Solving Partial Differential Equations by the Finite Difference Method on a Specialized Processor

Simen Håpnes

Thesis submitted for the degree of
Master in Computational Science: Physics
60 credits

Department of Physics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Autumn 2021

Solving Partial Differential Equations by the Finite Difference Method on a Specialized Processor

Simen Håpnes

Solving Partial Differential Equations by the Finite Difference Method on a Specialized Processor

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

The emergence of specialized processors in recent years has largely been driven by providing high computational performance for artificial intelligence (AI) workloads. However, these technological advancements are also of interest for high performance computing (HPC) for general scientific applications. In this thesis, selected stencil-based numerical schemes for solving partial differential equations (PDEs) have been implemented for execution on the Graphcore intelligence processing unit (IPU), a processor with 1,472 cores and distributed memory chunks of 624 kB located near each core. The schemes were also implemented to be executed on two 32-core CPUs, with the OpenMP framework.

The heat equation was solved on the IPU for structured 2D, and 3D meshes. The computations and problem sizes were scaled to execute in parallel on up to 16 IPUs. For all executions, the problem size pushed the limit of the available on-chip memory. Additionally, the PDE system of the Aliev-Panfilov model for cardiac excitation was solved for a structured 2D mesh. This demonstrates the IPU's applicability for a real-life physics-based application. The computations involved iteratively applying 5-point and 7-point stencils, for the 2D and 3D systems, respectively. The IPU was demonstrated to achieve remarkable performances, achieving a throughput of up to 1.44 TFLOPS. Careful programming led to an effective use of the distributed in-processor memory of the IPU, which is designed to provide high memory bandwidth. The 3D heat equation reached 5.15 TB/s memory bandwidth on one IPU.

The extension to multi-IPU computations also showed performances consistent with the scalable design of IPU systems, so-called IPU-PODs. The multi-IPU performance of the 2D heat equation achieved a better speedup than its 3D counterpart, while both showed performance increases that scaled well.

This thesis demonstrates an attempt to apply a specialized AI-processor to selected general scientific computing workloads. In this context, the advantages, challenges, and weaknesses of employing the IPU have been discussed.

Acknowledgments

I would like to thank my supportive supervisors: Xing Cai, Are Magnus Bruaset, and Morten Hjorth-Jensen, for the advice and guidance you have given. I also must sincerely thank Xing and Are for taking the time to have regular meetings, for providing thorough feedback to my drafts, and for teaching me a lot about computing research.

I would also like to thank Truls Edvard Stokke and Alexander Titterton for all you have taught me about IPUs, and for being available to answer my many technical questions. Additionally, a big thank you to Truls for contributing with codes in the early work of this thesis.

I am also incredibly thankful for the weekly meetings with Xing, Are, Truls, and Alex. These meetings have truly taught me a lot and inspired me to work hard.

Last but not least, a warm thank you to Felicia Jacobsen and Jon Andre Ottesen for having taken time out of your busy lives to review this thesis thoroughly.

Contents

List of Abbreviations

AI Artificial Intelligence.

BSP Bulk Synchronous Parallel.

CPU Central Processing Unit.

GPU Graphics Processing Unit.

HPC High Performance Computing.

IPU Intelligence Processing Unit.

MIMD Multiple Instruction, Multiple Data.

ML Machine Learning.

PDE Partial Differential Equation.

Chapter 1

Introduction

In recent years, many processors that are specialized for machine learning applications have emerged [1]. The IPU is one of these processors [2]. It is designed to be used as an accelerator for machine learning (ML) and for AI computations [2, 3]. This thesis tests the IPU's applicability for a specific class of algorithmic patterns often found in general scientific computing. More specifically, various stencil-based computations were implemented and benchmarked by solving PDEs with the finite difference method. Such problems are typical workloads in computational science, which require high performance.

Computational science is an important tool to many scientific fields and is only at the beginning of centuries of growth [4]. The number of applications of computational science is large, ranging from data science, quantum computing, image processing, evolutionary algorithms, process simulation, deep learning and big data [5]. This has led to a strive for computers and algorithms with higher performance, thus allowing more accurate simulations, data analyses of larger amounts of data, or faster program executions.

Increasing computational performance is a complex goal which the field of HPC addresses. Wilson [4] argues that there are four main barriers in increasing the performance in computational science: software productivity, error control, algorithmic development and the fostering of technological advances. This thesis emphasizes the fourth barrier. In this sense, in order to increase performance in scientific computing it is important to choose suitable hardware and software. In extension, it is essential to implement highly optimized codes that exploit characteristics of the software and hardware, as well as the features of the specific application at hand.

During the last year, the IPU has been used in several studies. Most of these have focused on its potential in deep neural networks [6]. In 2021, Graphcore released their first results for IPU's performance in *MLPerf*: a popular

benchmark for image classification (ResNet-50) and natural language processing (BERT) [7]. Further, some studies have also explored the IPU's applicability for HPC workloads. For instance, one study performed microbenchmarks for latency, bandwidth, and matrix multiplications on the IPU, and compared the performance to the theoretical limits [8].

This thesis presents the application of selected algorithmic patterns found in HPC workloads. One computational problem studied in the thesis features some similarities to a recent study conducted by Louw and McIntosh-Smith [9]. That study employed two stencil-based computations: a Gaussian blur image filter and a 2D Lattice Boltzmann fluid simulation. They achieved performances comparable to that of modern graphics processor units (GPUs) [9]. This thesis on the other hand, covers different computations, and additionally employs 3D meshes. First, the heat equation was discretized by the finite difference method, and solved on the IPU, both for structured 2D and 3D meshes. Second, the scalability of IPU systems was studied by scaling these computations from 1 IPU up to 16 IPUs. Additionally, a real-life application of a numerical model for cardiac excitation was implemented, namely the Aliev-Panfilov model. This model requires a much larger number of operations per stencil compared to the heat equation.

The scientific problems that this thesis seeks to answer can be summarized in four research questions:

1. How good performance can the IPU provide for the selected scientific computing workloads?
2. What technical challenges does low-level IPU programming involve?
3. What limitations does the IPU exhibit for general scientific computing workloads?
4. What differences and similarities are found when asking the three aforementioned questions for multi-IPU systems compared to single-IPU systems?

The areas studied in this thesis contribute to computational science by trying to adopt technological advancements made by the semiconductor industry. The IPU is one of many processors that have emerged for AI workloads [1], and this thesis demonstrates the development processes of scientific programs on the IPU, which also highlights some general aspects of parallel computing and heterogeneous computing. All the developed codes can be found at `https://github.com/simehaa/IPU`.

Chapter 2

Background

This chapter consists of three sections, a brief history of processors, an overview of the IPU's hardware architecture, and the key design principles behind the IPU.

2.1 A Brief History of Processors

2.1.1 The CPU and GPU

In November 1971, a 4-bit central processing unit (CPU) under the product name *Intel 4004* was released [10]. This was the world's first commercial single-chip microprocessor[1]. It marked the beginning of decades of exponential growth of compute power [10]. Today, the CPU is still one of the main components of a computer, but it has undergone a lot of development. A CPU consists of one or more *cores*, which is a computing unit that performs instructions such as arithmetic, logic, and I/O operations. A central concept in the history of the CPU's development was the introduction of multi-core CPUs. They turned out to provide much higher performance in many applications, and a higher performance per watt ratio [11].

An important milestone in the history of computing technology was the adoption of *parallelism*. Traditional computer programs were written to be executed sequentially, where one instruction executes after another. However, many computer programs consist of workloads that can be broken down into several independent parts. In order to take advantage of this, technology also had to support it, which is possible through the usage of multiple workers. One article argues that power, memory, and instruction level parallelism walls are some of the factors that have forced microprocessor manufacturers to produce parallel

[1]The Intel 4004 contained 2,300 transistors, ran at a clock speed of up to 740 kHz, and delivered 60,000 instructions per second while dissipating 0.5 watts [10].

microprocessors [12]. Today, there is a large demand for parallel computer programs, since they provide high performance in the tasks of which they solve. Modern CPUs typically exhibit parallelism through having multiple cores, typically in the range from two to several tens.

There are also several other types of processing units, with specialized functionality, most notably the GPU. This processor was specifically designed to perform 3D graphics rendering, which requires many floating-point calculations [13]. A GPU is a many-core processor which, similarly to the CPU, has undergone a long path of development. The modern GPU is a highly parallel programmable processor, often exhibiting core counts in the range of several thousands, and it features peak arithmetic and memory bandwidth that substantially outpaces its CPU counterpart [14]. The modern GPU has a wide list of applications and is considered a general-purpose processor. Its ability to perform a great variety of tasks is referred to as the general-purpose GPU [15], which has become a common term in the literature. Some of the driving forces of modern GPU development are the gaming industry, general purpose scientific computing, and machine learning applications: most notably the training of deep neural networks.

2.1.2 Specialized Architectures

An early observation about the doubling of the density of components per integrated circuit at regular intervals is popularly referred to as Moore's law. It served as an accurate prediction and became a central driving force in production of new processors for many years [16]. Today on the other hand, AI performance seems to be a central driving force to many processor manufacturers. An analysis done by *OpenAI* discusses that from 1959 to 2012, the largest AI-models exhibited a trend of a 2-year doubling time, like that of Moore's law. It was uncommon to adopt GPUs for AI applications before 2012, which marks the beginning of a second era of AI performance. In this modern era, the amount of compute in the largest AI models saw a rapid increase to a 3.4-month doubling time [17]. A 2020 survey of machine learning accelerators[2] demonstrates that there are many AI-specialized processors that have emerged in the recent years [1]. Although these processors are specialized for machine learning applications, they could also be of high interest for other computing fields if they provide the flexibility to be used on other workloads. It is important to try to adopt these technological advancements in order to overcome one of the barriers of increasing performance in computational science.

A heterogeneous computing system is a type of system that consists of mixed architectures, and the need for these systems was discussed as early as 1993 [18].

[2]The terms *AI accelerator*, *ML accelerator*, *machine intelligence accelerator*, and *deep neural network accelerator* are used in the literature. This can create confusion since they all more or less refer to processors that provide high performance in training and inference of advanced or deep neural network models.

Programs that can be broken down into various homogeneous subtasks can benefit from such systems, because a user can decompose and assign the subtasks to the various suitable architectures to reach an overall faster execution time [19]. Today, heterogeneous computing systems are very common, and some examples include general purpose GPUs, field-programmable gate arrays, Google's tensor processing unit, Intel's neural network processor, and AMD's accelerated processing unit [20]. An IPU machine is also a heterogeneous computing system since it consists of IPUs and a CPU. Program executions on IPU machines take advantage of both processors.

2.2 IPU Hardware Architecture

2.2.1 Overview

At the time of writing, there are two generations of the IPU, the *Colossus GC2 MK1* IPU and the *Colossus GC200 MK2* IPU. This study intends to emphasize the bleeding-edge technology, and only studies the second generation. An illustration of the MK2 IPU is shown in Figure 2.1.

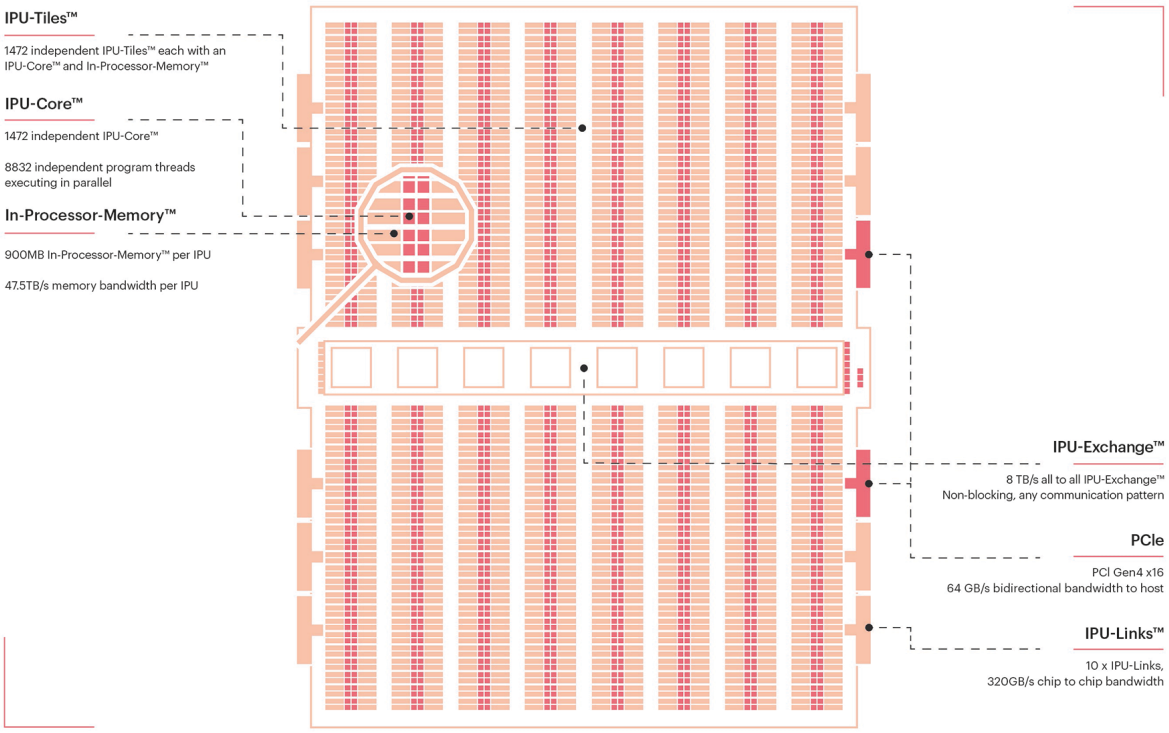
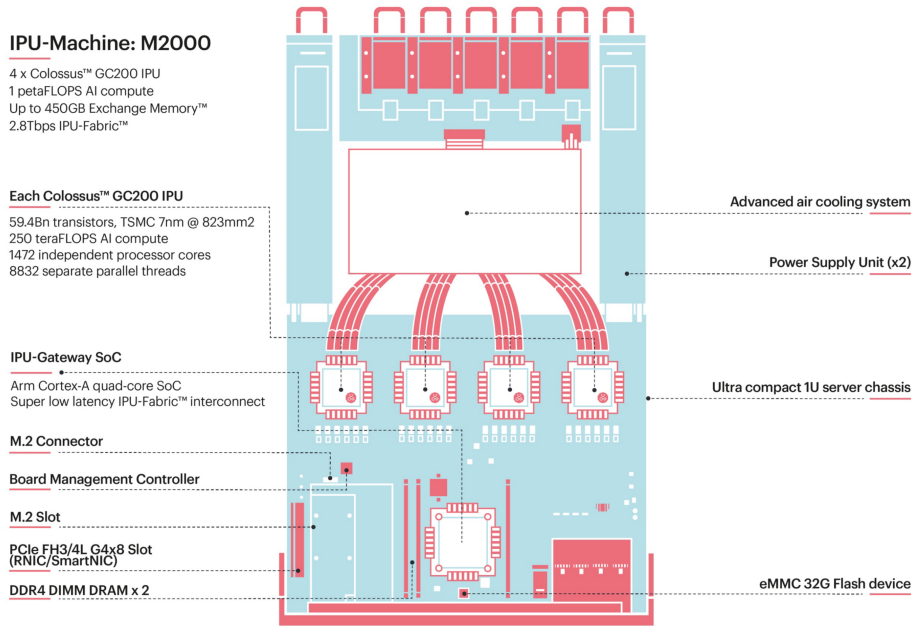


Figure 2.1: An illustration of the *Colossus MK2 GC200 IPU* [2]. The processor features 1,472 independent tiles, each containing a core and 624 kB local in-processor memory.

Figure 2.1 shows the hardware components found on an IPU. This processor is designed for exhibiting a large count of independent cores, and for scalability. Therefore, the IPU contains components that provide high bandwidth of communication between the tiles, namely the IPU-Exchange, featuring 8 TB/s all-to-all bandwidth. Additionally, IPU-Links allow for fast communication between several IPUs, featuring 320 GB/s chip-to-chip bandwidth.

Figure 2.2 shows the *M2000*: an IPU-machine that contains four *Colossus MK2 GC200* IPUs. Figure 2.2a shows an illustrative overview of the components, and Figure 2.2b shows an image of the M2000.



(a) An illustration of the *M2000*.

(b) An image of the *M2000*

Figure 2.2: An illustration and an image of the M2000, which is an IPU machine featuring four *Colossus MK2 GC200* IPUs [21]. It is designed as a blade that can be a part of larger blade servers.

The M2000 can be used as a building block in larger IPU systems. The IPU-POD64 is one of the larger systems that combines 16 M2000s, totaling at 64 IPUs [22]. To take things further, up to 1,024 IPU-POD64 can be combined, which totals 65,536 IPUs [22].

Terminology

The terms *IPU-Core*, *IPU-Exchange*, *IPU-Links*, *IPU-Tiles*, and *In-Processor-Memory* are registered trademarks of Graphcore. To simplify things, these will be referred to as *core*, *exchange*, *links*, *tiles*, and *in-processor memory*.

2.2.2 Tile

The MK2 IPU consists of 1,472 tiles. Each tile contains both a computing core and local in-processor memory. This design gives the IPU some important properties that make it differ from the traditional CPU and GPU. The in-processor memory is only accessible to its connected core during the computational phases, which means that the IPU exhibits a distributed memory model. Furthermore, the core has low-latency communication access to the in-processor memory, because it is in the core's vicinity. The in-processor memory is capable of both storing variables and executable code. This means that each individual tile can perform a completely unique computation, which ultimately makes the IPU a true *multiple instruction, multiple data* (MIMD) processor.

In the context of CPU and GPU programming, one often refers to their cores. However, it is more suitable to refer to the entire tiles in the context of IPU programming, due to the core's closely connected in-processor memory.

Core

The IPU-Core is a computing component that consists of six threads, which during execution, operates one-at-a-time in a round-robin schedule. All six threads are identical and contain two asymmetric pipelines, *main* and *aux*. Both pipelines can execute a large set of instructions [23]. The pipelines are designed for two different purposes:

- Main is designed primarily to perform control flow, address manipulation, integer arithmetic and load/store operations [23].
- Aux is designed primarily to perform floating-point computations [23].

In each thread, the two pipelines have their associated register files, the *main register file* and the *aux register file*. Both files only contain 64 bytes each, and some of this space is reserved. The remaining space is used to store values that will be used by instructions. This effectively means that the execution pipelines rely on often fetching data from the in-processor memory.

In-processor Memory

IPU exhibits a distributed memory model, where every core has its own dedicated in-processor memory, which is shared among the six threads. The memory is static random-access memory, which is typically used as fast on-chip memory. On the other hand, dynamic random-access memory is often used as off-chip memory[3] [24].

In Figure 2.3, the memory address space of the in-processor memory is illustrated.

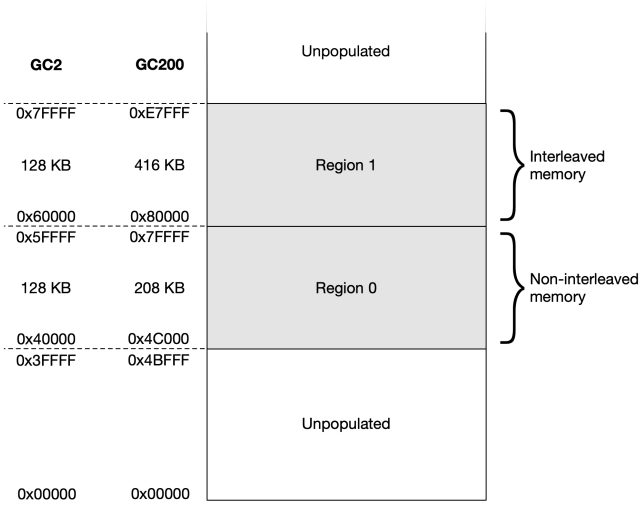


Figure 2.3: An illustration of memory address space of the in-processor memory on a single tile [25]. Note: the figure shows both IPU generations, where GC200 is the second one.

As shown in Figure 2.3, there are two regions in the address space, *non-interleaved memory* and *interleaved memory*. These two regions feature different characteristics. Both memory regions are grouped into *banks* and *elements*. A summary of the properties of the two memory regions is shown in Table 2.1.

[3]The IPU machine M2000 also include dynamic random-access memory as off-chip memory, since each machine includes 2 DDR4-2400 DIMM DRAM slots [21]. This allows for a much larger memory capacity, but also slower compared to the on-chip memory.

Region	Size	Interleaved	Banks (size)	Elements (size)
0	208 kB	No	13 (16 kB)	13 (16 kB)
1	416 kB	Yes	26 (16 kB)	13 (32 kB)

Table 2.1: An overview of the number of memory elements and banks within the two memory regions in the in-processor memory of a tile on the MK2 [23].

There are some practical differences between the two memory regions. Variables can be constrained to be stored in the interleaved region. The main advantage of the interleaved region is that it allows for up to 128-bit loads[4]. Furthermore, both memory regions support a variety of load and store instructions:

- Loading 64, 32, 16, and 8 bits.
- Storing 64 and 32 bits.

Memory Alignment

The *memory alignment* of a variable is defined as the highest power-of-two number that the memory address is divisible by. Memory addresses are typically represented as hexadecimal numbers, and the *memory alignment* is then the divisibility of these.

All load and store instructions must be naturally aligned, which means that the memory alignment of a variable must at least match the width of an operand. This requirement will be referred to as the *alignment requirement*. For example, in order to load a single precision float (4 bytes), the pointer address of that element must be divisible by 4.

It is possible for the programmer to specify the memory alignment of variables, and it can never be less than the size of the variable (in bytes). The alignment can however be constrained to higher multiples of the variable, e.g., for an array of floats (4 bytes) it can be set to 8. This could facilitate loading or storing two floats concurrently. Understanding and utilizing higher memory alignment can lead to higher performance. One reason to increase the alignment of variables is to use wider instructions, which is often referred to as vectorization. On the other hand, a constrained alignment might also impose *memory arrangement* phases, i.e., phases between execution phases where data must be re-arranged in memory. To achieve higher performance, a general goal is to achieve vectorization whilst avoiding memory arrangement phases. In summary, it is possible to explore the effects of varying alignment, and if used carefully, it might lead to great performance increases.

[4]In the interleaved memory region, it is both possible to perform one 128-bit load with the instruction `ld128` , or to concurrently do two 64-bit loads and one 64-bit store with the instruction `ld2xst64pace` [23].

2.3 Design Principles

2.3.1 Bulk Synchronous Parallelism

The IPU is designed according to the (BSP) paradigm. In short, this means that the executions cycle through four phases in this order: *synchronization*, *exchange*, *computation*, and *waiting*. Figure 2.4 conceptualizes these phases.

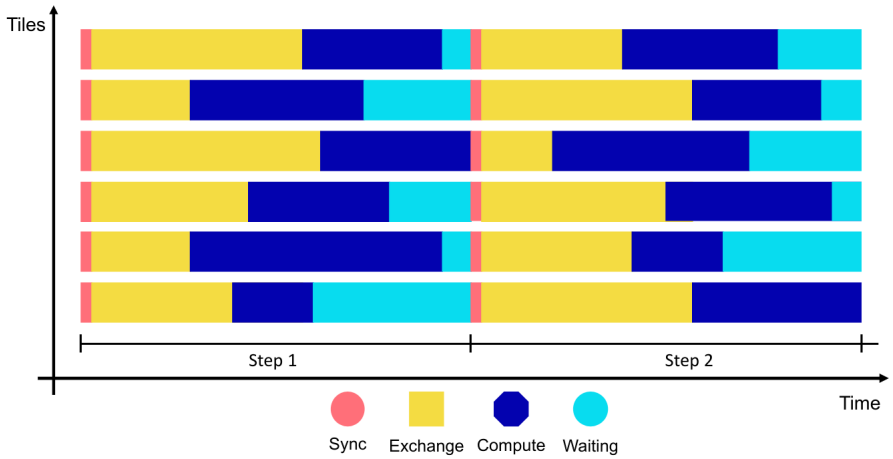


Figure 2.4: A conceptualization of BSP on the IPU [25]. The figure illustrates the phases of tiles along the timeline of an execution on the IPU.

During the computational phases, each tile operates using its core with six threads and its local in-processor memory [25]. When a tile is finished with its work, it enters the waiting/synchronization phase, which is regarded as the same phase in the documentation [25]. Eventually, when all tiles have reached the synchronization phase, the IPU enters an exchange phase where any necessary communication among tiles is performed. It is noteworthy to mention that waiting/synchronization and exchange are implicitly handled by the software frameworks, which makes it easier to develop parallel programs.

Execution Contexts

During execution of a program on the IPU, the six threads on each tile can execute in two types of hardware *execution contexts*, a supervisor context and six worker contexts [23]. The supervisor context is responsible for controlling the execution, synchronization, and exchanges, and initially during an execution, the supervisor context runs in all six threads [23]. Later, up to six independent computing processes can run as worker contexts, each occupying one thread. Whenever a worker context finishes its work, the thread is again occupied by the supervisor [23].

The execution contexts are handled implicitly by the software frameworks. However, for low-level assembly programming for the IPU, it could be useful to know that some instructions are supported in only the supervisor context or the worker context. Additionally, being aware of the execution contexts on the IPU can help in understanding how the BSP is incorporated on the IPU.

2.3.2 Scalability

The BSP paradigm is commonly used in scientific computing, and it can act as a foundation for scalable parallel performance [26]. One could argue that the IPU was built for scalability due to the limited 900 MB in-processor memory on one IPU, which is not sufficient for many modern computations, e.g., AI applications. Memory limitation is considered one of the main bottlenecks of modern deep learning models, which often require more memory than what one GPU accelerator (typically 8–32 GB) can accommodate [27]. Therefore, there is a need for scalability in the computer industry which is largely driven by AI performance [17].

The IPU-POD64 has a total of 57.6 GB of in-processor memory. If the need for memory is higher than the available in-processor memory, it is also possible to use the off-chip DRAM, which is referred to as streaming memory in the documentation. It is possible to connect systems with up to $2^{16} = 65,536$ IPUs, totaling 59.0 TB in-processor memory, which indicate that the supported scalability should not be a problem for multi-IPU programs.

2.3.3 Graph Programming

IPU programs are built around the idea of computational graphs that hold information about dependencies. An illustration of a computational graph is shown in Figure 2.5.

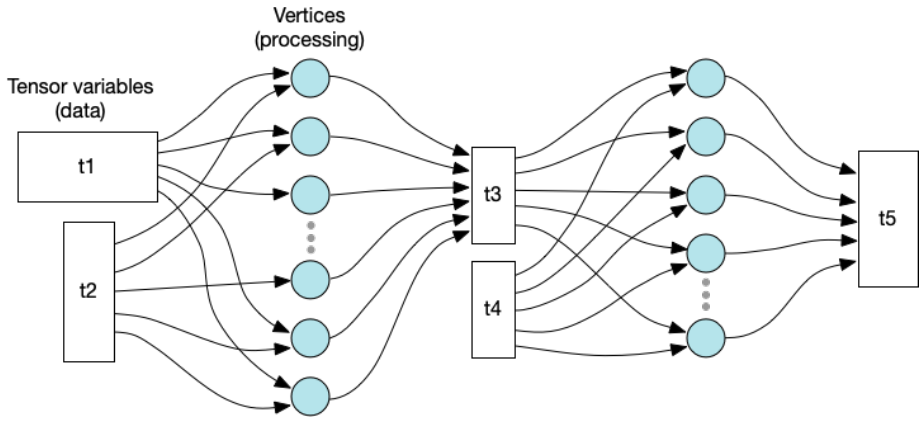


Figure 2.5: An illustration of a computational graph in an IPU program [25].

The most important data variable in IPU programs is the *tensor*, which is a multi-dimensional array-like object. Tensors can be stored across many tiles, and computations can operate on slices of these tensors. The computations in IPU programs are referred to as *vertices*. Since the IPU has a completely distributed memory model, both tensors and vertices must be assigned to tiles. Because of this, both data and computational dependencies quickly arise. The graph is a hierarchical organization of all tensors and vertices, including their dependencies throughout the execution.

The graph programming framework makes the IPU a flexible MIMD processor. Additionally, the graph implicitly handles both computational and data dependencies, which makes parallel programming feasible. This might have positive implications for general scientific computing on the IPU.

Chapter 3

IPU Programming

This chapter intends to serve as a practical guide to get started with Poplar programming in C++.

3.1 Overview

Developing programs for the IPU can be done in multiple ways. Figure 3.1 shows a wide overview of the frameworks that are supported on the IPU. It is possible to do Python programming with custom Python libraries for machine learning, most notably *TensorFlow* and *PyTorch*. This is highly relevant for those who want to use the IPU for ML applications. It is also possible to do C++ programming along with the *Poplar* framework [28, 29].

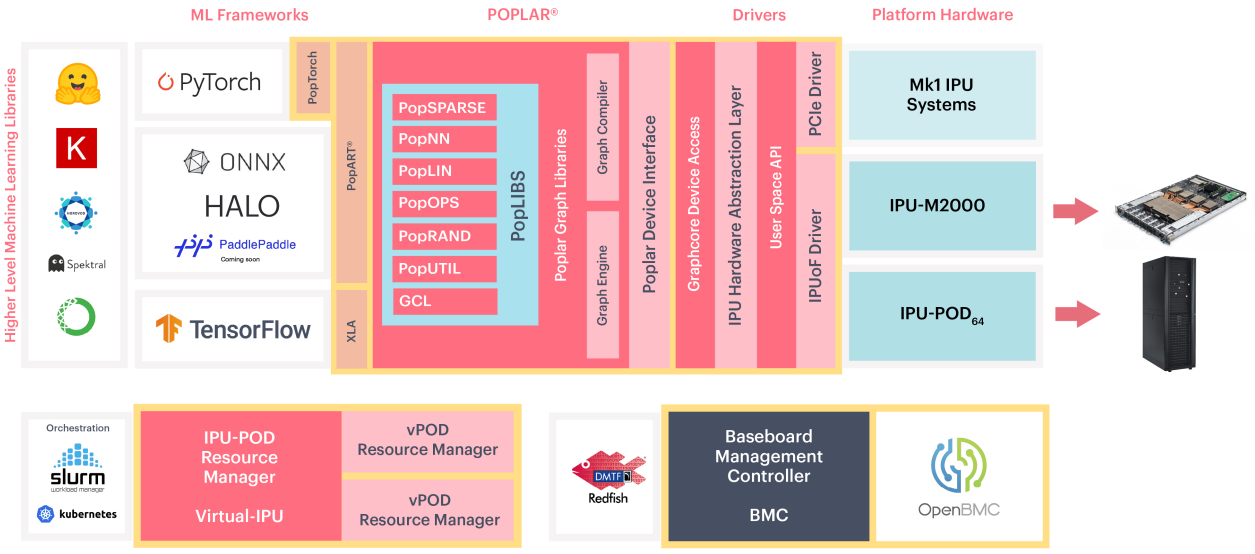


Figure 3.1: An overview ranging from hardware, drivers, and software frameworks for the IPU [28].

Poplar is a framework provided by Graphcore, which was co-designed along with the IPU [28]. It has a central role in most of the programming aspects regarding the IPU. As illustrated in Figure 3.1, Poplar builds the foundation on which the higher-level ML libraries are based on. At a low-level, the Poplar software development kit provides a C++ interface, which allows the programmer to develop fine-tuned IPU programs.

PopLibs is a set of libraries that provides application-level functions that can be used in Poplar programs (see Figure 3.1). PopLibs is useful for those who might want pre-built linear algebra functions, random number generators for tensors, sparse tensor operation functions, and more [29].

There are three resources that cover a wide variety of Poplar programming in C++:

- The *Poplar and PopLibs User Guide* [29] is an introductory user guide.
- The *Vertex Programming Guide* [23] covers more technical aspects of the IPU's architecture and introduces low-level assembly programming for the IPU.
- The *Poplar and PopLibs API Reference* [30] is a thorough documentation of the libraries within both Poplar and PopLibs.

This chapter intends to provide a practical guide to Poplar programming in C++.

A typical programming process is covered by implementing a benchmark and suggesting optimization steps that can increase the performance of the code.

3.2 Poplar

3.2.1 Glossary

The Poplar framework contains many technical terms. The most central terms are summarized Table 3.1.

Term	Description
Codelet	An implementation that declares a *vertex*.
Compute set	A set of *vertices* to be executed in parallel.
Device	A Poplar object referring to one or more IPUs used in the execution.
Engine	A Poplar object that controls *device* executions.
Graph	A Poplar object that is a representation of an execution and its data and computational dependencies.
Host	Referring to a CPU machine.
Program	A Poplar object that declares a *device* execution.
Target	A Poplar object that declares the execution target.
Tensor	A multi-dimensional array-like Poplar object.
Vertex	*Device* code that can be executed by a single thread.

Table 3.1: An alphabetically sorted list of technical terms that are used in Poplar programming.

The two terms *device* and *host* are often used when explicitly referring to either one or more IPUs or the connected CPU machine. This is relevant in the context of the heterogeneity that executing an IPU program involves.

The terms *compute set*, *device*, *engine*, *graph*, *program*[1], *target*, and *tensor* are some of the most central C++ objects in Poplar. Note that the term *device* is a Poplar object in addition to being a spoken/written reference to the one or more IPUs that are used in a program execution.

The terms *vertex* and *codelet* are closely related. *Vertices* are executable code that can run on one thread, or in parallel on many threads, on the IPU. *Codelets* are the actual implementations of vertices, being a C++ class with some implementation restrictions.

[1]The term *program* is under the Poplar namespace in C++. This term will be explicitly referred to as *Poplar programs* in order to distinguish it from the traditional *program* term.

3.2.2 Poplar Programming

Poplar provides fine-grained control over the program execution and handles some of the tedious aspects of parallel programming such as setting up data exchanges and synchronization. These features of Poplar highlight that it provides software productivity: an important attribute in the context of parallel programming. As stated in Chapter 1, software productivity can be considered one of four barriers in improving computational performance.

When getting started with Poplar programming, it is beneficial to familiarize oneself with all the central *Poplar objects*. As an overview, a general Poplar workflow involves these steps:

1. Define a *target* (choose to either use a physical IPU or an emulator).
2. Create the *device* object, which represents the target.
3. Create the *graph* object.
4. Build the program by adding data and computations to the graph:
   - Declare *tensors* and map them to tiles.
   - Declare *compute sets*, and assign *vertices* to them.
5. Create a vector of Poplar *program* objects.
6. Compile the graph and Poplar programs.
7. Create the *engine* object.
8. Use the engine to control input and output streams between host and device, and to control the Poplar program executions.

3.3 Implementation of a Benchmark

3.3.1 The STREAM Triad Benchmark

The standard benchmark for the measurement of computer memory bandwidth (STREAM) consists of programs that have been used for several decades as benchmarks for CPUs [31, 32]. The programs are written in standard C and Fortran. Table 3.2 shows the four different kernels that are included in the STREAM program.

Name	Kernel	FLOP/iter.	Read/iter.	Store/iter.
Copy	`a[i] = b[i]`	0	1	1
Scale	`a[i] = q*b[i]`	1	1	1
Sum	`a[i] = b[i] + c[i]`	1	2	1
Triad	`a[i] = b[i] + q*c[i]`	2	2	1

Table 3.2: The four benchmark kernels in the STREAM program, where `a`, `b`, and `c` are long arrays, and `q` is a scalar.

The STREAM benchmarks are common in high performance computing. Therefore, a customized benchmark from the STREAM suite can serve as a suitable introduction to IPU programming. The standard version of the benchmarks uses 64-bit elements and operands, but there are also 32-bit variants of the benchmarks. Only the *Triad* kernel was implemented, which is the last kernel shown in Table 3.2. Additionally, only single precision (32-bit) floats were used, because double precision is not supported on the IPU.

The STREAM benchmarks for the CPU have a rule for the length of the arrays involved. They should all be at least 4 times the size of the sum of the last-level caches used in the run, or 1 million elements, whichever is larger [32]. Since the IPU exhibits a different memory model, the situation becomes slightly different. There is no cache, and only distributed in-processor memory per tile. Therefore, the length of the arrays was set to be large enough to almost fill the in-processor memory on the IPU (900 MB).

3.3.2 Implementation

Device

In Poplar programming, the *device* object represents either a physical device or simulation that will execute the graph [29]. When defining the device, a *target* must be chosen. The target `poplar::TargetType::IPU` specifies that the device will represent physical IPU hardware. It is also possible to use `poplar::TargetType::IPU_MODEL`, which will result in an emulated CPU execution[2]. Listing 3.1 shows a function that can be used to create a device representing physical hardware.

[2]If *IPU_MODEL* is used, then the rest of the code can be treated as if the target was IPU, but the performance will probably be much worse. However, *IPU_MODEL* can be useful to familiarize oneself with IPU programming, or for testing.

```cpp
#include <Poplar/DeviceManager.hpp>

poplar::Device getIpuDevice(unsigned num_ipus) {
  auto manager = poplar::DeviceManager::createDeviceManager();
  auto device_ids = manager.getDevices(poplar::TargetType::IPU, num_ipus);

  for (auto &device_id: device_ids) { // Loop over accessible device IDs
    if (device_id.attach()) { // Check availability, i.e., not used by another
    ↪  program
      return std::move(device_id);
    }
  }

  throw std::runtime_error("No hardware device available.");
}

int main (int argc, char** argv) {
  unsigned num_ipus = 1;
  auto device = getIpuDevice(num_ipus); // Function on lines 3-14

  // Obtain useful information from device
  auto &target = device.getTarget();
  unsigned num_tiles = target.getNumTiles();
  unsigned num_worker_contexts = target.getNumWorkerContexts();
  double clock_frequency = target.getTileClockFrequency();
  std::uint64_t total_memory = target.getMemoryBytes();
}
```

Listing 3.1: A C++ code that creates a device: an object which represents a combined entity of one or more IPUs.

The function, `getIpuDevice()` in Listing 3.1 can be used to create a device with a desired number of IPUs. The `getDevices()` method on line 5 returns a list of devices that fulfills the criteria (number of IPUs and target type) and can potentially be an empty list. In the listing, a loop iterates over the possible device IDs, and checks the availability by using the `attach()` method. If this method returns `true`, then the device is available and can be used. Table 3.3 shows the possible device IDs in an IPU-POD64.

Number of IPUs	Device IDs
1	0–63
2	64–95
4	96–111
8	112–119
16	120–123
32	124–125
64	126

Table 3.3: Overview of device IDs for the IPU-POD64.

Tensors

A *tensor* is a common term from linear algebra. For instance, scalars, vectors and matrices are tensors, but so are also higher dimensional representations of numbers. In Poplar, a tensor is an object which can hold a collection of numbers. In this thesis, the term *tensor* will only refer to the Poplar object and not the mathematical term.

Tensors are arguably one of the most important variables in IPU programming. They are similar to *numpy.ndarrays* from the Python library *NumPy* and contain a wide variety of methods which use NumPy-like syntax. Within Poplar, Graphcore provides a framework called *PopLibs* which supports many application-level functions such as linear algebra operations, neural network functions, operations on tensors and on sparse tensors, and operations for populating tensors with random numbers.

Vertex

A *vertex* is executable code that can run in a *worker context* (see section 2.3.1) on a single thread on the IPU. The vertex implementation is referred to as a *codelet*, which is written as a C++ class with a special format in a separate file.

Both tensors and vertices are distributed on a per-tile basis. However, it is possible to assign multiple vertices to each tile. Then, the supervisor context on the tile will distribute the vertices to threads as worker contexts. In order to fully utilize the IPU, the workload should be split into six vertices per tile, because there can be six worker contexts per tile.

In Poplar, the codelet is implemented in a separate file, because it is compiled by a dedicated compiler, namely *popc*. Listing 3.2 shows a codelet that computes the STREAM Triad kernel.

```
1  #include <poplar/Vertex.hpp>
2
3  using namespace poplar;
4
5  class TriadVertex : public Vertex {
6  public:
7    TriadVertex();
8
9    Output<Vector<float>> a;
10   Input<Vector<float>> b;
11   Input<Vector<float>> c;
12   const float q;
13
14   bool compute () {
15     for (std::size_t i = 0; i < a.size(); ++i)
16       a[i] = b[i] + q*c[i];
17
18     return true;
19   }
20 };
```

Listing 3.2: A simple C++ file which contains one codelet, the class *TriadVertex*, which computes the Triad kernel.

The codelet, *TriadVertex* in Listing 3.2 highlights some syntax requirements. The codelet, which is a class, must inherit from the base class `Vertex`. Second, the codelet must include a `compute` function, which takes no arguments, and return `true` [3].

There are three possible *vertex-variables* that can connect tensors to the vertex: `Input` (read), `Output` (write), and `InOut` (read and write). These types can be used in combination with `Vector` (1D) and `VectorList` (2D). For instance, `Vector<Input<Vector<T>>>` becomes a 2D variable. Poplar will implicitly set up data streams for the vertex-variables. However, one must ensure that the dimension of a vertex-variable matches the dimension of the tensor when assigning the vertex. The tensor-connecting vertex types can at most be two-dimensional. This means that higher-dimensional tensors must be flattened to 1D or 2D in order to be connected to a vertex. In addition to the tensor-connecting types, additional C++ type parameters can also be assigned to the vertex, such as `float q` on line 12 in Listing 3.2.

Graph

The *graph* is a representation of data and computational dependencies throughout the entire IPU execution. It is one of the most central C++ objects provided

[3] When the vertex returns *true*, the worker context is finished and the thread is occupied by the supervisor context as the thread enters a waiting/synchronization phase in the context of BSP.

in Poplar, and it is responsible for putting together many of the concepts that have been introduced so far. Figure 3.2 shows a simple conceptualization of the graph for the Triad implementation.

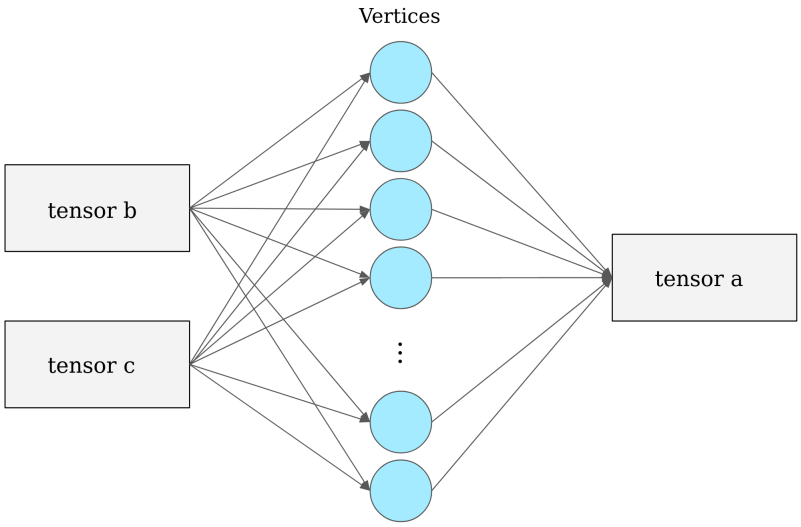


Figure 3.2: An illustration of a graph representation of the Triad benchmark. There are many vertices, each connected to read from slices of `b` and `c`, and to write to `a`. The vertices represent computations, and the arrows represent dependencies.

In Figure 3.2, the squares represent tensors, the blue circles represent vertices, and the arrows represent dependencies. Moving on, `b` and `c` are predetermined tensors that have no dependencies (there are no arrows pointing into them). The resulting tensor, `a` depends on the computations on all tiles, which in turn depend on the data in `b` and `c`. In parallel programming, the dependencies between variables and computations can quickly become complex (see for instance Figure 2.5), but these dependencies are implicitly handled by the graph.

In Poplar, creating and completing the graph object includes numerous steps. This includes declaring tensors, setting tile mappings for these, defining compute sets, adding vertices to these compute sets, and declaring data streams. Listing 3.3 shows the parts of the implementation of the Triad benchmark that involves the graph.

```
// Create graph object
poplar::Graph graph{target};
graph.addCodelets("codelets.gp"); // pre-compiled codelets.cpp

// Declare Tensors, device variables
auto a = graph.addVariable(poplar::FLOAT, {num_tiles, num_worker_contexts,
↪  size_per_worker}, "a");
auto b = graph.addVariable(poplar::FLOAT, {num_tiles, num_worker_contexts,
↪  size_per_worker}, "b");
auto c = graph.addVariable(poplar::FLOAT, {num_tiles, num_worker_contexts,
↪  size_per_worker}, "c");

// Perform tile mapping of a
for (std::size_t i = 0; i < num_tiles; ++i) {
  graph.setTileMapping(a[i], i); // 2D slices
}

// Apply same tile mapping to b and c
graph.setTileMapping(b, graph.getTileMapping(a));
graph.setTileMapping(c, graph.getTileMapping(a));

// Create compute set object
auto compute_set = graph.addComputeSet();

// Assign vertices to this compute set
for (std::size_t i = 0; i < num_tiles; ++i) {
  // There will be num_worker_contexts vertices per tile (not a requirement)
  for (std::size_t j = 0; j < num_worker_contexts; ++j) {
    // Assign vertices to tiles
    auto v = graph.addVertex(compute_set, "TriadVertex");
    graph.connect(v["a"], a[i][j]); // 1D slices: dimensionality must match with
    ↪  the vertex type
    graph.connect(v["b"], b[i][j]);
    graph.connect(v["c"], c[i][j]);
    graph.setInitialValue(v["q"], q);
    graph.setTileMapping(v, i); // Tile mapping of vertex
  }
}

// Define data streams
auto device_to_host_a = graph.addDeviceToHostFIFO("a_stream", poplar::FLOAT,
↪  a.numElements());
auto host_to_device_b = graph.addHostToDeviceFIFO("b_stream", poplar::FLOAT,
↪  b.numElements());
auto host_to_device_c = graph.addHostToDeviceFIFO("c_stream", poplar::FLOAT,
↪  c.numElements());
```

Listing 3.3: A code snippet that shows a workflow with the graph object in Poplar. The dependencies imposed by the graph in this code snippet is illustrated in Figure 3.2.

Listing 3.3 declares three 3D tensors, `a`, `b`, and `c`. The first dimension represents the number of tiles, the second dimension represents the number of

worker contexts, and the third dimension represents the number of elements that each worker context computes. This approach imposes the disadvantage of the total size needing to be a multiple of 8832 (the number of threads in an IPU). However, the workload is guaranteed to be balanced, which is an important aspect in parallel programming. This is especially true for the BSP paradigm, where the execution time is bound by the slowest worker.

The workflow of the graph in Listing 3.3 can be summarized in these steps:

- Lines 2–3 declare the graph and add the pre-compiled codelet.
- Lines 6–8 add tensors to the graph. These must be mapped to tiles, which is done for one of the tensors on lines 11–13. Further, the tile mapping is re-applied to the other two tensors on lines 16–17.
- Lines 20–34 add vertices to the graph. Every vertex must be a part of a *compute set*, which is a grouping of vertices that will be executed in parallel. Further, like tensors, vertices must also be mapped to tiles.
- Lines 37–39 declares three data streams that will later be used to send data between host variables and the tensors on the device.

Poplar Programs

A Poplar program is the base class for creating control programs that define how vertices will be executed [29]. It is possible to combine Poplar programs by using them as building blocks in more complex programs. There are several types of Poplar programs, and some of the most common ones are

- `Execute` is used to execute a compute set,
- `Repeat` is used to repeat a Poplar program for a specified number of iterations,
- `Sequence` is used to combine multiple Poplar programs together,
- `Copy` is used to copy data, between two tensors or between a tensor and a data stream, and
- `If` is used to conditionally execute a Poplar program.

There are several other types of Poplar programs which can be found in the API reference [30]. Listing 3.4 contains a code snippet which shows an example of a workflow with Poplar programs.

```
// Program 0: data stream to device
auto host_to_device = poplar::program::Sequence({
    poplar::program::Copy(host_to_device_b, b),
    poplar::program::Copy(host_to_device_c, c)
});

// Program 1: executions of vertex
auto inner_loop = poplar::program::Repeat(
    num_iterations,
    poplar::program::Execute(compute_set)
);

// Program 2: data stream from device
auto device_to_host = poplar::program::Copy(a, device_to_host_a);

// Combine programs in a vector
std::vector<poplar::program::Program> programs{
    host_to_device,
    inner_loop,
    device_to_host
};
```

Listing 3.4: An example implementation which sets up three Poplar programs and uses various program commands. The three data streams `host_to_device_b`, `host_to_device_c`, and `device_to_host_a`, and the compute set `compute_set` are parts of the graph, declared in Listing 3.3.

Listing 3.4 shows the implementation that declares the Poplar programs used for the Triad benchmark. There are three Poplar programs, which in the end are combined to a standard C++ vector. The first program is used to populate `b` and `c` by copying data from data streams. The second program repeatedly executes the compute set, which performs the Triad benchmark parallelized. The third program is used to copy the data from the tensor `a` to a data stream, which can be used to obtain the results, and to control that it is correct.

Engine

The *engine* object is the run-time component of Poplar [29]. It takes one parameter: the graph and control programs compiled into an executable Poplar object. Listing 3.5 contains a code snippet that shows the workflow with the engine object.

```
1  // Compile graph and programs, declare engine
2  auto exe = poplar::compileGraph(graph, programs);
3  poplar::Engine engine(std::move(exe));
4
5  // Connect the graph data streams to the host memory addresses
6  engine.connectStream("a_stream", &host_a[0], &host_a[host_a.size()]);
7  engine.connectStream("b_stream", &host_b[0], &host_b[host_b.size()]);
8  engine.connectStream("c_stream", &host_c[0], &host_c[host_c.size()]);
9  engine.load(device);
10
11 // Executions of programs
12 engine.run(0); // Program 0: copy host variables "b" and "c" to device
13 engine.run(1); // Program 1: repeatedly execute the compute set
14 engine.run(2); // Program 2: copy device variable "a" to host
```

Listing 3.5: An example of the set up and usage of the engine object in Poplar.

Listing 3.5 shows the engine workflow used in the Triad benchmark, and can be summarized in these steps:

- Lines 2–3 compile the graph and programs, and then use this executable to declare the engine.
- In lines 6–8, the data streams that were first declared in the graph (lines 37–39 in Listing 3.3), then included in Poplar programs (lines 2–5 and 14 in Listing 3.4), are now connected to host memory addresses by using the engine's `connectStream()` method.
- Line 9 loads the compiled program onto the device.
- Lines 12–14 are used to execute the three Poplar programs that are contained in the `programs` vector.

3.3.3 Optimization

Theoretical Peak Performance

The peak performance of this benchmark can be estimated by studying the possible memory and compute instructions. The widest memory operations on a single thread are 64-bit store and 128-bit load (only from the interleaved memory region). It is also possible to perform simultaneous load and store, which is ideal for the Triad benchmark as it needs to load from both `b` and `c`, and store to `a`. Theoretically, the instruction `ld2xst64pace` can load 64-bits from both `b` and `c` and store 64-bit to `a` during one clock cycle, facilitating two 32-bit Triad kernels per tile per clock cycle.

The aux pipeline supports multiply-accumulate instructions, which performs one multiplication and one addition in the same instruction, i.e., `b[i] + q*c[i]`. The

instruction `f32v2axpy` performs two 32-bit versions of this kernel, vectorized, during one clock cycle. Therefore, this compute instruction facilitates two 32-bit Triad kernels per tile per clock cycle.

Two ideal instructions for the Triad benchmark have been identified. They can execute synchronously in the two execution pipelines, main and aux. The six worker contexts operate sequentially. Hence, there can be at most 1,472 executing threads per clock cycle. On every tile, there are 4 floating point operations (FLOPS), 16 bytes loaded, and 8 bytes stored per clock cycle. By assuming a clock frequency of 1,330 MHz, the theoretical peak performance (both of this benchmark and the IPU in general) is shown in Table 3.4.

Processor	TFLOPS	Load BW	Store BW	Total BW
MK2 IPU	7.83	31.32 TB/s	15.66 TB/s	46.98 TB/s

Table 3.4: The theoretical best computational throughput (FLOPS) and memory bandwidth (BW) on a single IPU. A clock frequency of 1.330 GHz is assumed.

Preliminary Code

The Triad benchmark has been set up to exhibit a balanced workload on all the tiles and threads of the IPU. Therefore, when optimizing the code, the focus should lie on the codelets. In more complex programs, it could be beneficial to optimize the setup of the graph and Poplar programs as well. However, for this benchmark, the performance relies heavily on the codelet.

The Triad benchmark lacks *communication phases*, which arguably is one of the most important aspects of parallel programming. Communication is covered in later chapters.

Analysing Assembly Code

The compiler *popc* can be used to write the assembly code of the vertex into a file. Listing 3.6 shows the assembly code that corresponds to the loop of *TriadVertex* (lines 15–16 in Listing 3.2).

```
1  .LBB0_7:                                # =>This Inner Loop Header: Depth=1
2    ld32 $a3, $m9, $m15, 1
3    ld32step $a2, $m15, $m9+=, 2
4    ld32 $a5, $m8, $m15, 1
5    ld32step $a4, $m15, $m8+=, 2
6    f32v2mul $a4:5, $a0:1, $a4:5
7    f32v2add $a2:3, $a2:3, $a4:5
8    st32 $a3, $m7, $m15, 1
9    st32step $a2, $m15, $m7+=, 2
10   brnzdec $m6, .LBB0_7
```

Listing 3.6: A snippet of the assembly code that corresponds to the loop of *TriadVertex* (from Listing 3.2).

The assembly code in Listing 3.6 can be broken down. Lines 2–10 contain one instruction per line, where the first word is the instruction name. The following words are parameters, e.g., *a0-a15* and *m0-m15* are registers that can hold 32 bits. The 9 instructions in the loop perform two 32-bit Triad kernels:

- Lines 2–5 load four 32-bit elements from `b` and `c`.
- Lines 6–7 perform the multiplication and addition, `b[i] + q*c[i]`, vectorized for two kernels.
- Lines 8–9 store the results from the two kernels to `a`.
- Line 10 contains a loop overhead instruction that points to the beginning of the loop, if it is not finished.

Some of the weaknesses of *TriadVertex* is that all memory instructions are only 32-bit wide, when in theory, it is possible to use up to 128-bit loads, and 64-bit stores. Another weakness is that the two pipelines, main and aux, do not execute synchronously.

Implementing a Codelet with Memory Constraints

As a starting point, the simplest codelet introduced in Listing 3.2 is used as a template. Then, two constraints on how the tensors are stored in the in-processor memory were applied. This could potentially help the compiler in applying wider instructions and possibly achieve vectorization. The new codelet is shown in Listing 3.7.

```
1  class [[ poplar::constraint("elem(*b)!=elem(*c)") ]]
2    TriadVertexMemory : public Vertex {
3  public:
4    TriadVertexMemory();
5
6    Output<Vector<float, VectorLayout::SPAN, 8, false>> a;
7    Input<Vector<float, VectorLayout::SPAN, 8, true>> b;
8    Input<Vector<float, VectorLayout::SPAN, 8, true>> c;
9    const float q;
10
11   bool compute () {
12     for (std::size_t i = 0; i < a.size(); ++i)
13       a[i] = b[i] + q*c[i];
14
15     return true;
16   }
17 };
```

Listing 3.7: A codelet that introduces constraints on how the variables are stored in memory. The first constraint is on line 1, where the two variables, `b` and `c` should not be stored in the same memory element. Lines 6–8 employ additional constraints on the three vector variables.

The codelet in Listing 3.7 applies a memory constraint in an attempt to achieve wider memory instructions. On the first line, the two tensors, `b` and `c` are constrained to be stored in different memory elements. This is a requirement in order to concurrently load 64-bits from both tensors simultaneously. It is also possible to apply a similar constraint to specify that two variables should be stored in different memory *regions* by writing `"region(*a)!=region(*b)"`.

The second type of memory constraint can be found in the contents of the `Vector` types, on lines 6–8. In addition to specifying that it should be a vector of floats, it is possible to provide three additional arguments:

- The first parameter specifies how the variable is stored in memory. For instance, `VectorLayout::SPAN` stores the pointer address to the beginning of the vector and the size of it.
- The second parameter specifies the minimum alignment, in bytes. It was set to 8 in order to allow for 8 bytes wide memory instructions.
- The third parameter controls whether to store the variable in interleaved memory. This was set to `true` for `b` and `c`, because it is a requirement for concurrently loading 64-bit from both variables.

The memory constraints that were applied do not guarantee an increase in performance. However, satisfying these requirements could potentially help the compiler in applying wider memory instructions. Listing 3.8 shows the assembly that corresponds to the loop of *TriadVertexMemory*.

```
1  .LBB1_7:                                # =>This Inner Loop Header: Depth=1
2    ld64step $a2:3, $m15, $m9+=, 1
3    ld64step $a4:5, $m15, $m8+=, 1
4    f32v2mul $a4:5, $a0:1, $a4:5
5    f32v2add $a2:3, $a2:3, $a4:5
6    st64step $a2:3, $m15, $m7+=, 1
7    brnzdec $m6, .LBB1_7
```

Listing 3.8: The assembly code corresponding to the loop of *TriadVertexMemory*.

Listing 3.8 shows that every memory instruction is 64-bit wide, instead of being sequences of two 32-bit instructions. This is a consequence of specifying constraints on how the variables are stored in memory.

The constraints in *TriadVertexMemory* were applied to help the compiler in achieving 128-bit wide load. This memory operation requires that `b` and `c` should both be stored in interleaved memory, and in different memory elements. However, this was not applied by the compiler, since they are loaded sequentially in two 64-bit loads on lines 2–3. Another optimization that could have been applied by the compiler is co-issues. This means that the main and aux pipelines execute instructions synchronously. If this had occurred, the instructions would appear inside curly brackets in the assembly code, which is not the case. In Listing 3.8, on could for instance have loaded `b[i]` and `b[i+1]` whilst performing the multiplication `q*c[i]` and `q*c[i+1]`. If the compiler had performed either of the two optimizations mentioned in this paragraph, the number of clock cycles could have been 5 instead of 6. This indicates that the compiler is subject to improvement.

Implementing a Codelet with Inline Assembly

There are more optimization steps that can be applied in order to cut down the number of instructions per iteration. First, in order to utilize both execution pipelines simultaneously, it is beneficial to co-issue instructions whenever possible. Second, loop overhead instructions can be avoided. There is a so-called `rpt` (repeat) loop which does not contain loop overhead. Ideally, co-issuing the two instructions mentioned at the beginning of section 3.3.3 inside a repeat loop exploits the theoretical best performance that the IPU can offer. This can be done by implementing inline assembly code directly in the codelet, which is demonstrated in Listing 3.9.

```
class [[ poplar::constraint("elem(*b)!=elem(*c)") ]]
  TriadVertexAssembly : public Vertex {
public:
  TriadVertexAssembly();

  Output<Vector<float, VectorLayout::SPAN, 8, false>> a;
  Input<Vector<float, VectorLayout::SPAN, 8, true>> b;
  Input<Vector<float, VectorLayout::SPAN, 8, true>> c;
  const float q;

  bool compute () {
    const std::size_t iter = a.size()/2 - 2;
    auto packed_ptr = __builtin_ipu_tapack(&c[0], &b[0], &a[0]);
    __asm__ volatile(
      R"(
      {
        ld2x64pace $a0:1, $a2:3, %[ptr]+=, $m15, 0
        uput $TAS, %[q]
      }
      {
        ld2x64pace $a0:1, $a2:3, %[ptr]+=, $m15, 0
        f32v2axpy $a4:5, $a0:1, $a2:3
      }
      {
        ld2x64pace $a0:1, $a2:3, %[ptr]+=, $m15, 0
        f32v2axpy $a4:5, $a0:1, $a2:3
      }
      nop
      rpt %[iter], (2f - 1f)/8 - 1
      1:
      {
        ld2xst64pace $a0:3, $a4:5, %[ptr]+=, $m15, 0
        f32v2axpy $a4:5, $a0:1, $a2:3
      }
      2:
      {
        st64pace $a4:5, %[ptr]+=, $m15, 0
        f32v2gina $a4:5, $a14:15, 0
      }
      st64pace $a4:5, %[ptr]+=, $m15, 0
      )"
      :
      : [ptr] "r"(packed_ptr), [iter] "r"(iter), [q] "r"(q)
      : "$a0", "$a1", "$a2", "$a3", "$a4", "$a5", "memory"
    );
    return true;
  }
};
```

Listing 3.9: The codelet *TriadVertexAssembly*, which solves the Triad benchmark, and introduces inline assembly code. The highlighted lines (29–35) contain the `rpt` instruction and a loop body with one co-issue of instructions.

Listing 3.9 introduces a few new concepts of vertices. The assembly code is the raw string on lines 15–41. The memory constraints are now required, because 64-bits will be loaded from both `b` and `c` (in total: 128-bit load). The assembly code contains a *repeat loop*. The loop body is the co-issue on lines 32–33, which performs two Triad kernels in one clock cycle.

The instructions before the loop are required due to several reasons:

- The co-issue on lines 17–18 loads the first two floats from `b` and `c`. Additionally, the scalar `q` are loaded into the `TAS` internal state element, which is where the `f32v2axpy` instruction expects it to be.
- The co-issue on lines 21–22 loads two subsequent values from `b` and `c`, and performs the first `f32v2axpy` instruction. The results end up in the so-called *accumulator state*.
- The co-issue on lines 25–26 the two next subsequent values from `b` and `c` and performs the second `f32v2axpy` instruction. This time, the previous results are stored in registers, which is where the results must be before they can be stored into memory.

In the loop, the current results from the `f32v2axpy` instruction are stored in the accumulator state. The previous results are stored in registers, and the results before those are stored to memory. The compute instruction is two steps ahead of storing the results to memory. This effectively means that the last and second-to-last results must be stored after the loop:

- The co-issue on lines 37–38 stores the second-to-last result to memory and fetches the last results from the accumulator state.
- The last instruction on line 40 stores the last result to memory.

The number of loop iterations is set to `a.size()/2 - 2`. It is half the number of elements due to vectorization, and additionally subtracted by two, because two computations are done prior to the loop.

3.3.4 Performance

Table 3.5 shows the results of the three vertices from the STREAM Triad benchmark.

Vertex Name	Throughput [TFLOPS]	Minimal Bandwidth	% of Peak Performance	Relative Performance
TriadVertex	0.87	5.19 TB/s	11.05	1.00
TriadVertexMemory	1.30	7.79 TB/s	16.57	1.50
TriadVertexAssembly	7.76	46.55 TB/s	99.07	8.96

Table 3.5: Performance of three vertices that performed the Triad benchmark on the MK2 IPU at 1,330 MHz. The *time* was the average execution time of 100,000 executions of the benchmark. Each of the vectors consisted of 66.24 million single precision floats.

The results in Table 3.5 are averaged from 100,000 IPU executions. The threads solved 7500 elements each, which means that the total length of the tensors was 66.24 million elements. The memory usage of the three tensors was 794.9 MB, which is 84.5% of the 900 MB available memory. There are also some memory gaps due to the constraint, as well as vertices and other programs. Therefore, the size of the tensors approaches the maximum on-chip memory capacity.

The loops of `TriadVertex`, `TriadVertexMemory`, and `TriadVertexAssembly` used 9, 6, and 1 clock cycles, respectively. Therefore, the expected performance increases of the optimized vertices compared to the first vertex are 1.5 times, and 9 times. The results showed relative speed-ups of 1.50 times, and 8.96 times which is very consistent with expectations. This consistency suggests that the assembly code can provide a good prediction to the performance.

A programmer who is new to Poplar programming should at least be able to reach the performance of *TriadVertexMemory*. This is because the *vector* variables in vertices should always specify the additional parameters (vector layout, minimum alignment and whether to be stored in interleaved memory). *TriadVertexMemory* only achieved 16.57% of the theoretical peak performance. This result suggests that the compiler is not able to optimize the vertex to reach anywhere near peak performance by a standard Poplar implementation of this benchmark. The demonstration of the performance of *TriadVertexAssembly* also highlights the current importance of writing inline assembly code for HPC programs on the IPU, because it can lead to a significant performance increase. However, the IPU became available relatively recently, and the software, including the compiler, is under constant development. Therefore, it is reasonable to assume that the compiler could be able to increase the performance of *TriadVertexMemory* in future releases.

The *TriadVertexAssembly* achieved a performance of 7.76 TFLOPS on a single processor and reached 99.07% of the theoretical peak performance. This performance is unrealistically high in more complex programs due to two reasons. It is unfeasible to many developers to even resort to assembly programming, and the assembly programming itself could become increasingly harder to implement

than the demonstration in this chapter, because most applications are considerably more advanced than the Triad benchmark. Therefore, similar performances should not be expected in more complex programs. This result should rather be considered as an upper boundary for practically achievable performance.

3.4 Programming Advice

3.4.1 Environment Variables

There are several environment variables which can be helpful in Poplar programming. These must be set in the terminal shell before executing the program. Full information about environment variables can be found in the documentation [29]. Two of them should be emphasized:

- The `POPLAR_LOG_LEVEL` variable can be set to the values `OFF`, `WARN`, `ERR`, `TRACE`, `DEBUG`, or `INFO`. The value of this variable affects the verbosity of terminal outputs during the execution.
- The `POPLAR_ENGINE_OPTIONS` variable can be used to control profiling and other settings of the engine object. It must be set to a JSON-like object, e.g., `{"autoReport.all":"true","autoReport.directory":"./report"}` generates a profiling report in the current directory. For a full list of the options that this variable can take, see the *Poplar and PopLibs API Reference* [30].

3.4.2 Profiling

To study metrics of a program such as the BSP-phases throughout execution or the memory usage per tile, there is a GUI profiling tool called *PopVision Graph Analyzer* (PopVision). As mentioned in section 3.4.1, the environment variable `POPLAR_ENGINE_OPTIONS` can enable profiling for an execution. A profiling report is a large folder of detailed execution data. Whenever a program is executed with profiling enabled, the performance of the program will be affected. Therefore, one must typically run the program both with and without profiling.

A profiling report can be studied in PopVision. A graphical user interface provides a comprehensive overview of the execution:

- The execution trace shows a timeline of stages during the execution, presented in a "BSP-manner".
- The memory report, which shows detailed metrics of the memory usage.

3.4.3 General Workflow in Poplar

Consider an application that is to be implemented on the IPU. Before creating the two C++ files, the main program and the codelets, one should fetch a pen and paper. Poplar programming is designed around the idea of a computational graph. Therefore, it is beneficial to draw the computations as a graph. This

should involve writing down the variables, which presumably should be tensors, and the computations as vertices. Dependencies should also be included in the planning.

Once the graph is imagined, the main program should focus on constructing the Poplar graph. To the graph, tensors should be added and mapped to tiles. Furthermore, one or more compute sets should also be added to the graph. The compute sets should in turn include a high number of vertices, arguably one vertex for every thread.

Lastly, it is important to write optimized codelets. When compiled, they become vertices, which define the computations to be executed by a single thread. This constitutes the computational phases, where much of the execution time is spent. One should always specify the additional parameters that can be given to the *Vectors* and *VectorList* variables. It is especially important to choose suitable *memory alignment* and *memory region*. For instance, if the in-processor memory architecture is understood well, the Triad benchmark should undoubtedly store the two input variables in the interleaved region, and the output in the non-interleaved region.

Chapter 4

The 2D Heat Equation by Finite Differences

4.1 Motivation

Solving PDEs numerically is a common problem encountered in scientific computing. One of the main approaches for such computations is by using stencil computations, which are based on finite difference approximation of derivatives. The motivation for solving these types of problems on the IPU is that they could serve as relevant benchmarks for general purpose scientific computing problems. This chapter covers how the heat equation was discretized by finite differences in two dimensions and how these computations were implemented on the IPU. In the next chapter, this work will be extended to three dimensions.

The overarching goal in this chapter is to study how good performance 2D stencil computations can achieve on the IPU. The computational problem was implemented in C++ with Poplar, which highlights the process of developing a typical scientific application for this specialized processor. In Poplar, parallelism is set up by distributing data and instructions by a graph-based approach. This type of development could be interesting for readers looking to solve other types of computing problems on the IPU, by taking inspiration from the development process presented in this chapter.

The PDE that was implemented is the 2D isotropic diffusion equation (also known as the heat equation, since undirected flow of heat is a form of diffusion). This is a prototypic PDE problem that describes how a property, e.g., temperature, evolves in substances.

4.2 Background

This section will explain the underlying mathematics behind the diffusion equation. This is a general equation that can be used to model changes in physical properties, such as heat. In order to numerically solve the PDE with a stencil-based approach, the equation is discretized by a finite difference scheme. The diffusion equation is given by

$$\frac{\partial u}{\partial t} = \kappa \nabla^2 u, \tag{4.1}$$

where κ is a diffusive constant and $u = u(x,t)$ is a property, e.g., temperature, at the time t and point x in space, using Cartesian geometry. The diffusion equation can be discretized by approximating the derivatives by finite differences. The first and second-order derivatives are approximated as

$$\frac{\mathrm{d}u}{\mathrm{d}x} \approx \frac{u(x+h) - u(x)}{h}, \text{ and} \tag{4.2}$$

$$\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} \approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2}, \tag{4.3}$$

where h is the spatial step size. The exact solutions for the derivatives are given in the limit where $h \to 0$, and the finite difference approximations are given when h has a finite value. The discretization of the derivatives uses the *forward difference in time*, and the *central difference in space*, which is an explicit method for solving the heat equation numerically. Equation (4.2) is also known as the *forward Euler method*, which approximates the first order derivative.

When employing the heat equation numerically, u is represented by a finite mesh. A special case of the heat equation was applied, namely the *isotropic* heat equation. This corresponds to spatial invariance, which means that the equation exhibits the same dynamics in all dimensions. Numerically, this invariance is incorporated by separating the points with a uniform distance h. For a 2D mesh, the discretized heat equation becomes

$$u_{ij}^{t+\delta} = (1 - 4\alpha)\, u_{ij}^{t} + \alpha \left(u_{i+1,j}^{t} + u_{i-1,j}^{t} + u_{i,j+1}^{t} + u_{i,j-1}^{t} \right). \tag{4.4}$$

Here, $\alpha = \kappa\delta/h^2$, and δ is the time step. Equation (4.4) was employed as a stencil-based algorithm. The right-hand side represents a 5-point stencil that can be slid over the inner points of a regular 2D mesh, by looping over the indices i and j. However, the algorithm is only numerically stable and convergent when $\alpha \leq 0.5$.

Note that the stencils require four neighboring points in space and are therefore undefined at the boundaries. The *Dirichlet boundary condition* was used, which means the inner points of the 2D mesh were updated, whereas the boundary elements were kept constant.

The stencil-based numerical algorithm provides an approximated solution to the heat equation. The error depends on the spatial and temporal step sizes, h and δ. To achieve higher accuracy for a given problem, one could use a smaller step size between the points, or smaller time steps. Ultimately, more accurate simulations come at the cost of requiring more computations, which in turn could benefit from higher computational performance.

4.3 Methods

The discretized 2D heat equation was implemented for the IPU and CPU. The two processors solved the same computational problem, which was a 2D mesh of 8000×8000 single precision elements, for 1000 time steps. The IPU program was implemented in C++, along with the Poplar framework. The CPU program was implemented in the programming language C, along with the multi-threading framework OpenMP.

4.3.1 IPU Implementation

Work Division

Single-IPU executions were performed in a highly parallel fashion. The 2D mesh was represented by two tensors, `a` and `b`, which were distributed among the tiles. The computational workload was further divided to utilize all threads. This corresponds to assigning 6 vertices to each of the 1,472 tiles on a MK2 IPU.

The first consideration when solving the heat equation parallelized is how to partition and distribute the mesh. For this problem, an algorithm to find the number of partitions along both height and width was implemented. The goal of this algorithm was to minimize the total communication volume, also referred to as the halo region. This algorithm is shown in Listing 4.1.

```
// height = 8000, width = 8000, and tile_count = 1472;
float smallest_halo_region = std::numeric_limits<float>::infinity();
std::vector<std::size_t> partitions(2);

// Try all unique combinations where nh*nw = tile_count
for (std::size_t nh = 1; nh <= tile_count; ++nh) {
  if ((tile_count % nh) == 0) { // then nh (number along height) is a factor
    std::size_t nw = tile_count / nw; // and nw (number along width) must be the
    ↪  other factor
    std::size_t slice_height = (height - 2)/nh;
    std::size_t slice_width = (width - 2)/nw;

    // Circumference of a data partition
    std::size_t halo_region = 2.0*(slice_height + slice_width);
    if (halo_region < smallest_halo_region) {
      smallest_halo_region = halo_region;
      partitions = {nh, nw};
    }
  }
}
```

Listing 4.1: An algorithm to find the optimal number of partitions along the height and width directions. The algorithm was constrained to utilize all available tiles, and to choose the set that minimizes the communication volume, which is also referred to as the halo region.

The partitioning algorithm shown in Listing 4.1 was constrained to utilize all tiles. Therefore, the number of partitions along height and width can only take a finite set of values[1]. Furthermore, the split that minimizes the necessary communication volume was chosen.

After the partitioning was found, the tensors were divided into slices and distributed among the tiles. If the height and width were not be divisible by the number of partitions in each direction, some tiles would receive data partitions with at most one element wider and higher than the smallest partitions. For the 8000×8000 mesh, the workload to be divided was the inner mesh of 7998×7998 elements. The partitioning algorithm resulted in 46 partitions along the height and 32 along the width, and 7998 is not divisible by either. Therefore, the resulting data partitions contained between 173×249 and 174×250 elements. This corresponds to a 0.98% larger workload for the tiles that receive the largest data partitions.

The Poplar implementation of the tile mapping of `a` and `b` is shown in Listing 4.2.

[1] The possible sets of partitions when using one MK2 IPU are $\{1, 1472\}$, $\{2, 736\}$, $\{4, 368\}$, $\{8, 184\}$, $\{16, 92\}$, $\{32, 46\}$, $\{64, 23\}$, $\{23, 64\}$, $\{46, 32\}$, $\{92, 16\}$, $\{184, 8\}$, $\{368, 4\}$, $\{736, 2\}$, and $\{1472, 1\}$.

```
1  for (std::size_t x = 0; x < nh; ++x) {
2    for (std::size_t y = 0; y < nw; ++y) {
3
4      // tile_id runs from 0, 1, 2, ..., nh*nw (=num_tiles)
5      unsigned tile_id = y + x*nw;
6
7      // Evaluate offsets (partition the inner mesh, but include boundaries in
       ↪  tile mapping)
8      int offset_top = (x == 0) ? 0 : 1;
9      int offset_left = (y == 0) ? 0 : 1;
10     int offset_bottom = (x == nh - 1) ? 2 : 1;
11     int offset_right = (y == nw - 1) ? 2 : 1;
12
13     // Find low and high indices of the slice
14     int x_low = x*(height - 2)/nh + offset_top;
15     int y_low = y*(width - 2)/nw + offset_left;
16     int x_high = (x + 1)*(height - 2)/nh + offset_bottom
17     int y_high = (y + 1)*(width - 2)/nw + offset_right
18
19     // Map a slice of a to a specific tile ID
20     graph.setTileMapping(a.slice({x_low, y_low}, {x_high, y_high}), tile_id);
21   }
22 }
23
24 // Apply the same tile mapping to "b"
25 const auto& tile_mapping = graph.getTileMapping(a);
26 graph.setTileMapping(b, tile_mapping);
```

Listing 4.2: The tile mapping of tensor `a` to all the tiles on one IPU, where `nh` and `nw` are the number of partitions along the height and width direction, respectively.

Listing 4.2 highlights a convenient way to ensure that several tensors are tile mapped identically. On lines 25–26 the tile mapping of `a` is re-applied for `b`. Note that if a tensor is not completely mapped, Poplar will throw an error that says *incomplete tile mapping*.

The Poplar programs were set up to perform "back-and-forth" computations between the two tensors. More precisely, two unique compute sets were declared. These two compute sets were identical in all other aspects except for having swapped the tensors a and b. Listing 4.3 shows the implementation of one of the compute sets.

```
1  // In this compute set in=a and out=b. Similarly, there is an opposite compute
   ↪  set where in=b and out=a. The computation will alternate between these two
   ↪  compute sets.
2  auto compute_set = graph.addComputeSet("HeatEquation_a_to_b");
3
4  for (std::size_t x = 0; x < nh; ++x) {
5    for (std::size_t y = 0; y < nw; ++y) {
6
7      // tile_id should run from 0, 1, ..., nh*nw (=num_tiles)
8      unsigned tile_id = index(x, y, nw);
9
10     // Start indices
11     unsigned tile_x = block_low(x, nh, height-2);
12     unsigned tile_y = block_low(y, nw, width-2);
13
14     // loop over worker contexts
15     for (std::size_t worker_i = 0; worker_i < num_workers_per_tile; ++worker_i)
       ↪  {
16
17       // Dividing tile work by further splitting up the mesh along the height
         ↪  (x)
18       unsigned x_low = tile_x + block_low(worker_i, num_workers_per_tile,
         ↪  tile_height) + 1;
19       unsigned worker_height = block_size(worker_i, num_workers_per_tile,
         ↪  tile_height);
20       unsigned x_high = x_low + worker_height;
21
22       unsigned y_low = tile_y + 1;
23       unsigned worker_width = tile_width;
24       unsigned y_high = y_low + worker_width;
25
26       // Assign vertex to graph
27       auto v = graph.addVertex(compute_set, vertex); // std::string vertex is
         ↪  the name
28       graph.connect(v["in"], in.slice({x_low-1, y_low-1}, {x_high+1,
         ↪  y_high+1}));
29       graph.connect(v["out"], out.slice({x_low, y_low}, {x_high, y_high}));
30       graph.setInitialValue(v["worker_height"], worker_height);
31       graph.setInitialValue(v["worker_width"], worker_width);
32       graph.setInitialValue(v["alpha"], alpha);
33       graph.setTileMapping(v, tile_id);
34     }
35   }
36 }
```

Listing 4.3: The setup of a compute set, which includes assignment of vertices to it. This compute set performs one time step of equation (4.4). Note that the input slice for each vertex (line 28) is padded compared to the tile mapping shown in Listing 4.2. This imposes Poplar to fetch "missing" data in a communication phase prior to the execution of the vertex.

The executions of compute sets are the *computational phases* of the execution,

whereas between these phases there are *communication phases*. In Poplar, communication phases are not explicitly implemented. By assigning input slices that require more data than which is already mapped to the same tile, the necessary communication will automatically be handled. The data points that must be communicated will be referred to as the *halo region* for that tile. Poplar automatically constructs communication phases, because the graph implicitly knows that the halo regions must be fetched from the respective tiles where the data is mapped to, prior to the execution of this compute set.

The compute set shown in Listing 4.3 performs one time step of the heat equation. Each tile contained a workload which exactly corresponded to the tile mapping in Listing 4.2. However, the work was further divided among multiple workers, by splitting up the slices further along the height. Therefore, six vertices were assigned to each tile.

Vertices

In this section, two different codelets that compute the heat equation are presented. Vertices are compiled codelets: a set of instructions executed by a single worker context. The first codelet was fast and simple to implement, whereas the second was more technically challenging to implement. The second codelet included several optimization steps for improved performance. Listing 4.4 shows the simple codelet that solves the 2D heat equation.

```
class HeatEquationSimple : public Vertex {
public:
  HeatEquationSimple();

  // "in" is padded, thus has shape [worker_height+2, worker_width+2]
  Vector<Input<Vector<float, VectorLayout::SPAN, 4, false>>> in;
  Vector<Output<Vector<float, VectorLayout::SPAN, 4, false>>> out;
  const int worker_height;
  const int worker_width;
  const float alpha;

  bool compute () {
    const float beta{1.0f - 4.0f*alpha};
    for (int i = 1; i < worker_height + 1; ++i) {
      for (int j = 1; j < worker_width + 1; ++j) {
        out[i-1][j-1] = beta*in[i][j] + alpha*(in[i-1][j] + in[i+1][j] +
        ↪  in[i][j-1] + in[i][j+1]);
      }
    }
    return true;
  }
};
```

Listing 4.4: A simple codelet that computes the 2D heat equation on an input, `in`, and writes the results to `out`, both of which are connected to slices of a tensors.

The codelet in Listing 4.4, involves a double loop that slides the 5-point stencil row-wise across the input slice. Line 16 corresponds to equation (4.4), where a constant factor `beta` is computed prior to the loop. Next, Listing 4.5 shows the optimized codelet.

```
#include <ipudef.h> // for float2

class HeatEquationOptimized : public Vertex {
public:
  HeatEquationOptimized();

  Vector<Input<Vector<float, VectorLayout::SPAN, 8, false>>> in;
  Vector<Output<Vector<float, VectorLayout::SPAN, 4, false>>> out;
  const int worker_height;
  const int worker_width;
  const float alpha;

  bool compute () {
    const int half_worker_width = worker_width/2 + (worker_width % 2);
    const float beta{1.0f - 4.0f*alpha};
    typedef float float2 __attribute__((ext_vector_type(2)));
    float2 temp; // Temporary variable

    // Loop over rows, only update left/right column
    for (int i = 1; i < worker_height + 1; ++i) {
      // Left column
      int j = 1;
      out[i-1][j-1] = beta*in[i][j] + alpha*(in[i+1][j] + in[i-1][j] +
      ↪  in[i][j+1] + in[i][j-1]);

      // Right column (only if worker_width is even)
      if (worker_width % 2 == 0) {
        j = worker_width; // right column
        out[i-1][j-1] = beta*in[i][j] + alpha*(in[i+1][j] + in[i-1][j] +
        ↪  in[i][j+1] + in[i][j-1]);
      }
    }

    // Loop over rows, update inner columns, vectorized
    for (int i = 1; i < worker_height + 1; ++i) {
      // Illustration of two stencils along worker_width
      //      a   b
      //  c   d   e   f
      //      g   h

      // Declare float2 pointers, a:b, d:e, and g:h
      const float2 * __restrict__ north  = (float2 *) &in[i - 1][0];
      const float2 * __restrict__ middle = (float2 *) &in[i + 0][0];
      const float2 * __restrict__ south  = (float2 *) &in[i + 1][0];

      // Loop over inner columns, vectorized
      for (int j = 1; j < half_worker_width; ++j) {
        temp.x = middle[j - 1].y + middle[j].y; // c + e
        temp.y = middle[j].x + middle[j + 1].x; // d + f
        temp = beta*middle[j] + alpha*(north[j] + temp + south[j]);
        out[i - 1][2*j - 1] = temp.x; // Store left stencil
        out[i - 1][2*j - 0] = temp.y; // Store for right stencil
      }
    }
    return true;
  }
};
```

Listing 4.5: An optimized codelet that performs the 2D heat equation for a structured 2D mesh.

The codelets in Listing 4.4 and 4.5 perform the exact same computation. However, the latter includes optimization steps to enforce vectorization: handling two-and-two 32-bit elements in parallel. On line 7, `in` is constrained to have an alignment of 8 bytes, which means that 8-byte wide instructions can be applied to the first element in every row, and every second element from there. The `float2 temp` variable can hold two 32-bit floats, each accessible through `temp.x` and `temp.y`.

The vectorization in Listing 4.5 was set up to simultaneously solve two neighboring stencils along the width-direction. However, the leftmost column had to be solved unvectorized due to memory alignment requirement[2]. Therefore, the loop on lines 20–30 was included to handle the leftmost column. Later, the vectorized loop solves two-and-two stencils. On lines 46–48, `temp` is computed so that it holds the solution of both stencils. The results are stored sequentially, because the alignment of `out` cannot be guaranteed to be 8-byte, which is a requirement for a vectorized store operation.

IPU Execution

The performance of the computations relies heavily on two factors. First, achieving efficient communication phases relies on the partitioning of the mesh, and the general setup that imposes communication. Second, the computational phases rely solely on the vertices, which can be affected by the codelet implementations. The performance was measured by evaluating the wall time of 1000 time steps by using the *chrono* tools in C++.

The PDE solver was executed on a single IPU. The 2D mesh was set up to nearly fill the in-processor memory with a problem size of $8000 \times 8000 = 64$ million elements. The performance was measured by two metrics, computational throughput and minimal memory bandwidth. The throughput is a measurement of the average number of floating-point operations per second. For the 2D heat equation, the throughput was calculated by taking 6 arithmetic operations multiplied with the number of stencils and the number of time steps, and divided by the wall time. The calculation reads

$$\textit{throughput} = \frac{(6\ \textit{operations})(\textit{inner area})(\textit{no. time steps})}{(\textit{measured wall time})}. \tag{4.5}$$

The minimal memory bandwidth represents the lower bound of the total memory traffic that must have taken place. This includes both the amount of data that was stored to and loaded from the in-processor memory per second. The amount of memory traffic in the communication phases was added up. The amount of memory traffic in the computational phases was found by counting 5 memory

[2]The leftmost column in the output corresponds to the leftmost stencils which are centered at the *second* column of the input. Thus, the center values have an alignment of 4. Hence, 64-bit vectorization cannot be applied, because it requires 8-byte alignment

operations per stencil, multiplied with the amount of data in the inner area of the mesh. Lastly, these two numbers were added together, and multiplied with the number of time steps, and divided by the wall time. Hence, the calculation becomes

$$\textit{communication ops.} = (2\ \textit{ops.})(\textit{communication volume}\ ), \tag{4.6}$$

$$\textit{computation ops.} = (5\ \textit{ops.})(\textit{inner area}), \tag{4.7}$$

$$\textit{minimal bandwidth} = (\textit{comm. ops.} + \textit{comp. ops.})\,\frac{(4\ \textit{bytes})(\textit{no. time steps})}{(\textit{measured wall time})}. \tag{4.8}$$

Someone with a sharp eye might say that there should be 6 memory operations per stencil: loading 5 elements plus storing 1. However, the limited register files can hold enough data to solve two stencils per iteration. Furthermore, after inspecting the assembly code, two stencils only required loading 8 elements and storing 2, which is an average of 5 memory operations per stencil.

The heat equation is more generally known as the *isotropic diffusion equation*, which can also be used for noise reduction in images. As a demonstration of this application, the heat equation was applied on an input image of 4289×2835 pixels and one channel. This computation was included for both illustrational purposes, and as a double check that the computations worked as intended.

The codes were compiled with Poplar SDK 2.2.0, GCC 7.5.0, and the -O3 optimization flag. Lastly, PopVision was used to study executions that were performed with profiling enabled, which could provide further insight and understanding of the performance.

4.3.2 CPU Implementation

The discretized 2D heat equation was implemented in C with *OpenMP* for the CPU to serve as a comparison to the IPU implementation. The CPU execution featured the same problem size as the IPU execution with a 2D mesh of 8000×8000 single precision elements, and the execution ran for 1000 time steps. A snippet of the OpenMP CPU code is shown in Listing 4.6.

```
#pragma omp parallel private(i,j,k)
{
  for (k = 0; k < num_iterations; ++k) {
    #pragma omp for
    for (i = 1; i < height - 1; ++i)
      for (j = 1; j < width - 1; ++j)
        b[i][j] = beta*a[i][j] + alpha*(a[i-1][j] + a[i][j-1] + a[i][j+1] +
        ↪  a[i+1][j]);

    #pragma omp single
    {
      // pointer swap
      tmp = b;
      b = a;
      a = tmp;
    }
  }
}
```

Listing 4.6: A snippet of the OpenMP implementation of the 2D heat equation that was executed on a CPU. The code allows for a multi-threaded CPU execution.

The time usage was measured by using OpenMP's built-in wall time tool. This number was used to calculate two metrics. The throughput of the CPU code was calculated the same way as for the IPU code, as shown in equation (4.5). The minimal memory bandwidth was calculated differently than for the IPU execution. The main reason for this is that the CPU system features a shared memory model, and the CPUs additionally feature several levels of cache. This means that the theoretical *minimal* memory traffic is considerably lower than for the IPU, because on the CPU, many of the elements can be re-used from cache. Therefore, the minimal memory bandwidth is calculated by assuming the ideal case where the entire mesh is loaded and stored once. This corresponds to 2 memory operations, multiplied with the number of bytes per element (4), the total area, and number of time steps, and divided by wall time.

$$\textit{minimal bandwidth} = \frac{(2\ \textit{operations})(4\ \textit{bytes})(\textit{total area})(\textit{no. time steps})}{(\textit{measured wall time})}. \tag{4.9}$$

In total, the minimal bandwidth represents the least amount of memory traffic that must have taken place between the memory and the CPUs.

The CPU execution was performed on a Linux server with two AMD Epyc 7601 (Naples) 32-core CPUs, connected in a dual-socket. In total, the execution ran on 128 threads. The code was compiled with GCC 11.1.0, linked with OpenMP 4.5, and with the optimization flag -O3.

4.4 Results

Table 4.1 shows the performance for the two different vertices, on the MK2 IPU, and the OpenMP implementation on the CPUs. All three executions performed the 2D heat equation on an 8000×8000 mesh of single precision elements and ran for 1000 time steps.

Processor	Vertex	Time	Throughput	Minimal Bandwidth
CPU	N/A	4.05 s	94.73 GFLOPS	126.37 GB/s
IPU	*HeatEquationSimple*	0.39 s	0.98 TFLOPS	3.31 TB/s
IPU	*HeatEquationOptimized*	0.30 s	1.28 TFLOPS	4.28 TB/s

Table 4.1: The execution time, throughput, and minimal bandwidth, for three codes that all performed the 2D heat equation on a 8000×8000 mesh for 1000 time steps.

An inspection of the assembly codes of the two IPU codelets revealed that *HeatEquationSimple* used a total of 22 clock cycles per two stencils, while *HeatEquationOptimized* on the other hand, used 16 clock cycles per two stencils. Further, a study of execution profile using PopVision revealed several other metrics about the execution. Table 4.2 shows the clock cycle counts for both the compute and exchange phases per time step of the heat equation.

Codelet	OnTileExecute [Cycles]	DoExchange [Cycles]
HeatEquationSimple	516,189	44,943
HeatEquationOptimized	398,492	44,943

Table 4.2: The number of clock cycles for one time step during executions of the heat equation on the IPU. *OnTileExecute* denotes the computational phases, and *DoExhcange* denotes the internal exchange (tile-to-tile).

Figure 4.1 shows two images: an initial noisy image of Mona Lisa, and the resulting smoothed image after the heat equation was applied on it for 100 time steps with a fixed $\alpha = 0.1$.

(a) A noisy image. **(b) A smoothed image.**

Figure 4.1: The left image is a noisy 1-channel image of the famous Mona Lisa painting, containing 4289×2835 pixels. The right image shows the result after the heat equation has been applied on it, on the IPU for 100 time steps.

4.5 Discussion

Table 4.1 shows that the IPU executions achieved substantially higher performance than the CPU execution. The two IPU executions, the *HeatEquationSimple* vertex and the *HeatEquationOptimized* vertex, achieved 10 and 14 times higher throughput than the CPU execution, respectively. However, the IPU features $1472/64 = 23$ times more cores. Therefore, if the throughput is adjusted to a per-core basis, the IPU cores achieved 45% and 59% of the performance of a CPU core, for the two IPU implementations. This result suggests that even on a per-core basis, the IPU was not very far behind a modern CPU.

Comparing the achieved minimal memory bandwidth of the executions can be considered a less fair metric than the computational throughput, because the two processors exhibit very different architectures. Regardless, the minimal bandwidth measurements were 26 and 34 times higher for the IPU executions than the CPU execution. The performance gaps are much larger compared to the corresponding gaps from the throughput measurements. The cacheless IPU

architecture relies on fetching data from the in-processor memory more often than the CPU. Further, the IPU features a distributed memory model, which means that data must be communicated among the tiles. Lastly, due to the large tile count, the IPU has considerably higher theoretical peak memory bandwidth than the CPU.

When comparing the two IPU implementations against each other, the optimized vertex executed 1.3 times faster than the simple vertex. This was expected because the optimized vertex enforced a higher degree of vectorization, which means that the execution pipelines will handle more elements simultaneously. An inspection of the assembly codes also revealed that the inner loop of the optimized vertex cut down the number of clock cycles from 22 to 16. These numbers could be used to estimate an upper bound of the performance increase 1.4 times faster for the optimized vertex. This number is likely higher than the measured speedup, because it only takes the inner loop body into consideration, and not the rest of the vertex code. By looking at the total clock cycle counts in Table 4.2, the expected performance increase is $(516189 + 44943)/(392492 + 44943) \approx 1.3$ times, which corresponds to the observed speedup. Lastly, vectorization was something that the compiler could have applied to the simple vertex too. Therefore, the performance gap between the two vertices highlights a potential for improvement for the compiler.

The IPU computations were controlled against a CPU computation to ensure that the results were correct. First, a very small mesh of 4×4 elements and 1–2 time steps were controlled against both a by-hand calculation, and a CPU code. Second, executions on larger meshes were controlled against the CPU code. This highlights the incorporation of error-control, which is important in general scientific computing.

Figure 4.1 shows the effects of the heat equation when applied to a 1-channel image. This application served as a verification that the computations worked as intended. For instance, if the code was set up wrong and did not perform the computations for a row or a column, it would be apparent in the smoothed image.

Solving the heat equation by finite difference on the IPU required many floating-point computations. This scientific application can benefit from high degrees of parallelism. Low-level Poplar programming in C++ was able to reach a high performance for 2D stencil-based computations. The implementation of *HeatEquationSimple* is feasible to new users of Poplar. Even this implementation, which was considerably faster to implement than the optimized vertex, substantially outperformed the dual-CPU execution, parallelized by OpenMP with 128 threads.

The 2D heat equation was implemented and tested on the IPU to discuss challenges in adopting this AI-specialized processor for scientific computing workloads. It would be interesting to extend the computation to unstructured meshes, which can be used in physics-simulations of more advanced real-life

problems. In developing such programs, it would be harder to partition the data, and to equally distribute the workload. However, Poplar's graph programming framework could ease some of the challenges of implementing such programs.

Chapter 5

The 3D Heat Equation by Finite Differences

5.1 Background

In Chapter 4, the discretized heat equation was introduced. It was derived by using an explicit finite difference scheme. The equation was applicable to a regular 2D mesh, where the distances between the points were uniform and constant. The discretized heat equation can be extended to three dimensions, which is given by

$$\begin{aligned} u_{ijk}^{t+\delta} = &(1-6\alpha)\, u_{ijk}^{t} + \\ &\alpha \left(u_{i+1,j,k}^{t} + u_{i-1,j,k}^{t} + u_{i,j+1,k}^{t} + u_{i,j-1,k}^{t} + u_{i,j,k+1}^{t} + u_{i,j,k-1}^{t} \right), \end{aligned} \tag{5.1}$$

where u is a regular 3D mesh, $\alpha = \kappa\delta/h^2$, κ is a constant property of the substance, δ is the time step, and h is the uniform spatial step size. The finite difference discretization applied the *forward difference in time*, and *central difference in space*, which is an explicit method for solving the heat equation numerically. The right-hand side of equation (5.1) represents a 7-point stencil, which can be employed as a numerical algorithm. The scheme is only stable and convergent for $\alpha \leq 0.5$.

The stencils are only defined for the inner nodes of a 3D mesh. The Dirichlet boundary condition was applied. Hence, the inner elements were updated by the stencil, whereas the boundary nodes were kept constant.

5.2 Methods

For the 3D heat equation, a mesh that pushed the limited memory capacity of one IPU was used. This was a mesh of $360 \times 360 \times 360$ single precision elements. The same mesh and precision were also used in the CPU implementation.

5.2.1 IPU Implementation

Work Division

The tensors were instantiated as 3D tensors, which were partitioned into smaller 3D slices. Each partition was mapped to a different tile on the IPU. It is desirable to divide the mesh in a way that minimizes the surface area of each partition. Therefore, an algorithm that searched for the number of partitions in each dimension was implemented. The algorithm was constrained to utilize all tiles and searched for the partitioning that minimized the communication volume. The algorithm is shown in Listing 5.1.

```
float smallest_surface_area = std::numeric_limits<float>::max();
std::size_t nh, nw, nd;
for (std::size_t i = 1; i*i <= tile_count; ++i) {
  if (tile_count % i == 0) { // then i is a factor

  // Further, find two other factors, to obtain exactly three factors
  std::size_t other_factor = tile_count/i;
  for (std::size_t j = 1; j <= other_factor; ++j) {
    if (other_factor % j == 0) { // then j is a second factor
      std::size_t k = other_factor/j; // and k is the third factor
      std::vector<std::size_t> splits = {i,j,k};

      // test all (6) ways to assign i, j, k as nh, nw, nd
      for (std::size_t l = 0; l < 3; ++l) {
        for (std::size_t m = 0; m < 3; ++m) {
          for (std::size_t n = 0; n < 3; ++n) {
            if (l != m && l != n && m != n) {
              float slice_height = float(height)/float(splits[l]);
              float slice_width = float(width)/float(splits[m]);
              float slice_depth = float(depth)/float(splits[n]);
              float surface_area = 2.0*(slice_height*slice_width +
              ↪  slice_depth*slice_width + slice_depth*slice_height);
              if (surface_area <= smallest_surface_area) {
                smallest_surface_area = surface_area;
                nh = splits[0];
                nw = splits[1];
                nd = splits[2];
              }}}}}}}}
```

Listing 5.1: An algorithm to find a partitioning of the mesh that minimizes the total communication volume.

The algorithm in Listing 5.1 can be summarized as follows:

1. Factorize the tile count into three factors. All combinations will be tested.
2. Given the three factors, test all ways of dividing the mesh, which are $3! = 6$ combinations.
3. For every given partitioning possibility, calculate the surface area of the resulting partition, which corresponds to the communication volume.
4. Choose the combination that minimizes the surface area.

Note that the surface area calculated in step 3 does not represent the complete communication volume. Some tiles will be given partitions that consist of one extra element per dimension. However, the surface area of the smallest partition is a sufficiently good estimate of the communication volume.

When the partitioning was found, the graph could be constructed. The tensors were split up and distributed among the tiles. Since the height, width, or depth might not be divisible by the respective number of partitions in that direction, some tiles got assigned workloads that were larger than the others. The data imbalance would at most result in one extra element per dimension. For the $360 \times 360 \times 360$ mesh, the number of partitions were 23, 8, and 8. This resulted in tile workloads between $15 \times 44 \times 44$ and $16 \times 45 \times 45$ elements. Hence, the largest workloads were 11.6% larger than the smallest.

Vertices

After the tensors were distributed, the vertices were assigned. Poplar implicitly handled necessary communication. This was done by requiring tensor data that did not reside on the tile the vertex was assigned to. The inputs to each vertex were padded compared to the tile mapping. Hence, the inputs have a data requirement beyond the data that was already stored on the tile. This imposes Poplar to set up communication messages to fetch the missing data from other tiles, prior to the execution of the vertex. Listing 5.2 shows how the vertices are assigned to the graph.

```
1  out_slice = out.slice(
2      {x_low, y_low, z_low},
3      {x_high, y_high, z_high}
4  );
5
6  // Pad the input
7  in_slice = in.slice(
8      {x_low-1, y_low-1, z_low-1},
9      {x_high+1, y_high+1, z_high+1}
10 );
11
12 auto v = graph.addVertex(compute_set, vertex);
13 graph.connect(v["in"], in_slice.flatten(0,2));
14 graph.connect(v["out"], out_slice.flatten(0,2));
15 graph.setInitialValue(v["worker_height"], worker_height);
16 graph.setInitialValue(v["worker_width"], worker_width);
17 graph.setInitialValue(v["worker_depth"], worker_depth);
18 graph.setInitialValue(v["alpha"], alpha);
19 graph.setTileMapping(v, tile_id);
```

Listing 5.2: A code snippet that shows the part of the program where a vertex is assigned. This piece of code is taken from a nested loop that iterates over all tiles and threads.

The vertex-variables that can be connected to tensor slices can at most be 2D (even though tensors can be of higher dimensions), as first introduced in section 3.3.2. This effectively means tensors with three or more dimensions must be flattened in order to be connected to a vertex. On lines 2–3 in Listing 5.2, the tensor slices are flattened from 3D to 2D, where the depth dimension is kept. The first index represents both height and width, and the second index only goes along the depth[1].

Two codelets with different purposes were implemented. The first served as a fast and simple implementation, and the second was optimized by enforcing a higher degree of vectorization. Listing 5.3 shows the simple codelet.

[1]With this flattening, accessing element `[i][j][k]` of the original 3D tensor, are now done with two indices, `[j + worker_width*i][k]` instead.

```
class HeatEquationSimple : public Vertex {
public:
  HeatEquationSimple();

  Vector<Input<Vector<float, VectorLayout::SPAN, 8, false>>> in;
  Vector<Output<Vector<float, VectorLayout::SPAN, 4, false>>> out;
  const unsigned worker_height;
  const unsigned worker_width;
  const unsigned worker_depth;
  const float alpha;

  unsigned idx(unsigned x, unsigned y, unsigned w) {
    /* Index corresponding to [x,y] for a row-wise flattened 2D variable */
    return y + x*w;
  }

  bool compute () {
    const float beta{1.0f - 6.0f*alpha};
    const unsigned iw = worker_width + 2; // width of "in" which is padded

    for (std::size_t x = 1; x < worker_height + 1; ++x) {
      for (std::size_t y = 1; y < worker_width + 1; ++y) {
        for (std::size_t z = 1; z < worker_depth + 1; ++z) {
          out[idx(x-1,y-1,worker_width)][z-1] = beta*in[idx(x,y,iw)][z] +
            alpha*(
              in[idx(x+1,y,iw)][z] +
              in[idx(x-1,y,iw)][z] +
              in[idx(x,y+1,iw)][z] +
              in[idx(x,y-1,iw)][z] +
              in[idx(x,y,iw)][z+1] +
              in[idx(x,y,iw)][z-1]
            );
        }
      }
    }

    return true;
  }
};
```

Listing 5.3: A codelet that solves the discretized heat equation for a 3D mesh. Since the input is padded, the corresponding element of the output must be indexed with an offset of -1.

In Listing 5.3, the tensor-connecting variables are 2D. The codelet was written to iterate over and update every element of the output. Inside the loop, the discretized heat equation was implemented. Note that the input is padded compared to the output. The loop indices are set up with respect to the input, which means that the corresponding element of the output must be indexed with offsets.

In addition to the *HeatEquationSimple* vertex, the *HeatEquationOptimized* vertex was implemented much like the optimized 2D vertex. Vectorization was enforced, which in this case meant that two-and-two stencils were solved simultaneously. The vectorization was performed along the depth dimension, because this dimension was kept when the tensors were flattened. Therefore, the alignment of this dimension was controlled and set to 8, which meant that vectorization could be enforced.

IPU Execution

Equation (5.1) was applied to a $360 \times 360 \times 360$ mesh of single precision elements, for 1000 time steps on the IPU. The wall time around the execution was measured by using the *chrono* tools in C++ and was used to calculate two metrics. The first was the computational throughput, which represents the average rate of floating-point operations per second (single precision). The throughput was calculated by taking 8 arithmetic operations per stencil, multiplied with the number of inner elements and number of time steps, and divided by the wall time. The number 8 was found by adding up the additions and multiplications in equation (5.1). The calculation of the throughput is given by

$$\textit{throughput} = \frac{(8\ \textit{operations})(\textit{inner volume})(\textit{no. time steps})}{(\textit{measured wall time})}. \tag{5.2}$$

The second metric that was measured was the minimal memory bandwidth. This was calculated by summing up the number of memory operations in both communication phases and computational phases. Next, this number was multiplied with 4 bytes (size per element) and number of time steps, and divided by the measured wall time. These steps are summarized as

$$\textit{communication ops.} = (2\ \textit{ops.})(\textit{communication volume}), \tag{5.3}$$

$$\textit{computational ops.} = (7\ \textit{ops.})(\textit{inner volume}),\ \text{and} \tag{5.4}$$

$$\textit{minimal bandwidth} = (\textit{comm. ops.} + \textit{comp. ops.}) \tag{5.5}$$

$$\frac{(4\ \textit{bytes})(\textit{no. time steps})}{(\textit{measured wall time})}. \tag{5.6}$$

The computational phases only require 7 (32-bit) memory operations per stencil instead of 8. This is because the registers can hold enough values to solve two stencils at a time, and two stencils only require loading 12 and storing 2 elements, which on average is only 7 memory operations per stencil. On the other hand, the number of memory operations in the communication phases was found by adding up the halo regions for all tiles. In total, the minimal bandwidth is the least amount of memory traffic that must have taken place during the execution.

The codes were compiled with Poplar SDK 2.2.0, GCC 7.5.0, and the -O3 optimization flag. The IPU program was executed twice, both with and without profiling enabled. This is important because profiling significantly affects performance. The profiling report was analysed in PopVision to provide further insights into the performance.

5.2.2 CPU Implementation

The discretized 3D heat equation was implemented in standard C along with OpenMP. A 3D mesh of $360 \times 360 \times 360$ single precision elements was used. Therefore, two float arrays were allocated by using triple pointers. Then, the heat equation was computed by allowing OpenMP to use 128 different threads. At the end of each time step, the pointers were swapped. Listing 5.4 shows a code snippet from the CPU implementation.

```
#pragma omp parallel private(t,i,j,k)
{
  for (t = 0; t < num_iterations; ++t) {
    #pragma omp for
    for (i = 1; i < height - 1; ++i)
      for (j = 1; j < width - 1; ++j)
        for (k = 1; k < depth - 1; ++k)
          b[i][j][k] = beta*a[i][j][k] + alpha*(a[i+1][j][k] + a[i-1][j][k] +
          ↪  a[i][j+1][k] + a[i][j-1][k] + a[i][j][k+1] + a[i][j][k-1]);
    #pragma omp single
    {
      // pointer swap
      tmp = b;
      b = a;
      a = tmp;
    }
  }
}
```

Listing 5.4: The OpenMP CPU implementation of the discretized heat equation for a 3D mesh.

The wall time was measured by using OpenMP's built-in wall time tool. Two metrics of the execution were computed by using the measured wall time. First, the throughput was calculated by the same equation as for the IPU executions, as shown by equation (5.2). Second, the minimal bandwidth was calculated by assuming the ideal case where the entire mesh must be loaded and stored once per time step. Therefore, this metric was calculated by multiplying the entire volume with 2 memory operations, 4 bytes, and number of time steps, and divided by the measured wall time. The calculation can be summarized as

$$\textit{minimal bandwidth} = \frac{(2\ \textit{operations})(4\ \textit{bytes})(\textit{total volume})(\textit{no. time steps})}{(\textit{measured wall time})}. \tag{5.7}$$

The CPU code was executed on a Linux server with two AMD Epyc 7601 (Naples) 32-core CPUs. The code was compiled with GCC 11.1.0, along with OpenMP 4.5, and the -O3 optimization flag.

5.3 Results

Table 5.1 shows the results from computing the discretized heat equation on a 3D mesh of $360 \times 360 \times 360 \approx 46.6$ million elements, for 1000 time steps.

Processor	Vertex	Time	Throughput	Minimal Bandwidth
CPU	N/A	8.83 s	41.55 GFLOPS	42.25 GB/s
IPU	*HeatEquationSimple*	0.43 s	0.87 TFLOPS	3.11 TB/s
IPU	*HeatEquationOptimized*	0.26 s	1.44 TFLOPS	5.15 TB/s

Table 5.1: The measured performance for three executions of the discretized heat equation on a $360 \times 360 \times 360$ mesh of single precision elements, for 1000 time steps.

Table 5.2 shows additional information of the IPU executions extracted from the *execution trace* in PopVision: the clock cycle counts for both computational and communications phases in one time step of the heat equation.

Vertex	OnTileExecute [Cycles]	DoExchange [Cycles]
HeatEquationSimple	570,165	43,505
HeatEquationOptimized	343,893	43,505

Table 5.2: The number of clock cycles for one time step during the execution of the heat equation on the IPU. *OnTileExecute* denotes a computational phase, and *DoExhchange* denotes a communication phase.

5.4 Discussion

Table 5.1 shows that both IPU executions significantly outperformed the CPU execution. This can be shown by the two measured metrics, throughput and

minimal memory bandwidth. The *HeatEquationSimple* and the *HeatEquationOptimized* vertices achieved throughput measurements of 21 and 35 times higher than the CPU, respectively. This was a substantial performance increase. It was expected that the measured throughput would be a lot higher on the IPU, because of the abundance of tiles and low-latency in-processor memory. Furthermore, since the IPU system exhibits $1472/64 = 23$ times more cores than the CPU system, the throughput can be adjusted to a per-core basis. The IPU core then achieves 91% and 151% of the computational throughput of a CPU core, for the two IPU implementations, respectively. This result suggests that for this computational workload, the IPU is substantially better than the CPU, even per-core. This shows that the many-core design of the IPU does not necessarily mean that the performance of each individual core is compromised. This could be partially explained by a memory bandwidth bottleneck for this application. If that is the case, then the result suggests that the distributed in-processor memory architecture can limit this bottleneck considerably.

The minimal memory bandwidth measurements of the two IPU implementations were 3.11 and 5.15 TB/s. Based on the minimal memory bandwidth in Table 5.1, the simple and optimized IPU implementations reached 74 and 122 times higher rates than the CPU, respectively. These performance gaps are much higher than the corresponding gaps based on throughput measurements. This is because the minimal bandwidth was calculated differently for the two processors, which arise from the fact that the CPU and IPU exhibit very different memory models. The IPU does not have cache, and thus relies on fetching data from memory much more often than the CPU, at least in the ideal situation where the CPU can reuse data from cache. An advantage of the IPU's distributed memory model is that it has a much higher core count (1,472), and each core has its own low-latency memory located on the same tile. The CPUs on the other hand, have much fewer cores (64) and exhibit a shared memory model. Furthermore, the CPUs also have several levels of shared cache, which means that they theoretically do not have to load as much data from the memory as the IPU.

The 2D heat equation in Chapter 4 and the 3D heat equation in this chapter have very similar implementations. On the other hand, the results showed some noticeable differences. First, the computational throughput measurements of the IPU codes achieved comparable performance, where the 2D vertices achieved 0.98 and 1.28 TFLOPS. In 3D, the vertices achieved 0.87 and 1.44 TFLOPS. This means that the 3D case achieved lower performance than the 2D case in the simple implementations, but higher performance in the optimized ones. This result also indicates that the optimization had a much larger effect for the 3D application, achieving a speedup of 1.65 times, whereas the optimization in 2D only resulted in a speedup of 1.30 times.

It is important to include the simple vertices, even though they were expected to perform worse than their optimized counterparts. This is because it is most realistic for a programmer of general scientific computing to implement these, at least as a starting point. The optimized vertices are more technically difficult to

implement, requiring additional time and effort. The simple 2D vertex achieved a higher performance than the corresponding simple 3D vertex, of $0.98/0.87 \approx 1.13$ times. A possible explanation for this is that the compiler finds it harder to optimize the 3D case.

An interesting observation is that when comparing the optimized vertices, the performance gap is inversely proportional to the performance gap between the simple vertices. The optimized 3D vertex executed $1.44/1.28 \approx 1.13$ times faster than its 2D counterpart. An inspection of the assembly codes for the 3D vertices revealed that the inner loops used 27 (simple) and 16 (optimized) clock cycles. In contrast, the inner loops of the 2D vertices used 22 (simple) and 16 (optimized) clock cycles.

The simple vertex in 3D achieves quite low performance compared to the other vertices, which is also shown by the performance tables (4.1 and 5.1). The need for 27 clock cycles could be due to the limited number of registers, which ultimately could force the use of more memory operations. However, the observed performance of the optimized vertex immediately disproves this. Consider that the optimized vertex demonstrates the upper bound of achievable performance for this application. The simple vertex only achieves 59% of the optimized one. Therefore, it can only be concluded that this result suggests that the compiler has a large potential in improving the simple 3D vertex.

The optimized heat equation vertices used 16 clock cycles in both 2D in 3D, which is a remarkable result. The 3D vertex strictly requires four additional floating-point operations (two extra additions per stencil) and loading an additional 16 bytes of data. It might seem unfeasible that the additional workload 3D imposes was performed without compromising the number of clock cycles (compared to 2D). However, this is possible through co-issues. Each thread on the IPU has two execution pipelines, *main* and *aux*. This means that the additional operations needed in the 3D case can be included "for free".

Table 5.2 shows that the *communication phases* remained unchanged for the two vertices, and that the optimization only cut down the clock cycle counts for the *computational phases*. This result is consistent with the expectation, because the optimization steps that were taken should only affect the computational phases.

The CPU implementation of the 3D heat equation was very similar to the 2D implementation from Chapter 4. However, the performance was much worse in the 3D case, at 41.55 GFLOPS and a minimal bandwidth of 42.25 GB/s. The 2D code achieved 94.73 GFLOPS and a minimal bandwidth of 126.27 GB/s. The performance gap was significant, and it was likely caused by a memory bandwidth bottleneck. Every 3D stencil requires neighboring data points from the 3D mesh in all three dimensions. Since the mesh was a triple pointer float array, two of the neighboring points will be close in memory (depth), two other points will be further away in memory (width), and the two remaining points will be furthest away (height). The data was physically very spread out, which

was likely costly to fetch. On the other hand, the 2D CPU code achieved a much higher performance. In 2D, each stencil only requires four neighbor elements in total, and only two of them are stored further away in memory. Surely, this is a requirement that is easier to satisfy than the 3D case and will probably be more successful with cache. This strengthens the idea that the 3D CPU code suffers from a memory bottleneck, at least to a larger degree than in 2D.

The extension of the heat equation from 2D to 3D highlighted a general challenge in Poplar programming. Higher-dimensional tensors (3D and up) must be flattened to 1D or 2D in order to be connected to a vertex. This is arguably a weakness in Poplar programming for scientific computing, because 3D variables often are involved, and it can be inconvenient and cumbersome to manually handle the more advanced indexing of higher dimensional variables in lower dimensional types. As a consequence, this could increase the chance of user errors.

Scientific computing is interested in both 2D and 3D computations. On single-IPU executions, the IPU was demonstrated to achieve good performances for the selected stencil-based numerical schemes for solving partial differential equations. The measured performance of the IPU computations seemed less dependent on the problem at hand compared to the CPU. This was demonstrated by comparing the 2D and 3D application: the IPU achieved comparable performances, whereas the CPU seemed to suffer from a memory bandwidth bottleneck in 3D.

A suggestion for future improvement is included. The heat equation has been implemented and solved for a structured 3D mesh on the IPU. The code was highly parallelized and each tile was assigned similar workloads. However, the IPU is highly flexible and is able to run MIMD programs. In this regard, it could be interesting to solve the heat equation for unstructured meshes. This type of computation is closer to real-life use cases of the heat equation. Therefore, it would be interesting to see if the IPU is able to achieve high performance in such computations.

Chapter 6

Multi-IPU Executions of the Heat Equation

Scalability is one of the key design goals of the IPU. Since the IPU's applicability to general scientific computing workloads is studied, it is important to investigate the performance when scaling such computations to run on multiple IPUs. Additionally, new aspects and challenges are also discussed in this chapter.

6.1 Methods

6.1.1 Scaling the Problem Sizes

Following the principles explained in Chapter 4 and 5, the discretized heat equations in two and three dimensions were solved on multiple IPUs. To perform these computations, chip-to-chip communication is required. Global communication has a bandwidth of 320 GB/s, which is much slower than the internal communication, with a bandwidth of 8 TB/s [2]. Therefore, it was particularly important to optimize the work division.

It is desirable to push the limited memory capacity of the in-processor memory. Hence, when going from 1 to several IPUs, the number of elements in the meshes were subsequently increased. Although the total memory capacity increases proportionally with the number of IPUs, the *available* memory increases less than this. This is because more of the limited space must be dedicated to communication buffers. Therefore, the 2D and 3D meshes were scaled almost linearly with the number of IPUs, but slightly less. This was done with trial and error. Table 6.1 shows the resolutions of the meshes.

Number of IPUs	Dimensions	Elements	Total Data
1	2D	8000×8000	512 MB
2	2D	10000×10000	800 MB
4	2D	14000×14000	1.59 GB
8	2D	19000×19000	2.89 GB
16	2D	27000×27000	5.83 GB
1	3D	$360 \times 360 \times 360$	373 MB
2	3D	$403 \times 403 \times 403$	524 MB
4	3D	$508 \times 508 \times 508$	1.05 GB
8	3D	$640 \times 640 \times 640$	2.10 GB
16	3D	$806 \times 806 \times 806$	4.19 GB

Table 6.1: The increasing 2D and 3D meshes with the number of IPUs. These meshes were used for the multi-IPU executions in this chapter.

6.1.2 Work Division

The total 2D and 3D meshes were partitioned twice, hierarchically. The reason behind this approach is that the execution of the heat equation involved two levels of communication: chip-to-chip communication and tile-to-tile communication. The overall goal can be summarized as two prioritized tasks:

1. Minimize the chip-to-chip (global) communication costs.
2. Minimize the tile-to-tile (internal) communication costs.

First, the meshes were distributed among the available IPUs along *one* dimension: the 2D mesh was partitioned along width, and the 3D along depth. This imposed workload imbalances of at most one element along the respective dimension of the partitioning. The reason why the top-level work division only partitioned along *one* dimension, was that the number of IPUs was relatively low.

The second-level partitioning was on each IPU. The data partitions were in turn distributed among all 1,472 tiles of every IPU. This partitioning was done by finding the number of partitions in each dimension based on the respective algorithms (Listing 4.1 for 2D and Listing 5.1 for 3D). To recall, these algorithms minimize the internal exchange volume.

The *total mesh* refers to the full 2D or 3D mesh. Due to the Dirichlet boundary condition, the stencils were only applied to the *inner mesh*. The boundary nodes were kept constant. In order to construct balanced workloads, it was desirable to equally distribute the inner, and not the total mesh. The work division can be summarized in these steps:

1. Start with a 2D mesh of size $h \times w$ or 3D mesh of size $h \times w \times d$.

2. Divide the inner mesh among n IPUs, by splitting along one dimension. Thus, each IPU gets assigned a workload of $(h-2) \times (w-2)/n$ in 2D or $(h-2) \times (w-2) \times (d-2)/n$ in 3D. The division might impose a workload imbalance of at most one element in the respective dimension.

3. On each IPU, use an algorithm to find the partitioning that uses all tiles and minimizes the tile-to-tile communication volume.

To summarize, the results of the partitioning are illustrated in Table 6.2.

No. IPUs	Dims.	Total Mesh	Workload/Tile	Work Balance
1	2D	8000×8000	173×249	99.0%
2	2D	10000×10000	156×217	98.9%
4	2D	14000×14000	152×218	98.9%
8	2D	19000×19000	206×148	98.9%
16	2D	27000×27000	146×210	98.9%
1	3D	$360 \times 360 \times 360$	$15 \times 45 \times 45$	89.6%
2	3D	$403 \times 403 \times 403$	$17 \times 25 \times 50$	90.9%
4	3D	$508 \times 508 \times 508$	$22 \times 31 \times 31$	93.9%
8	3D	$640 \times 640 \times 640$	$20 \times 27 \times 40$	89.3%
16	3D	$806 \times 806 \times 806$	$25 \times 35 \times 25$	93.4%

Table 6.2: The partitioning of the 2D and 3D meshes. The workload per tile represents the *smallest* data partition solved by a tile for the given problem: other tiles might receive workloads containing up to one extra element per dimension. The work balance is found by taking the smallest tile workload divided by the largest.

6.1.3 Measurements

The time usage of the IPU executions was measured by using the *chrono* tools in C++. This number was in turn used to calculate two metrics: computational throughput and minimal memory bandwidth. The computational throughput was measured by equation (4.5) for 2D, and by equation (5.2) for 3D.

The minimal memory bandwidth was calculated to account for all memory operations (*ops.*) in computational phases, global exchange phases, and internal exchange phases, given by

$$\textit{global} = (4\ \textit{ops.})(\textit{global communication volume})(\textit{no. IPUs} - 1), \tag{6.1}$$

$$\textit{internal} = (2\ \textit{ops.})(\textit{internal communication volume})(\textit{no. IPUs}), \text{ and} \tag{6.2}$$

$$\textit{compute} = (N\ \textit{ops.})(\textit{inner volume}). \tag{6.3}$$

To summarize, the number of *global* memory operations is found by taking 4

operations (2 loads and 2 stores) between every IPU that should communicate with each other. Here, the external communication volume is simply the *height* in 2D, and *height*×*width* in 3D. The number of *internal* memory operations accounts for both storing and loading the entire internal communication volume on every IPU. Lastly, the number of memory operations in *compute* phases counts N memory operations per stencil, where $N = 5$ in 2D and $N = 7$ in 3D. Next, the three numbers of memory operations in these three phases were added together. Then, the bandwidth was found by

$$\textit{minimal bandwidth} = (\textit{internal} + \textit{global} + \textit{compute}) \frac{(4\ \textit{bytes})(\textit{no. time steps})}{(\textit{measured wall time})}. \tag{6.4}$$

To summarize, equation (6.4) represents the minimum rate of data that is both loaded from and stored to the in-processor memory per second.

The IPU codes were compiled with Poplar SDK 2.2.0, GCC 7.5.0, and the -O3 optimization flag.

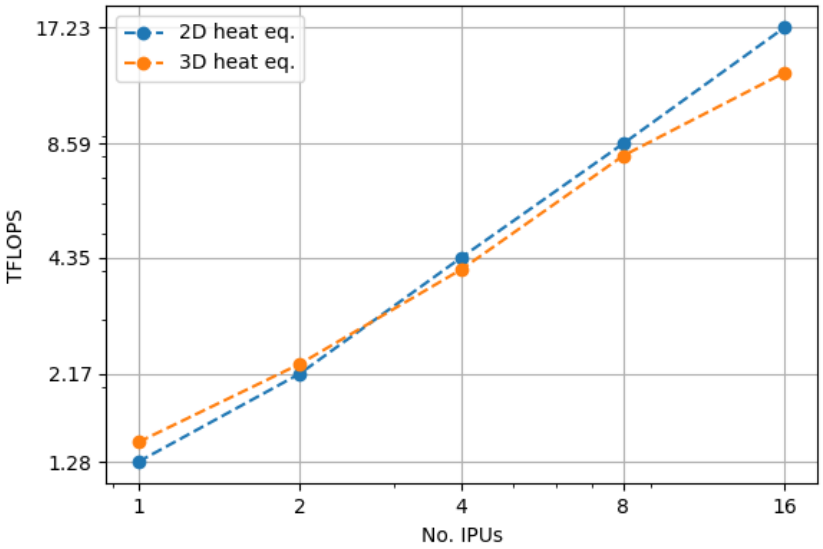
6.2 Results

The measured performance of all executions, including both the 2D and 3D heat equation solved on various number of IPUs are shown in Table 6.3.

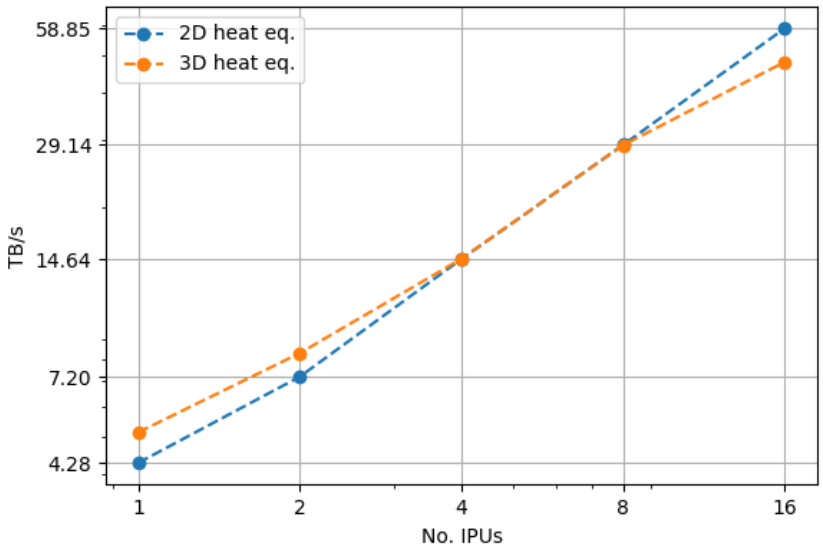
No. IPUs	Dimensions	Time	Throughput	Minimal Bandwidth
1	2D	0.30 s	1.32 TFLOPS	4.28 TB/s
2	2D	0.28 s	2.17 TFLOPS	7.20 TB/s
4	2D	0.27 s	4.35 TFLOPS	14.64 TB/s
8	2D	0.25 s	8.59 TFLOPS	29.14 TB/s
16	2D	0.25 s	17.23 TFLOPS	58.85 TB/s
1	3D	0.26 s	1.44 TFLOPS	5.15 TB/s
2	3D	0.23 s	2.30 TFLOPS	8.30 TB/s
4	3D	0.26 s	4.05 TFLOPS	14.67 TB/s
8	3D	0.26 s	7.99 TFLOPS	29.10 TB/s
16	3D	0.32 s	13.15 TFLOPS	48.05 TB/s

Table 6.3: The measured performance of executions solving the heat equation on the IPU, with an increasing number of IPUs as well as problem sizes.

To better illustrate the performance increases, Figure 6.1a shows the computational throughput vs. number of IPUs, and Figure 6.1b shows the minimal bandwidth vs. number of IPUs.



(a) Computational throughput



(b) Minimal memory bandwidth

Figure 6.1: The measured performance of the heat equation in both 2D and 3D, and for executions ranging from 1 to 16 IPUs. Note that all axes are scaled logarithmically.

The computational throughput per core was calculated by dividing by the number of cores (1,472) involved in each computation. This performance metric is shown in Figure 6.2.

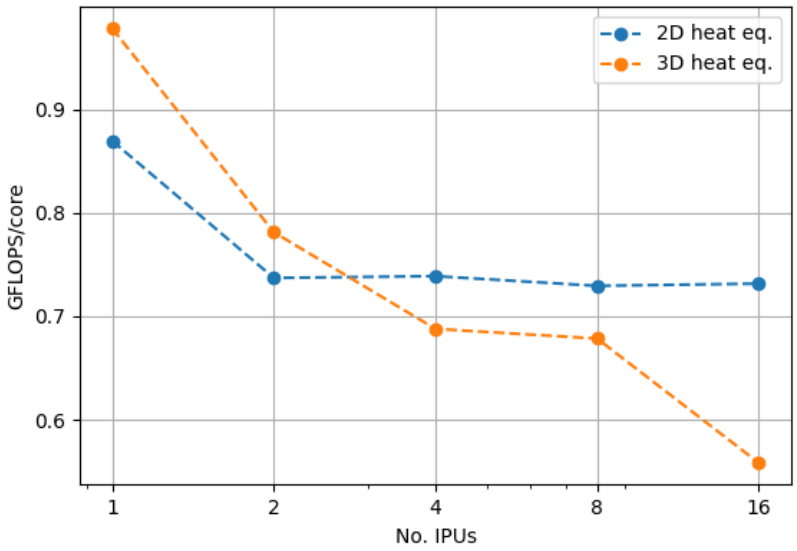
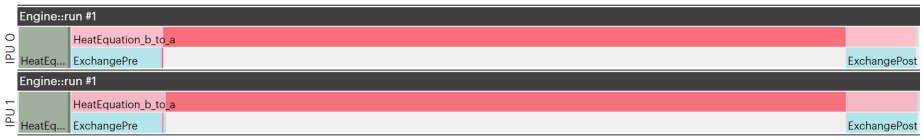
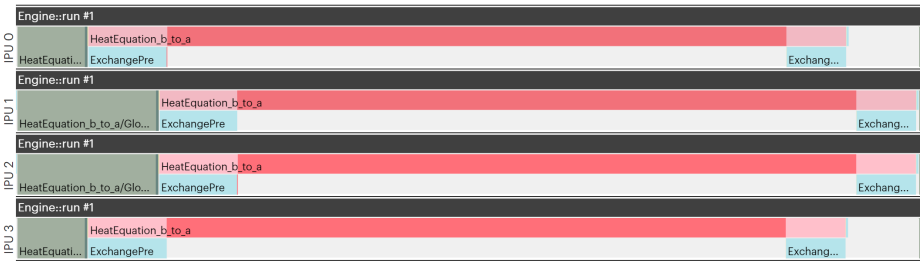


Figure 6.2: The computational throughput performance per computing core for executions of the 2D and 3D heat equation on 1 to 16 IPUs.

Next, screenshots from the *execution trace* view in PopVision are included. These illustrate the different BSP phases of the execution timelines. Figure 6.3a shows one time step of the 2-IPU execution, and Figure 6.3b shows one time step of the 4-IPU execution.



(a) BSP phases on 2 IPUs



(b) BSP phases on 4 IPUs

Figure 6.3: Screenshots from PopVision illustrating the BSP phases during one time step of the 3D heat equation executed on 2 and 4 IPUs, respectively. The timeline goes from left to right.

The colors in Figure 6.3 represent various execution phases: green is global exchange, red is compute, yellow is synchronization (not visible). The blue and pink represent internal exchange, i.e., tile-to-tile communication. PopVision groups together the blue tile-to-tile communication phases and the red computational phases because both execute independently on one tile. However, when the blue phases occur, the red becomes pink instead.

6.3 Discussion

The measurements in Figure 6.1 show how the performance scales with the number of IPUs. Both computational throughput and minimal memory bandwidth show that the heat equation scaled steeper for 2D compared to 3D. The latter achieved a noticeably higher performance on 1 and 2 IPUs. On 4 and 8 IPUs, the computational throughput was slightly higher in 2D, whereas the minimal bandwidth was almost identical. Lastly, on 16 IPUs, the 2D application outperformed its 3D counterpart. The overall trend of a steeper performance increase in 2D compared to 3D is expected, because the communication costs in 3D increases more than in 2D when scaling.

The computational throughput per core is shown in Figure 6.2. Starting at 1 IPU, this number is 0.90 GLOPS/core and 0.98 GFLOPS/core for 2D and 3D, respectively. The figure shows that the throughput per core decreases for all 3D executions, which is expected to be caused by exponentially increasing communication costs. On the other hand, the 2D heat equation shows a performance

decrease from 1 to 2 IPUs, which is caused by the introduction of global communication. However, from 2 to 16 IPUs, the 2D application roughly maintains the computational throughput per core. This result has positive implications for scaling 2D PDE solvers to execute on multiple IPUs.

Figures 6.3a and 6.3b show the BSP phases of executions on 2 and 4 IPUs, respectively. The time steps consist of global exchange (green), tile-to-tile exchange (blue), and compute (red). When contrasting these two figures, a communication imbalance becomes apparent. The code was set up to send and receive data between IPU i and IPU $i + 1$. This effectively means that the "outer" IPUs only have to communicate with one other IPU, whereas the "inner" IPUs must communicate with two IPUs. Therefore, the performance of chip-to-chip communication phases is bound by the amount of data that was communicated by the inner IPUs. The 2-IPU execution is balanced, whereas the 4-IPU execution features an imbalance. This imbalance will occur for all executions involving 4 or more IPUs. However, the relative performance loss is less significant for executions that involve a higher number of IPUs.

The work division is one of the key aspects to consider when developing multi-IPU codes. In this chapter, the top-level partitioning was only performed along one dimension. Another possibility was to divide the meshes into two dimensions. By using a 2D partitioning, the execution is expected to be more balanced for the higher number of IPUs, because the communication volume among all IPUs is more balanced. However, 1D partitioning among the IPUs was employed because the number of IPUs was relatively low.

The internal communication costs were minimized by using an algorithm, which minimized the internal communication volume. A disadvantage that the 3D problem exhibits can be seen in Table 6.1. The 3D problems feature larger workload imbalances than the 2D ones. This is caused by the fact that it becomes harder to evenly distribute the workload with an extra dimension. Due to the limited memory capacity, each tile can only fit a given problem size. In 2D, this max size was roughly in the range of 200×200 elements. For 3D however, the largest workloads per tile were only roughly $30 \times 30 \times 30$ elements. Since the 3D problem features significantly smaller side lengths and an additional dimension, one extra element per dimension will have a relatively larger impact on the workload balance.

Multi-IPU development involved an additional technical challenge compared to single-IPU development. For the 16-IPU executions, the compilation required more than 200 GB of host memory and several hours to complete. Larger executions of the 2D and 3D heat equation on 32 IPUs were attempted but were unsuccessful due to running out of host memory (755 GB) when the graph and programs were compiled at runtime. This could be caused by the demand of a very large number of communication messages, and the compiler likely faced a combination problem that skyrocketed the memory usage. It should be noted that the software ecosystem around the IPU, including compilers and run-time libraries, are under continuous improvement as the user-community

and the range of use cases expand. Therefore, the observed compilation cost for multi-IPU execution might be reduced in the future.

This chapter demonstrated that the performance of multi-IPU executions of the heat equation met the expectation of the scalable design principle. The software frameworks allowed the programming process to easily extend the implementation to run on multiple IPUs. This is valuable for general scientific computing. Further, a particularly interesting finding was that the 2D application featured a steeper performance increase than a 3D counterpart, which was likely caused by higher communication costs in 3D.

Lastly, some suggestions for future improvements are in order. For the heat equation, which now has been tested on multiple IPUs, it would be interesting to compare these results with multi-GPU executions of similar problems. Such comparisons could provide further insight into how the measured performances depended on IPU characteristics, and even more interestingly: which processor would achieve the best performance? It is also of interest to study other common scientific computing workloads, both on a single IPU, and how the performance would scale on multiple IPUs for such problems. Such studies could strengthen the findings in this chapter.

Chapter 7

The Aliev-Panfilov Model

This chapter demonstrates the IPU's applicability of a real-life physics-based application, namely the Aliev-Panfilov model of electrocardiology. The application in this chapter takes inspiration from a 2011-study by Hanslien et al. [33]. That study included two numerical schemes for solving the Aliev-Panfilov model, and in this chapter, one of those is implemented for the IPU.

7.1 Background

The Aliev-Panfilov model is a set of PDEs that model electric pulse propagation in cardiac tissue. It was constructed to include the qualitative behavior of cardiac tissue, while being computationally feasible. There is a *bidomain* model that accounts for both the transmembrane and extracellular electrical potentials, and a *monodomain* model that only includes the transmembrane potential. The monodomain Aliev-Panfilov model reads

$$\frac{\partial e}{\partial t} = \delta \nabla^2 e - ke(e-a)(e-1) - er, \text{ and} \tag{7.1}$$

$$\frac{\partial r}{\partial t} = -\left[\epsilon + \frac{\mu_1 r}{\mu_2 + e}\right][r + ke(e-b-1)]\,. \tag{7.2}$$

Here, e is the scaled transmembrane potential, r is a variable that represents recovery of the tissue, and a, b, μ_1, μ_2, k, ϵ, and δ are positive constants.

The *forward Euler* solution scheme for the Aliev-Panfilov model was derived by Hanslien et al. [33], and is given by

$$\frac{e_{i,j}^{n+1} - e_{i,j}^{n}}{\Delta t} = \delta \frac{e_{i+1,j}^{n} - 2e_{i,j}^{n} + e_{i-1,j}^{n}}{\Delta x^2} + \frac{e_{i,j+1}^{n} - 2e_{i,j}^{n} + e_{i,j-1}^{n}}{\Delta y^2} - k e_{i,j}^{n}(e_{i,j}^{n} - a)(e_{i,j}^{n} - 1) - e_{i,j}^{n} r_{i,j}^{n}, \tag{7.3}$$

$$\frac{r_{i,j}^{n+1} - r_{i,j}^{n}}{\Delta t} = -\left[\epsilon + \frac{\mu_1 r_{i,j}^{n}}{\mu_2 + e_{i,j}^{n}}\right]\left[r_{i,j}^{n} + k e_{i,j}^{n}(e_{i,j}^{n} - b - 1)\right]. \tag{7.4}$$

Equation (7.3) represents a 5-point stencil, which is undefined at the boundary nodes. The *zero gradient* boundary condition was employed, given by $\vec{n} \cdot \delta \nabla e = 0$, where $\vec{n}$ is a unit vector, orthogonal to the boundary. The boundary condition is satisfied when two opposing neighbors in a stencil are equal, because that translates to a zero gradient in the direction they lie along. For instance, on the east-edge boundary, the right stencil-element $e_{i,j+1}^{n}$ is undefined. However, if this element is replaced with the left stencil-element, $e_{i,j-1}^{n}$, the boundary condition is satisfied.

The constants were set to the same values as in the study of Hanslien et al. [33]. They were $\mu_1 = 0.07$, $\mu_2 = 0.3$, $k = 8.0$, $\epsilon = 0.01$, $a = b = 0.1$, and $\delta = 0.00005$. Further, the distance between the points was set to $h = 1/7000$, and the time step was set to $\Delta t = 0.0001$.

Bounds on the Numerical Method

The forward Euler scheme suffers from a time step restriction [33]. Given a uniform 2D mesh where $\Delta x = \Delta y$, the two equations

$$0 \leq e_{i,j}^{n} \leq 1, \text{ and} \tag{7.5}$$

$$0 \leq r_{i,j}^{n} \leq r_{+} \tag{7.6}$$

must be satisfied, where,

$$r_{+} = k\left(\frac{b+1}{2}\right)^2. \tag{7.7}$$

It was showed by Hanslien et al. [33] that equations (7.5) and (7.6) were satisfied when

$$\Delta t \leq \min\left\{\frac{1}{\frac{4\delta}{\Delta x^2} + \max(ka, k(1-a)) + r_{+}}, \frac{1}{\epsilon + \frac{\mu_1}{\mu_2} r_{+}}\right\}. \tag{7.8}$$

With the constants used, this becomes

$$\Delta t \leq \min\{0.000102, 1.74\} = 0.000102. \tag{7.9}$$

The bound on Δt was incorporated as a test function in the code.

7.2 Methods

7.2.1 IPU Implementation

The numerical algorithm was the implementation of equations (7.3) and (7.4). This section presents new challenges of the implementation of the Aliev-Panfilov model compared to the 2D heat equation.

The Aliev-Panfilov model requires two different meshes, e and r, unlike the heat equation, which only involves one. Since the right-hand side of equation (7.3) represents a 5-point stencil, e cannot be updated in-place. Therefore, two tensors named `e_a` and `e_b` were allocated. The r mesh on the other hand, can be updated in-place, and was therefore represented by one tensor.

The 2D meshes contained 7000×7000 single precision floating-point elements. The work division was found by using an algorithm that minimizes the communication volume. This was the same algorithm as for the 2D heat equation in Listing 4.1. This resulted in 46×32 partitions. Consequently, the tiles were assigned workloads of different sizes between 152×218 and 153×219 elements. Hence, some tiles had 1.1% larger workloads than the tiles with the smallest workloads.

The zero gradient boundary condition on e had to be implemented. The two tensors representing e were padded with one element on all four sides. This method presents a way of handling the boundary condition outside the vertex, and without changing the stencil. The boundary nodes were set equal to the immediate inner nodes prior to each time step:

- For the north boundary, copy the values of the third row to the first.
- For the south boundary, copy the values of the third last row to the last.
- For the west boundary, copy the values of the third column to the first.
- For the east boundary, copy the values of the third last column to the last.

The computations was still only be performed on the inner mesh of 7000×7000 elements. However, the introduced padding is used to satisfy the boundary condition. To recall, when two opposing neighbors in a stencil are equal, the net gradient is zero in the direction they lie along.

Listing 7.1 shows the codelet solving the Aliev-Panfilov model.

```
#include <poplar/Vertex.hpp>

using namespace poplar;

class AlievPanfilov : public Vertex {
public:
  AlievPanfilov();

  Vector<Input<Vector<float, VectorLayout::SPAN, 4, false>>> e_in; // padded
  Vector<Output<Vector<float, VectorLayout::SPAN, 4, false>>> e_out;
  Vector<InOut<Vector<float, VectorLayout::SPAN, 4, false>>> r;
  const int worker_height;
  const int worker_width;
  const float delta;
  const float epsilon;
  const float my1;
  const float my2;
  const float h;
  const float dt;
  const float k;
  const float a;
  const float b;

  bool compute () {
    const float d_h2 = delta/(h*h);
    for (int i = 1; i < worker_height + 1; ++i) {
      for (int j = 1; j < worker_width + 1; ++j) {
        // Computation of new e
        e_out[i-1][j-1] = e_in[i][j] + dt*(
          d_h2*(-4*e_in[i][j] + e_in[i+1][j] + e_in[i-1][j] + e_in[i][j+1] +
          ↪  e_in[i][j-1]) -
          k*e_in[i][j]*(e_in[i][j] - a)*(e_in[i][j] - 1) -
          ↪  e_in[i][j]*r[i-1][j-1]
        );

        // Computation of new r
        r[i-1][j-1] += dt*(-epsilon - my1*r[i-1][j-1]/(my2 + e_in[i][j]))*
          (r[i-1][j-1] + k*e_in[i][j]*(e_in[i][j] - b - 1));
      }
    }
    return true;
  }
};
```

Listing 7.1: A codelet that solves the Aliev-Panfilov model. In total, the inner loop body involves 28 floating-point operations, and at least 24 bytes are loaded and 8 bytes stored.

Listing 7.1 contains the vertex type `InOut` for the first time in this thesis. This type is convenient as it gives the vertex the ability to both read from and write to a tensor. When compared to the 2D heat equation, the Aliev-Panfilov model involves a lot more arithmetic operations compared to the heat equation, which does not increase the complexity much. Additionally, the model involves the

usage of two fields instead of one, which does somewhat increase the complexity.

The wall time of an execution of 1000 time steps was measured by using the *chrono* tools in C++. With this, the computational throughput was calculated. This is a measure of the number of floating-point operations per second (single precision), and it was calculated by counting 28 floating-point operations per stencil per time step, giving

$$throughput = 28\frac{(area)(no.\ time\ steps)}{measured\ wall\ time}. \tag{7.10}$$

A second metric that was calculated by using the measured wall time, was the minimal memory bandwidth. This represents the minimal amount of memory traffic that must have taken place during the execution. It was calculated by adding together the number of memory operations in the communication phases and in the computational phases. This number was in turn multiplied by 4 bytes and number of time steps, and divided by wall time. As a summary, the minimal bandwidth was found by

$$communication\ ops. = (2\ ops.)(communication\ volume\ ), \tag{7.11}$$

$$computation\ ops. = (8\ ops.)(area), \tag{7.12}$$

$$minimal\ bandwidth = (comm.\ ops. + comp.\ ops.)\ \frac{(4\ bytes)(no.\ time\ steps)}{(measured\ wall\ time)}. \tag{7.13}$$

The number of communication phase memory operations was 2 times the communication volume, because it accounts for both storing and loading this data. Further, the number of computational phase memory operations was 8 times the area of the mesh, because each stencil must load 6 and store 2 elements (accounting for both e and r).

The codes were compiled with Poplar SDK 2.2.0, GCC 7.5.0, and the -O3 optimization flag.

7.2.2 CPU Implementation

The Aliev-Panfilov model was implemented in standard C along with *OpenMP* to serve as a benchmark for the IPU code. Three double-pointer float arrays were used, two to represent e and one for r. However, e was padded in order to handle the boundary condition, in the same manner as for the IPU code.

A snippet of the CPU code is shown in Listing 7.2.

```
#pragma omp parallel private(i,j,t)
{
  // Perform Forward-Euler Aliev-Panfilov model
  for (t = 0; t < num_iterations; ++t) {
    // Handle boundary conditions west and east
    #pragma omp for
    for (i = 1; i < height - 1; ++i) {
      e[i][0] = e[i][2];
      e[i][width - 1] = e[i][width - 3];
    }

    // Handle boundary conditions north and south
    #pragma omp for
    for (j = 1; j < width - 1; ++j) {
      e[0][j] = e[2][j];
      e[height - 1][j] = e[height - 3][j];
    }

    // Perform stencil computations
    #pragma omp for
    for (i = 1; i < height - 1; ++i) {
      for (j = 1; j < width - 1; ++j) {

        // Computation of new e
        e_bar[i][j] = e[i][j] + dt*(
        d_dx2*(-4*e[i][j] + west + east + south + north) -
        k*e[i][j]*(e[i][j] - a)*(e[i][j] - 1) - e[i][j]*r[i][j]
        );

        // Computation of new r
        r[i][j] = r[i][j] + dt*(-epsilon - my1*r[i][j]/(my2 +
        ↪  e[i][j]))*(r[i][j] + k*e[i][j]*(e[i][j] - b - 1));
      }
    }

    // Pointer swap
    #pragma omp single
    {
      tmp = e_bar;
      e_bar = e;
      e = tmp;
    }
  }
}
```

Listing 7.2: A snippet of the OpenMP CPU code, which allows for a multi-threaded execution. The code solves the Aliev-Panfilov model by the forward Euler scheme.

The wall time of the execution was measured by using OpenMP's built-in wall time tool. This number was used to calculate the computational throughput and minimal memory bandwidth. The throughput was calculated similarly as for the IPU code, shown by equation (7.10).

The minimal memory bandwidth was calculated differently for the CPU code, because the two processors feature very different memory architectures. For the CPU execution, the minimal bandwidth was calculated by considering that the both meshes must be loaded and stored at least twice, which gives

$$\textit{minimal bandwidth} = \frac{(4\ \textit{ops.})(4\ \textit{bytes})(\textit{area})(\textit{no. time steps})}{\textit{measured wall time}}. \tag{7.14}$$

The CPU code was compiled with GCC 11.1.0, along with OpenMP 4.5, and the -O3 optimization flag. The code was executed on a Linux server with two AMD Epyc 7601 (Naples) 32-core CPUs, totalling 128 threads.

7.3 Results

Table 7.1 shows the measured performance of the IPU and CPU executions of solving the Aliev-Panfilov model the same problem size.

Processor	Time	Throughput	Minimal Bandwidth
CPU	20.49 s	66.94 GFLOPS	19.13 GB/s
IPU	1.086 s	1.26 TFLOPS	1.45 TB/s

Table 7.1: The performance for solving the Aliev-Panfilov model on 2D meshes of 7000×7000 single precision elements, for 1000 time steps.

7.4 Discussion

The performance of the IPU implementation, shown in Table 7.1, shows a promising result. The computational throughput of the IPU execution was 19 times higher than the CPU execution. If adjusted to per-core, an IPU core achieved on average 82% of the throughput of a CPU core. The results were computed significantly faster on the IPU. Furthermore, the IPU reaches 0.86 GFLOPS/core, and the CPU reaches 1.05 GFLOPS/core. The performance per core suggests that an IPU core can almost provide as high performance as a CPU core, which is remarkable. Compared to the CPU, the IPU has a big advantage in its large number of cores, but additionally does not compromise much on the performance per core.

The measured minimal memory bandwidth on the IPU was 1.45 TB/s. The computational phases contributed to 99.5% of this number, whereas the communication phases only accounted for 0.5%. Compared to the CPU execution, the IPU reached 76 times higher minimal memory bandwidth. However, using the minimal bandwidth to compare the two processors might not be suitable, because they feature very different memory architecture designs. Regardless,

the result undoubtedly shows that the IPU can achieve a very high memory bandwidth, even for an application that arguably is bound by arithmetic operations. This suggests that the low-latency in-processor memory model of the IPU can be suitable for a physics-based computational workload.

Implementing a parallel program to solve the Aliev-Panfilov model was more technically challenging in *Poplar*. This is because the meshes and workloads must be partitioned to all the tiles involved. In the *OpenMP* framework on the other hand, the work is automatically distributed by allowing different workers (CPU threads) to handle different indices of the loop.

The Aliev-Panfilov model simulates a much more complicated set of equations compared to e.g., the heat equation. Hence, these results demonstrate that the IPU can provide high performance to more complex applications, borrowed from general scientific computing.

Lastly, a suggestion for future improvements is presented. An interesting class of computational workloads for general scientific computing is sparse matrix-vector multiplications. These operations are central to many physics-based applications. It would be particularly interesting to benchmark the performance of such computations on the IPU. The results of such applications could strengthen the results in this thesis to hold for a wider variety of scientific applications.

Chapter 8

Conclusion

A selection of general scientific computing workloads was performed on the IPU. The heat equation, discretized by finite differences, was solved for structured 2D and 3D meshes, by parallel computations scaling from 1 to 16 IPUs. Additionally, a real-life application was implemented and solved on the IPU, namely the Aliev-Panfilov model for cardiac excitation.

There were four research questions proposed in Chapter 1:

1. How good performance can the IPU provide for the selected scientific computing workloads?
2. What technical challenges does low-level IPU programming involve?
3. What limitations does the IPU exhibit for general scientific computing workloads?
4. What differences and similarities are found when asking the three aforementioned questions to multi-IPU systems compared to single-IPU systems?

Answer to Research Question 1: The selected computational workloads were solved on IPU and CPU systems, by setting up identical problems with single precision floating point elements. The IPU significantly outperformed the CPUs for all computations, even providing a higher rate of FLOPS per core in one computation. The 3D heat equation achieved the highest performance on one IPU, reaching a minimal memory bandwidth of 5.15 TB/s and a throughput of 1.44 TFLOPS. For the 2D heat equation, 3D heat equation, and the Aliev-Panfilov model, the IPU achieved 14, 35, and 19 times more FLOPS than the corresponding CPU computations, respectively. If adjusted to a per-core performance, this corresponds to 0.59, 1.51, 0.82 times, respectively. This shows that one IPU-core reached lower performances than one CPU-core for the 2D

applications, but remarkably higher performance for the 3D application. Additionally, the IPU's architecture exhibits the advantage of containing 1,472 cores, which is in the same order of magnitude as some GPUs. This suggests that the IPU, which was mainly designed for AI, can provide high performance for selected scientific computing applications.

Answer to Research Question 2: The development process of IPU programs is built around two well-established concepts in parallel programming: BSP and computational graphs. Using Poplar, most of the effort in developing programs for the IPU lies in constructing the graph object and writing the vertices. The programmer has a very fine-grained control over the execution and has the possibility to implement MIMD programs. The graph-based framework implicitly handles communication and data dependencies, which is an advantage with regards to the simplicity of writing IPU programs. A programmer that is used to shared memory models might have to reassess how they approach parallel programming when coding for the IPU. A distributed memory model imposes additional challenges. In addition to partitioning the workload, the data must also be distributed. Furthermore, the partitioning and data distribution must be optimized with respect to each other in order to achieve high performance.

Answer to Research Question 3: The limited memory capacity is the main limitation of the IPU. The distributed in-processor memory is the cornerstone for IPU applications, and it imposes strengths and weaknesses. On the positive side, the architecture was demonstrated to be able to provide remarkably high memory bandwidths. This is likely due to the memory components being in the vicinity of each core, which combined makes a tile, and the large number of tiles. This can undoubtedly be a significant advantage for many applications. A potential negative aspect of the architecture is that the memory has a limited capacity of 900 MB per chip, which translates to 624 kB per tile. Therefore, if an application has a memory requirement that exceeds the capacity of an IPU, the programmer is left with two choices. One could either resort to taking advantage of off-chip memory which will considerably reduce performance, or one could scale the application to include enough IPUs to fit the problem.

Answer to Research Question 4: The IPU is designed for scalability by being able to provide higher performance in larger multi-IPU systems for AI workloads [7]. However, this thesis specifically intended to study the scalability of the IPU for non-AI workloads. When scaling the heat equation to multi-IPU systems, the 2D heat equation showed a steeper performance increase than its 3D counterpart. However, the performance scaled well with the number of IPUs for both applications. Therefore, the results for these workloads were consistent with the IPU's design principle for AI workloads. The scaling of the implementation from single to multi-IPU executions was effortless, due to the purposeful design of the Poplar framework. The main challenge in these extensions was to optimally partition the problems. There are two clear weaknesses of multi-IPU development. The first weakness is the limited memory. The selected applications demanded that each tile must communicate with several others, which

in turn imposes a need for more communication buffers on the in-processor memory. This puts additional pressure on the already limited memory capacity. Effectively, this means that the size of the application, e.g., resolution of the meshes, cannot be scaled linearly with the number of IPUs involved. The second weakness is the surprisingly high demand for resources when the graph and Poplar programs are compiled at runtime. However, the software ecosystem around the IPU is under continuous improvement. Therefore, the observed compilation cost for multi-IPU execution might be reduced in the future.

As closing words, some suggestions for future improvements are in order. One of the overarching intentions of this thesis was to study the performance and challenges of using a specialized processor for general scientific computing workloads. Only stencil-based algorithms for solving PDEs of varying complexity have been tested. This only includes a narrow selection of scientific applications, and it would be of interest to test other common computational workloads. For instance, in physics and astronomy, many-particle simulations are common computational problems that demand high performance. Another interesting use-case is sparse matrix-vector multiplications, which is a central computational operation in many physics-based applications. If the IPU achieves promising results in such applications, it would strengthen the findings of this thesis by extrapolating them to a wider variety of applications. Lastly, it would be interesting to benchmark the results in this thesis against state-of-the-art GPU computations of similar problems. For a fair comparison, this would require significant work on optimization of the GPU code, for instance in CUDA. This would be out of scope for this thesis, which is dictated by the given time limitation for the master's project.

Bibliography

[1] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey of machine learning accelerators. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–12, 2020.

[2] IPU processors. `https://www.graphcore.ai/products/ipu`, 2021. Accessed: 2021-07-08.

[3] About us. `https://www.graphcore.ai/about`, 2021. Accessed: 2021-07-08.

[4] Kenneth G. Wilson. Grand challenges to computational science. *Future Generation Computer Systems*, 5(2):171–189, 1989. Grand Challenges to Computational Science.

[5] Anupama Luthra. *Computational Science and its Applications*. 04 2020.

[6] Research papers. `https://www.graphcore.ai/resources/research-papers`, 2021. Accessed: 2021-07-13.

[7] Performance results. `https://www.graphcore.ai/performance-results`, 2021. Accessed: 2021-09-23.

[8] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore IPU architecture via microbenchmarking. *CoRR*, abs/1912.03413, 2019.

[9] Thorben Louw and Simon McIntosh-Smith. Using the graphcore IPU for traditional hpc applications. EasyChair Preprint no. 4896, EasyChair, 2021.

[10] Andrew Danowitz, Kyle Kelley, James Mao, John P Stevenson, and Mark Horowitz. CPU DB: recording microprocessor history. *Communications of the ACM*, 55(4):55–63, 2012.

[11] P. Gepner and M.F. Kowalik. Multi-core processors: New way to achieve high system performance. In *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, pages 9–13, 2006.

[12] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. 2006.

[13] Chris McClanahan. History and evolution of GPU architecture. *A Survey Paper*, 9, 2010.

[14] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[15] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. GPGPU processing in CUDA architecture. *CoRR*, abs/1202.4347, 2012.

[16] R.R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, 1997.

[17] AI and compute. `https://openai.com/blog/ai-and-compute/#fnref1`, 2018. Accessed: 2021-07-13.

[18] A.A. Khokhar, V.K. Prasanna, M.E. Shaaban, and C.-L. Wang. Heterogeneous computing: challenges and opportunities. *Computer*, 26(6):18–27, 1993.

[19] Howard Jay Siegel, John K Antonio, Richard C Metzger, Min Tan, and Yan Alexander Li. Heterogeneous computing. *ECE Technical Reports*, page 206, 1994.

[20] Olivier Terzo, Karim Djemame, Alberto Scionti, and Clara Pezuela. *Heterogeneous Computing Architectures: Challenges and Vision*. CRC Press, 2019.

[21] Graphcore Ltd. *IPU-MACHINE: M2000 Datasheet*, dec 2020.

[22] Graphcore Ltd. *IPU-M2000 SCALE-OUT: IPU-POD64 Datasheet*, dec 2020.

[23] Graphcore Ltd. *Vertex Programming Guide*, jun 2021.

[24] Preeti Ranjan Panda, Nikil D Dutt, and Alexandru Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3):682–704, 2000.

[25] Graphcore Ltd. *IPU Programmer's Guide*, oct 2020.

[26] Rob H Bisseling and William F McColl. Scientific computing on bulk synchronous parallel architectures. 1993.

[27] David Ojika, Bhavesh Patel, G. Anthony Reina, Trent Boyer, Chad Martin, and Prashant Shah. Addressing the memory bottleneck in AI model training. *CoRR*, abs/2003.08732, 2020.

[28] Poplar graph framework software. `https://www.graphcore.ai/products/poplar`, 2021. Accessed: 2021-07-15.

[29] Graphcore Ltd. *Poplar and PopLibs User Guide*, mar 2021.

[30] Graphcore Ltd. *Poplar and PopLibs API Reference*, mar 2021.

[31] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[32] John D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[33] Monica Hanslien, Robert Artebrant, Aslak Tveito, Glenn Terje Lines, and Xing Cai. Stability of two time-integrators for the Aliev-Panfilov system. *International Journal of Numerical Analysis & Modeling*, 8(3), 2011.