, 2011, pp. 75–92.

[23] Petr Pospichal, Jiri Jaros, and Josef Schwarz. "Parallel genetic algorithm on the cuda architecture". In: *European conference on the applications of evolutionary computation*. Springer, 2010, pp. 442–451.

[24] Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Müller-Wittig. “Accelerating molecular dynamics simulations using Graphics Processing Units with CUDA”. In: *Computer Physics Communications* 179.9 (2008), pp. 634–641.

[25] Tobias Preis, Peter Virnau, Wolfgang Paul, and Johannes J. Schneider. “GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model”. In: *Journal of Computational Physics* 228.12 (July 1, 2009), pp. 4468–4477.

[26] Xi Qin, Wenzhe Zhang, Lin Wang, Yuxi Zhao, Yu Tong, Xing Rong, and Jiangfeng Du. “An FPGA-based hardware platform for the control of spin-based quantum systems”. In: *IEEE Transactions on Instrumentation and Measurement* 69.4 (2019), pp. 1127–1139.

[27] Chen Yang, Tong Geng, Tianqi Wang, Rushi Patel, Qingqing Xiong, Ahmed Sanaullah, Chunshu Wu, Jiayi Sheng, Charles Lin, and Vipin Sachdeva. “Fully integrated FPGA molecular dynamics simulations”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pp. 1–31.

[28] *Domain-specific hardware accelerators | Communications of the ACM*. URL: https://dl.acm.org/doi/abs/10.1145/3361682?casa_token=mO44yGzT1noAAAAA:GWa9ajPlw6qI5ebjlYcmV9zXGiIVVsJKcqFgnzUZonuFg5Q1njb2G9Ver9JKSUgPfvmiwwQiChIg (visited on May 19, 2021).

[29] Jens Domke, Emil Vatai, Aleksandr Drozd, *et al.* “Matrix Engines for High Performance Computing:A Paragon of Performance or Grasping at Straws?” In: *arXiv:2010.14373 [cs]* (Feb. 27, 2021). arXiv: 2010.14373.

[30] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. “Deep learning with limited numerical precision”. In: *International conference on machine learning*. PMLR, 2015, pp. 1737–1746.

[31] Vasily Volkov and James W. Demmel. “Benchmarking GPUs to tune dense linear algebra”. In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. SC ’08. Austin, Texas: IEEE Press, Nov. 15, 2008, pp. 1–11.

[32] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. “Nvidia tensor core programmability, performance & precision”. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 522–531.

[33] Anumeena Sorna, Xiaohe Cheng, Eduardo D’azevedo, Kwai Won, and Stanimire Tomov. “Optimizing the fast fourier transform using mixed precision on tensor core hardware”. In: *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*. IEEE, 2018, pp. 3–7.

[34] *Implementation of IFS Cycle 47r2 - Forecast User - ECMWF Confluence Wiki*. URL: https://confluence.ecmwf.int/display/FCST/Implementation+of+IFS+Cycle+47r2 (visited on May 20, 2021).

[35] Samuel S. Schoenholz and Ekin D. Cubuk. “JAX, M.D.: A Framework for Differentiable Physics”. In: *arXiv:1912.04232 [cond-mat, physics:physics, stat]* (Dec. 3, 2020). arXiv: 1912.04232.

[36] *Using Cloud TPU Tools*. Google Cloud. URL: https://cloud.google.com/tpu/docs/cloud-tpu-tools (visited on May 19, 2021).

[37] Donald E. Barrick. “A coastal radar system for tsunami warning”. In: *Remote Sensing of Environment* 8.4 (1979), pp. 353–358.

[38] Robert E. Lynch, John R. Rice, and Donald H. Thomas. “Direct solution of partial difference equations by tensor product methods”. In: *Numer. Math.* 6.1 (Dec. 1, 1964), pp. 185–199.

[39] Kun Yang, Yi-Fan Chen, Georgios Roumpos, Chris Colby, and John Anderson. “High Performance Monte Carlo Simulation of Ising Model on TPU Clusters”. In: *arXiv:1903.11714 [physics]* (Nov. 17, 2019). arXiv: 1903.11714.

[40] William J. Dally, Yatish Turakhia, and Song Han. “Domain-specific hardware accelerators”. In: *Commun. ACM* 63.7 (June 18, 2020), pp. 48–57.

[41] P. Pfeuty. “An exact result for the 1D random Ising model in a transverse field”. In: *Physics Letters A* 72.3 (July 9, 1979), pp. 245–246.

[42] Lars Onsager. “Crystal Statistics. I. A Two-Dimensional Model with an Order-Disorder Transition”. In: *Phys. Rev.* 65.3 (Feb. 1, 1944), pp. 117–149.

[43] Dion Häfner. *dionhaefner/pyhpc-benchmarks*. Apr. 16, 2021.

[44] *Thermodynamic Equation of SeaWater TEOS-10*. URL: http://www.teos-10.org/software.htm (visited on May 15, 2021).

[45] *Performance Guide | Cloud TPU*. Google Cloud. URL: https://cloud.google.com/tpu/docs/performance-guide (visited on May 14, 2021).

[46] Octavi Obiols-Sales, Abhinav Vishnu, Nicholas Malaya, and Aparna Chandramowlishwaran. “CFDNet: a deep learning-based accelerator for fluid simulations”. In: *Proceedings of the 34th ACM International Conference on Supercomputing* (June 29, 2020), pp. 1–12. arXiv: 2005.04485.

[47] Dmitrii Kochkov, Jamie A. Smith, Ayya Alieva, Qing Wang, Michael P. Brenner, and Stephan Hoyer. “Machine learning accelerated computational fluid dynamics”. In: *arXiv:2102.01010 [physics]* (Jan. 28, 2021). arXiv: 2102.01010.

A Finite Differences

Considering $F(x)$ to be a well behaved function with itself and its derivative being single valued, the Taylor theorem

$$F(x+h) = F(x) + hF'(x) + \frac{1}{2}h^2F''(x) + \frac{1}{6}h^3F'''(x) + O(h^4) \tag{A.1}$$

and likewise on the other direction

$$F(x-h) = F(x) - hF'(x) + \frac{1}{2}h^2F''(x) - \frac{1}{6}h^3F'''(x) + O(h^4) \tag{A.2}$$

Therefore, if by adding them we obtain the following relation

$$F(x-h) + F(x+h) = 2F(x) + h^2F''(x) + O(h^4) \tag{A.3}$$

which can the be reformulated as the finite difference equation

$$F''(x) = \frac{F(x-h) - 2F(x) + F(x+h)}{h^2} - O(h^2) \tag{A.4}$$

hence, assuming an error in the orders of h^2, we have

$$F''(x) \simeq \frac{F(x-h) - 2F(x) + F(x+h)}{h^2} \tag{A.5}$$

Similarly, if we were to subtract (A.2) from (A.1), we would end up with

$$F'(x) \simeq \frac{F(x+h) - F(x-h)}{2h} \tag{A.6}$$

which calculates the slope of the tangent on the B point. of Figure A.1, by using the point A and C. Hence, (A.8) is called the central-difference scheme. Computing the slope on B by only using one of the other points instead, we would be using the forward-difference scheme (using C)

$$F'(x) \simeq \frac{F(x+h) - F(x)}{h} \tag{A.7}$$

and the backwards-difference scheme

$$F'(x) \simeq \frac{F(x) - F(x-h)}{h} \tag{A.8}$$

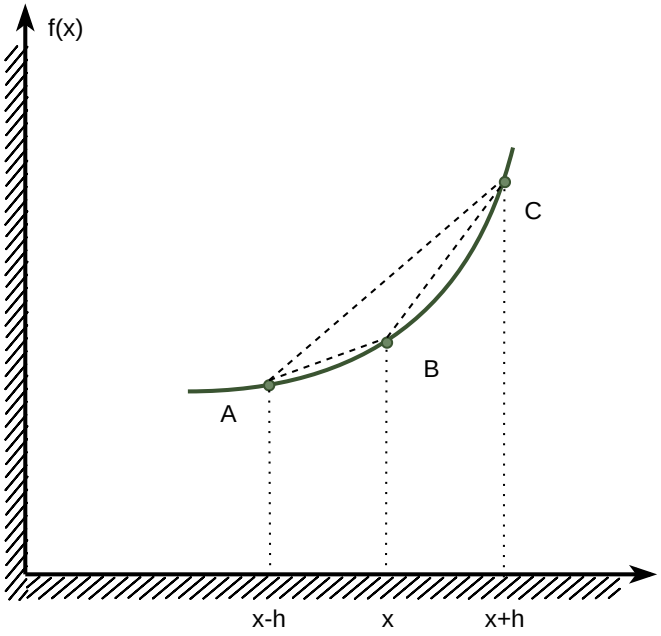


Figure A.1.: Graphical representation of the tangent of a curve.

B

Hardware Specifications

A detail of the hardware used during the benchmarks is shown.

	TPU Unit v3	TPU Chip v3	TPU Core v3	NVIDIA RTX 2080 Ti
Architecture	-	-	-	Turing
Memory size (GiB)	128	32	16	12
Memory type	HBM2	HBM2	HBM2	GDDR6
Memory speed (Gb/s)	900	900	900	616.0
Peak TFLOPs (16 bits)	420	123	61.5	26.90
Peak TFLOPs (32 bits)	16	4	2	13.45
TDP (Watts)	1800	450	225	250
Clock rate (MHz)	900	900	900	1350
Cloud price ($/hour)	8.80	-	-	-

Table B.1.: Comparison between the specifications of all the hardware used during benchmarks.

	NVIDIA Tesla V100	NVIDIA Tesla P100
Architecture	Volta	Pascal
Memory size (GiB)	16	16
Memory type	HBM2	HBM2
Memory speed (Gb/s)	897	732.2
TFLOPs (16 bits)	28.26(112)	19.05
TFLOPs (32 bits)	14.13	9.526
TDP (Watts)	300	250
Clock rate (MHz)	1245	1190
Cloud price ($/hour)	2.55	1.60

Table B.2.: Extra GPU accelerators added for reference.

List of Figures

List of Tables

Listings