Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Exploiting Graphcore IPU for Finite Difference Stencil Computation

Author:
Ko, Tsz Wang

Supervisor:
Prof. Paul, Kelly
Dr Bisbas, George

Second Marker:
Prof. Casale, Giuliano

June 23, 2023

Abstract

Stencil computation is essential in numerically solving Partial Differential Equations (PDE). It is often used in engineering context and presents a major opportunity for optimisation due to the vast amount of computation it requires. Improving the time required to solve stencil computation has huge potential to enable new applications. Improvement of hardware is one of the main contributors to improving stencil computation. This project investigates the potential of the Intelligent Processing Unit (IPU) in improving stencil computation.

The Intelligent Processing Unit (IPU) is a hardware architecture created by Graphcore. Its unique hardware architecture enables a combination of different parallelism techniques to be employed. On top of the traditional Single Instruction Multiple Data (SIMD), the fine-grained threaded model employed on modern multi-core CPUs, the fabric of tiles in a Graphcore IPU allows programs to be subdivided into subprograms, effectively enabling Multiple Instruction Multiple Data (MIMD), so that further parallelism can be achieved.

Its uniqueness lies in the close proximity between the processor and its memory, which allows for extremely fast memory read and write. This project shows that for the diffusion computation on mesh with elements in the scale of a hundred million elements, the IPU is able to outperform all alternative hardware. The performance of the IPU is 59.5% better than the next best candidate, the *NVIDIA RTX A6000*. With a more complex wave equation, however, the IPU performs 3.7% worst than *NVIDIA RTX A6000*. The generalisation of the result to larger problems with different operational intensities is left as future work.

The experiments revealed IPU's notable weak and strong scaling capabilities. However, it was observed that the weak scaling capability is lacking when dealing with high memory density problems. At the same time, space order is also found to have a greater influence on execution time than the total problem size.

The project also highlights the challenges in coding an IPU machine. Specifically, the lack of native support for higher-order arrays in the IPU *poplar* programming framework makes programming for higher-order tensors challenging. Moreover, there is a notable increase in control flow overhead as the time order of the problem grows, and the compile time exhibits poor scalability with increased space order.

Acknowledgements

I would like to express my gratitude to Prof. Paul Kelly for his mentorship and patience in helping me overcome numerous hurdles throughout this journey. I am very grateful for the opportunity to collaborate with the talented individuals in the Devito Compiler Group.

I am particularly indebted to George Bisbas for his exceptional guidance and patience in mentoring me, as well as for providing me with relevant resources that have propelled the progress of this project. Our discussions have immensely broadened my understanding of the landscape of High-Performance Computing and highlighted the significance of the Devito Compiler.

I would also like to extend my thanks to Neel Dugar and Junhyuk Bae for engaging in insightful discussions about their respective projects on GPU and Cerebras. These interactions have deepened my comprehension of the distinctions between the IPU, GPU, and Cerebras hardware architectures and programming models.

Furthermore, I would like to acknowledge the groundwork laid by Thorben Louw, Simon McIntosh-Smith, and Simen Hipnås from the University of Bristol and the University of Oslo, respectively, in relation to the IPU, which has greatly benefited this project.

Lastly, I would like to express my appreciation to Mark Pupilli and the compiler team from Graphcore for their prompt and helpful responses to the challenges I encountered while programming the IPU.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Stencil computation is a widely utilised technique for numerically solving Partial Differential Equations (PDEs). It holds great significance in engineering contexts, particularly in seismic applications, and optimising stencil computation can greatly enhance the performance of various applications. Leveraging the hardware capabilities of modern computing systems is a crucial approach to achieving efficient stencil computation.

The Intelligence Processing Unit (IPU) is a recently developed hardware by Graphcore [6] that initially targeted Machine Learning applications. However, its remarkable parallel computing capabilities make it a potential candidate for High-Performance Computing (HPC) as well.

As complicated hardware architecture is often accompanied by complex programming models, it often requires a large number of manual hours to implement solutions on novel hardware. To solve this problem, Domain-Specific Language (DSL) was developed as a sort of high-level compiler. It allows users to generate optimised stencil code specifically tailored for the underlying hardware architecture without requiring in-depth knowledge of it. An example of such a DSL is *Devito*. With a language similar to the *SymPy* library in Python, *Devito* provides a user-friendly interface to specify the PDEs to be solved while generating highly optimised code. Moreover, the *Devito* compiler offers users access to its backend, enabling developers with expertise in hardware architecture to obtain optimised stencil code, which proves invaluable for those lacking expertise in physics and mathematics required for manual problem discretisation.

1.1 Objective

This project aims to explore the potential of utilising the IPU in HPC applications and to inform whether the IPU is a potential candidate to be incorporated in the *Devito* compiler.

To achieve this, two Partial Differential Equations, namely the 3D diffusion equation and wave equation, were tested on the IPU. Specifically, these equations are discretised on the *Devito* compiler using the *Finite Difference* method. The stencil equation is obtained from the back end of the compiler and then implemented on the IPU.

The diffusion equation represents one of the most extensively studied stencil equations for solving PDEs using the *finite difference* method [7]. As a result, it serves as an effective benchmark for evaluating hardware performance in HPC applications. additionally, the diffusion equation also provides grounds for exploring the relationship between increased space order and performance throughput on the IPU. On the other hand, the wave equation represents a more realistic use case of stencil computation commonly employed in seismic imaging. It is one of the equations supported by the *Devito* compiler [8]. As the wave equation requires a time order of at least 2, it enables exploring IPU performance in solving stencil equations of higher time order.

The performance of the IPU in calculating these equations is compared against alternative hardware architectures such as CPU and GPU. This comparison aims to assess the position of IPU hardware architecture within the current stencil computation landscape. In addition, the strong and weak scaling capability of the IPU is also explored by increasing the number of IPU units in solving

fixed and increase-sized problems involving diffusion and wave stencil. At last, the challenges in the development process on the IPU are also documented to provide a picture of the software development efforts needed in developing programs for the IPU machine.

1.2 Contributions

Past analysis on the IPU has been conducted by Thorben Louw, Simon McIntosh-Smith [5], and Simen Håpnes [9] by benchmarking it on HPC problems involving 3D stencils. The project aims to build upon their works and contribute in the following categories:

1. Evaluate the performance of the IPU on stencil computation in relation to other hardware
2. Benchmark the IPU on stencil computation in relation to space and time order
3. Investigate weak and strong scaling capabilities of the IPU
4. Discuss the advantages and disadvantages of creating a *Devito* backend for the IPU.

1.3 Conclusion

The investigation of the IPU's performance in stencil computation has yielded compelling results. In solving diffuse and wave equations with a scale of hundreds of millions of elements and a space order of four, the IPU excelled over commercially available alternative hardware solutions. Notably, the IPU outperforms the next best candidate, NVIDIA A6000, by 59.5% on the diffusion equation but loses to it by 3.7% on the wave equations.

The weak scaling capability of the IPU is noteworthy, as the execution time for nearly double-sized problems remains largely unchanged when the number of IPUs is doubled. This desirable weak scaling characteristic indicates the IPU's ability to handle larger problems efficiently. However, it should be noted that doubling the number of IPUs does not necessarily double the problem size that can be accommodated, as additional logic is required to manage the increased data movement associated with larger-sized problems.

Furthermore, the IPU exhibits strong scaling capability when multiple IPUs are utilized to perform calculations on the same-sized diffusion problem. The performance throughput increases by 81.1% from one to two IPUs and by 63.6% from two to four IPUs. However, the strong scaling capability of the IPU is less pronounced in the wave equation. The performance throughput increases by 87.4% from one to two IPUs but only by 38.3% from two to four IPUs.

The investigation reveals a notable observation regarding the impact of increased space order on IPU execution time. Specifically, it has been observed that the execution time of the diffusion problem increases with the space order of the problem. This indicates that the diffusion problem's execution time is limited by the space order rather than the problem size, assuming a proportional number of IPUs to support the calculation. Consequently, it can be inferred that the IPU is I/O bound for the diffusion problem, as the execution time correlates with the amount of communication required.

While the time ordering also increases communication volume, it affects the IPU in other more significant ways. Increased time order requires additional memory buffers to store extra iterations of the mesh. As the memory buffers represent different time instances in the wave stencil equation at each iteration, each memory rotates to represent different time instances when iterating through the stencil. To optimize the control flow of the problem, the IPU programming framework favours complete memory rotation at each program iteration. Consequently, the iteration of the stencil is performed in batches, requiring additional program logic to handle cases where the iteration is not divisible by the batch size. Therefore, an increase in time order adds overhead and complexity to programming the control flow of the problem.

One of the primary challenges in programming the IPU lies in the extensive compilation time and substantial memory usage during the compilation process. The IPU compiler automates many of the exchange codes required for communication between tiles. This reduces the burden on the programmer but results in significant compilation time for IPU programs, particularly for problems

with higher space orders. In an effort to mitigate this challenge, multiple threads are utilised during compilation, albeit at the expense of considerable memory usage.

To compile a program that fully utilises the memory of four IPUs, the compilation process can take several hours and consume approximately 100 GB of memory. Additionally, there are no effective means of estimating the amount of exchange code generated by the compiler and, consequently, the maximum memory that an IPU can accommodate for a given problem. As a result, programmers are incentivised to underestimate the maximum memory an IPU can accommodate significantly.

1.4 Report Structure

Chapter 2 provides an overview of the project's background knowledge, encompassing the IPU architecture, programming model, introduction to the *Devito* Framework, and relevant mathematical terminologies.

Chapter 3 offers a comparison between the IPU and other hardware options, namely the Wafer Scale Engine (WSE)[10] and GPU. Additionally, the chapter offers an overview of the works by Thorben Louw, Simon McIntosh-Smith [5], and Simen Håpnes [9], which serve as the foundation for this project.

Chapter 4 and 5 showcase the implementation of the stencil equations using the *Devito* compiler. The chapters detail the process of adapting the solutions to the IPU machines and present the necessary modifications to Simen Håpnes's implementation [9] to accommodate higher space or time order. Furthermore, a validation check is performed by comparing the results of the implemented IPU program with those generated by the *Devito* compiler.

Chapter 6 conducts experiments utilizing the implemented stencil equations to evaluate the IPU's performance compared to other hardware options. The chapter also assesses the weak and strong scalability of the IPU.

Chapter 7 draws conclusions from the project, discusses the limitations of the experiments conducted, and suggests potential avenues for future work.

Chapter 2

Background

This section aims to provide the necessary background information and terminologies to provide grounds for discussion for the following Chapters.

The structure of this chapter is as follows. Section 2.1 introduces the Graphcore IPU by providing an overview, followed by a description of the hardware architecture and its programming model. Section 2.2 gives an overview of the *Devito* compiler, explains the internal compiler structure, and provides an example to demonstrate the development process using the *Devito* Domain Specific Language (DSL). At last, Section 2.3 defines the related terminologies used in this project.

2.1 Intelligence Processing Unit

The Intelligent Processing Unit (IPU) is a highly parallel processing unit that is designed to help alleviate computational loads from the host computer. Its memory architecture consists of large amount of In-Processor-Memory composed of SRAM for fast memory access as well as a significant amount of streaming memory chip composed of DRAM which is used for memory transfers to and from the the host CPU. Although it's designed to target machine learning purposes, its highly parallel architecture is also suitable for other immensely repetitive tasks such as solving Partial Differential Equations.

This section provides an overview of the IPU structure and establishes the necessary terminology for the subsequent chapters.

2.1.1 Hardware Overview

The IPU comprises two generations: the *Colossus GC2 MK1* and the *Colossus GC200 MK2*. The focus of this project revolves around the latter, which represents the latest generation of the IPU.

The *MK2* IPU is composed of 1472 tiles, with a total of 900MB In-Processor-Memory. As shown in Figure 2.1, Each tile contains both a on-tile processor and 624kB of disjoint memory and can only execute code within local memory. All the tiles are interconnected through a low-latency all-to-all communication network with a high bandwidth of 8TB/s. Additionally, 10 IPU-Links with a bandwidth of 320GB/s facilitate fast communication between multiple IPUs. This architectural design enables scalability by utilizing multiple IPUs in tandem, allowing for efficient handling of larger problem sizes.

2.1.2 Tile Architecture

Within each of the tiles contains a computing core and 624kB of local In-Processor-Memory. The unique characteristic of this architecture is its distributive memory model, where each tile gets its own local processor and In-Processor-Memory. This is in contrast to the shared memory model used by CPUs and GPU. As the memory is only accessible by the local processing core within its compute phase, the IPU is largely immune to issues like data races. With the In-Processor-Memory capable of storing both executable code and variables, each tile can be programmed to execute

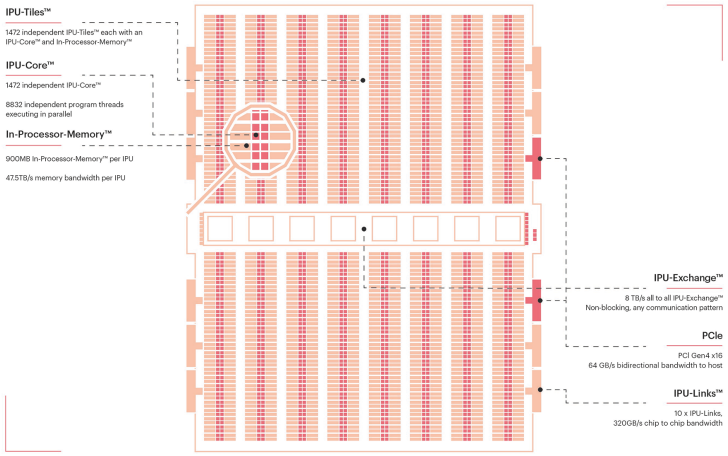


Figure 2.1: The IPU Tile Structure. Figure from [1]

different computations concurrently. Consequently, the IPU functions as a multiple instruction, multiple data (MIMD) processor.

Computing Core

Each computing core in each tile is capable of handling multiple threads. The threads in each tile are categorised by 2 categories: supervisor and worker threads. Supervisor threads is responsible for overseeing processes and spawning worker threads. A maximum of 6 worker threads can be created in a tile, and they execute in a round robin fashion. When all 6 slots are occupied, the supervisor thread is suspended until one of the work thread completes its tasks and frees up on of the execution slots.

Each tile contains a pair of asymmetric execution pipelines, *main* and *aux*. While workers can use both, supervisor threads can only use the main pipeline. The two pipelines have distinct roles:

- *main* performs control flow, address manipulation, integer arithmetic and load/store operations
- *aux* performs floating-point based compute

Instructions can be issued to either of the pipelines or both. The term *Execution Bundle* refers to instructions that are issued in parallel to both pipelines. Maximizing the utilization of *Execution Bundles* is advantageous as it enables concurrent memory access and floating-point operations.

Each pipeline has their associated register file, namely main register file (MRF) and aux register file (ARF), and both register files contain 64 bytes each.

On-Tile Memory

Accompanied with the computing core in each tile is a 624kB In-Processor-Memory. The memory uses a contiguous unsigned-21 bit address space, and is divided into 2 regions. Figure 2.2 shows the two memory regions and their associated memory address.

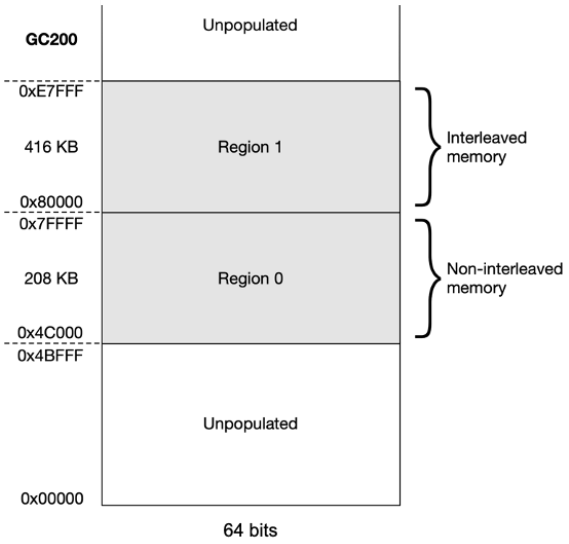


Figure 2.2: Tile Memory Regions. Figure from [1]

Each memory region is divided into 64-bit wide banks and concurrent accesses are allowed for different banks. Bit 3 of the address is used to allocate memory to alternating odd and even banks. As banks in region 1 are interleaved, this allows for concurrent access of two 64-bit float.

Host/Device communication

IPUs has the capability to send and receive data to and from the host computer. This capability is enabled via the use of Remote Direct Memory Access (RDMA) over Ethernet. Both the IPU-Machine and the host computer uses RDMA Network Interface Card (NIC) to transfer and write data. The NIC is interfaced to directly read and write to the DDR memory , while the IPU will read and write to the same DDR memory during its data exchange phase (see Section Bulk-Synchronous Parallel execution model)(Figure 2.5)

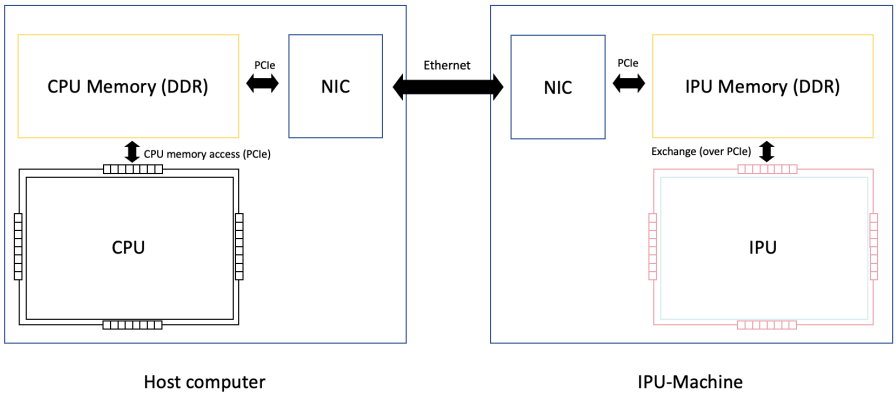


Figure 2.3: IPU/Host Communication. Figure from [1]

2.1.3 Programming Model

To fully utilise the parallel architecture of Graphcore IPUs, the Poplar SDK is developed in the C++ Language. This section references the work of Simon Håpnes to briefly explain the programming model and the process of creating a program to run on a Graphcore IPU Machine.

Bulk-Synchronous Parallel execution model

The IPU follows the Bulk-Synchronous Parallel (BSP) execution model, with which tasks are split tasks into three phases: [1]

1. Local Tile Compute
2. Global Cross-Tile Synchronisation
3. Data Exchange

The Local Tile Compute phase refers to the phase where all tiles are tasked to do work; The Global Cross-Tile Synchronisation refers to the phase where tiles wait for all processes to finish; The Data Exchange does what its name suggests, and exchanges data for the next compute phase.

As BSP separates the computing and communication phases, it means optimizations that involve overlapping communication and computation cannot be applied on the IPU.[5]

Program Flow

When programming an IPU devices, the structure of the program is as follows:

1. Define a *Target* (the *Target* could either be an IPU or an emulator)
2. Create a *Device* object from the *Target*
3. Create a *Graph* object for the *Device*
4. Build a *Program* using the *Graph* Object, which includes:
    - Mapping *Tensors* to the IPU tile
    - Creating *Vertices* that operates on the *Tensors*
    - Group Vertices that can be ran in parallel in to *Compute Sets*
    - Use the Compute Set to build the *Program*
5. Create a vector of *programs*, which includes streaming data from *host*
6. Create an *Engine* object
7. Compile the graphs and load the programs on to the *Engine* object
8. Use the Engine object to execute the programs sequentially

Host and Device The *Host* refers to the CPU machine which interfaces with the IPU. While the Device refers to the IPU machine itself.

Target and Device A *Target* points towards the hardware that executes the program. It could either be a physical Emulator, or a physical IPU. While the *Device* is a C++ object used to represent the *Target*. From the Device, a *Graph* object can then be initialised.

Tensor A *Tensor* is a representation of a vector of elements. it defines a variables used on the IPU and must be mapped on to one of the tiles on the IPU.

Vertices *Vertices Vertices* define the operations that are performed on the *Tensor*. Each Vertices corresponds to a set of instructions that will be ran by a single thread in the Computing phase. *Vertices* must be mapped the tiles of the IPU.

Computing Sets *Computing Sets* represents a collection of *Vertices* that runs concurrently in the computing phase. By adding *Vertices* together to form a computing set, it allows the compiler to understand these *Vertices* are meant to run in parallel to each other.

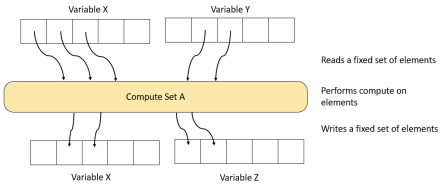


Figure 2.4: Graphical representation of a *Compute Set* . Figure from [1]

Graph With the *Computing Sets*, *Vertices*, and *Tensors* defined, the *Graph* can then be completed. A *Graph* essentially represents the logic behind the program to be executed.

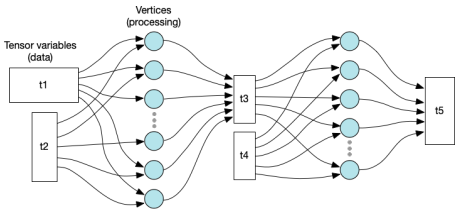


Figure 2.5: A Graph formed using *Compute Sets* and *Tensors*. Figure from [1]

Program The *Graph* can then be compiled into a *Program*. A vector of *Programs* is then ready to be sent to the IPU for execution. The other *Programs* are normally added to stream variables to the IPU prior execution and from the IPU after execution.

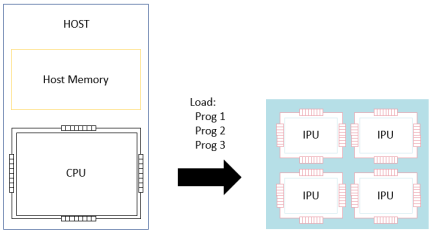


Figure 2.6: *Programs* loaded to the IPU. Figure from [1]

Engine The *Engine* object then controls the *device* to execute the vector of *Programs*

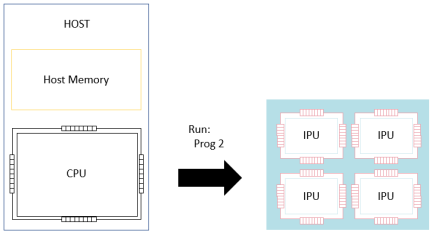


Figure 2.7: The Engine Object instructs the IPU to execute a *Program*. Figure from [1]

2.2 The Devito DSL and compiler framework

This section aims to provide an overview of the *Devito* Domain Specific Language (DSL) and compiler framework.

2.2.1 Overview

Devito is a tool for generating stencil codes from a high-level symbolic abstraction aimed at real-world applications, primarily seismic imaging. It allows users to express symbolic equations in *Python*, yet obtain a highly optimised *C* code comparable to performances of manual optimisation.

A typical programming process on *Devito* is demonstrated using a time-iterative Laplacian 3D stencil example obtained from [2].

2.2.2 Compiler Structure

The core structure of the *Devito* Compiler is illustrated in Figure 2.8.

The compiler first performs equations lowering from the DSL to obtain the actual expressions that is to be computed. Clustering then allows the compiler to identify groups of computation that has to be performed, which is used later to develop a Iteration/Expression Tree (IET). The IET can be thought of as an intermediate representation of a stencil operation. Symbolic optimisation is then used to perform higher level optimisations, such as factorisation, to reduce the amount of computations needed to evaluate an expression. The compiler then creates IET from the simplified expressions, which essentially form the loops (stencil). At last, the code is synthesised and sent the Just-In-Time (JIT) compiler, where the machine instructions are created.

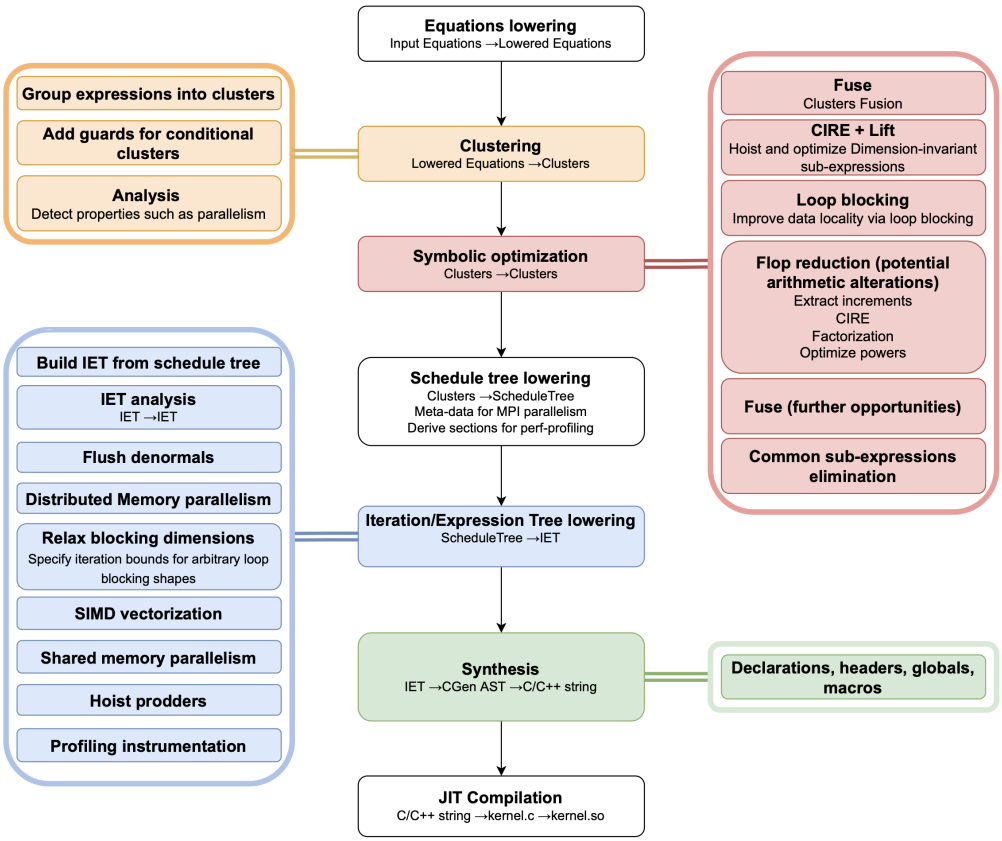


Figure 2.8: Compiler Structure. Figure from [2]

2.2.3 Example

Listing 2.1 shows the code to model a time-iterative Laplacian 3D stencil. The keyword *advanced* on line 25 instructs the compiler to apply optimisation when generating stencil code in c.

```
1  from devito import Grid, TimeFunction, Eq, solve, Operator, Constant
2
3  # Some variable declarations
4  nt, nx, ny, nz = 50, 10, 10, 10
5  dx = 2. / (nx - 1)
6  dy = 2. / (ny - 1)
7  dz = 2. / (nz - 1)
8  sigma = .25
9  nu = .5
10 dt = sigma * dx * dy * dz/ nu
11
12 # Grid initialization
13 grid = Grid(shape= (nx, ny, nz))
14 u = TimeFunction (name='u', grid=grid, space_order=2)
15
16 # Initialise u
17 init_value = 6.5
18 u.data[:, :, :] = init_value
19
20 a = Constant(name='a')
21 eq = Eq(u.dt, a*u.laplace + 0.1, subdomain=grid.interior)
22 stencil = solve (eq, u.forward)
23 eq0 = Eq(u.forward, stencil)
24
25 op1 = Operator (eq0, opt= ('advanced' ))
```

Listing 2.1: Generation of optimised kernel for a Laplacian time-iterative model

Listing 2.2 presents an optimised stencil code snippet generated by the *Devito* Compiler. The optimisation techniques applied to this stencil code encompass, cache blocking, loop invariant code motion, and common sub-expression elimination, among others.

```
1   float r0 = 1.0F/dt;
2   float r1 = 1.0F/(h_x*h_x);
3   float r2 = 1.0F/(h_y*h_y);
4   float r3 = 1.0F/(h_z*h_z);
5
6   for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M;
      ↪ time += 1, t0 = (time)%(2), t1 = (time + 1)%(2))
7   {
8     /* Begin section0 */
9     START_TIMER(section0)
10    for (int x0_blk0 = x_m; x0_blk0 <= x_M; x0_blk0 += x0_blk0_size)
11    {
12      for (int y0_blk0 = y_m; y0_blk0 <= y_M; y0_blk0 += y0_blk0_size)
13      {
14        for (int x = x0_blk0; x <= MIN(x0_blk0 + x0_blk0_size - 1, x_M); x += 1)
15        {
16          for (int y = y0_blk0; y <= MIN(y0_blk0 + y0_blk0_size - 1, y_M); y += 1)
17          {
18            #pragma omp simd aligned(u:16)
19            for (int z = z_m; z <= z_M; z += 1)
20            {
21              float r4 = -2.0F*u[t0][x + 2][y + 2][z + 2];
22              u[t1][x + 2][y + 2][z + 2] = dt*(a*(r1*r4 + r1*u[t0][x + 1][y + 2][z
    ↪ + 2] + r1*u[t0][x + 3][y + 2][z + 2] + r2*r4 + r2*u[t0][x + 2][y + 1][z + 2]
    ↪  + r2*u[t0][x + 2][y + 3][z + 2] + r3*r4 + r3*u[t0][x + 2][y + 2][z + 1] +
    ↪ r3*u[t0][x + 2][y + 2][z + 3]) + r0*u[t0][x + 2][y + 2][z + 2] + 1.0e-1F);
23            }
24          }
25        }
26      }
27    }
28    STOP_TIMER(section0,timers)
29    /* End section0 */
```

Listing 2.2: Optimised Jacobi-Like Laplacian stencil in C generated by the Devito Compiler

As shown in the example, the *Devito* DSL is tightly integrated with mathematical expressions for stencil computation. It provides flexibility in various aspects of stencil computation, including grid, stencil, space order, and time order. The definitions of these terms in the project context are presented in Section 2.3.

2.3 Related Terminologies

Grid

In this project, *Grid* is used to referred to a structured grid. It is one of the classes of problem mentioned by Asonoví et al in their classification of parallel computing problems[11]. It refers to a class of method, where the domain investigated is discretised at regular intervals to create a structured grid of cells. A feature of this class of problem is its regular access patterns and high spatial locality. Both of which has great potential to to be benefited by the IPU's tile architecture.

Stencil

In mathematics, a stencil refers to a pattern or template used to perform calculations or operations on a grid or discrete domain. It is commonly used in numerical analysis and computational mathematics, particularly in the context of solving partial differential equations (PDEs) and finite difference methods.

Space Order

Space order relates to the amount of grid points that a stencil uses to update the grid. A space order of 1 means that 1 layer of of the neighbouring points in each direction is used. In a 3D problem, that corresponds to 6 neighbouring points: top, bottom, left, right, front and back.

Time Order

Time order relates to the number of time instances one have to keep track of to update the grid. A time order 1 means that 1 time instance is needed to update a given point in the grid.

Chapter 3

Related Work

This chapter researches on studies related to the IPU and is divided into two sections.

The first section of the chapter 3.1 explores alternative hardware architectures suitable for similar workloads, specifically NVIDIA's Graphics Processing Unit (GPU) and Cerebra's Wafer Scale Engine (WSE). Its objective is to examine how Graphcore's IPU fits into the competitive landscape of High-Performance Computing. This is achieved by comparing the IPU to these hardware options across two key areas: Hardware Architecture and Programming Model.

The second section 3.2 aims to evaluate and examine relevant studies that have employed the IPU for stencil computations, namely the work of Simon Håpnes[9] and Louw and McIntosh-Smith[5]. The objective of the examination is to establish expectations of IPU performance in stencil computation, explore common IPU optimization techniques, and provide context on IPU performance in relation to other hardware, such as GPU.

Furthermore, to enhance the credibility of related papers, efforts are made to reproduce and replicate the results of the study whenever feasible. This verification and validation process aims to contribute to the reliability and robustness of the findings reported in these papers.

3.1 Related Hardware Comparison

This section compares the Colossus GC200 MK2 IPU with the Tesla S1070 GPU and Cerebras Wafer-Scale Engine 2. While achieving a direct hardware comparison among these architectures may present challenges, conducting a comparative analysis can still yield valuable insights into the role of the IPU within the realm of high-performance computing.

3.1.1 Hardware Architecture

GPU Hardware Architecture Comparison

A GPU is comprised of multiple multiprocessors, each equipped with its own set of stream processors and shared memory acting as a user-managed cache. The stream processors within the GPU possess the capability to perform integer and single-precision floating-point arithmetic, while additional cores are dedicated to double-precision computations. All multiprocessors within the GPU have direct access to global device memory, which is not subject to hardware caching. [3] An illustration of the GPU architecture is presented in figure 3.1.

The main difference between the GPU and IPU is its memory architecture. A GPU features a small shared memory utilised by multiple processors and mainly relies on fetching memory from a global pool of of memory. IPU, on the other hand, does not have a global pool of memory, but instead features a much larger in-processor-memory. This means memory access in the IPU is typically faster compared to that of the GPU. An IPU and GPU also differs in the precision it supports. While there exists GPU that support double precision arithmetic, the IPU has yet to provide support for the same precision.

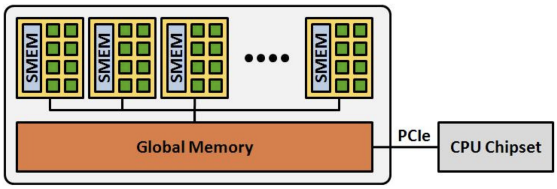


Figure 3.1: High-level view of GPU Architecture. Figure from [3]

WSE Hardware Architecture Comparison

A Wafer Scale Engine (WSE) has all its memory and compute resources embedded within one silicon wafer. An overview of the WSE is shown in figure 3.2. The second generation of the hardware features a total of 850,000 processing elements (PE), each containing 48KB of dedicated SRAM, a core capable of running 9 concurrent threads, and a router with 5 full-duplex channels. The router allows data to flow from a PE to four of its neighbors and to itself. Upon data arrival, a *task* can be programmed to react to data with a specific tag, which Cerebras refers to as the *color* of the data. This thread is asynchronous and does not affect the main operating thread, so no context switch is needed. In addition, the WSE is also designed to work with vectors or higher-dimensional objects with the help of a *data structure descriptor* (DSD). A DSD contains information regarding a particular object, such as its address, size, and stride, which assists with addressing problems in higher-dimensional tensors. The DSD is natively supported by the hardware, with each PE having a dedicated *Data Structure Register* (DSR) file specialized for storing DSD.

While both the IPU and the WSE follow a cache-less non-hierarchical memory architectural design, they differ in many regards. The WSE contains many more 'tiles' or PEs than the IPU. The WSE is designed to be a powerful hardware on its own, while the IPU is designed to work with other IPUs for scaling. While each PE on the WSE contains less memory than an IPU tile, it contains more hardware to assist with tensor operations. With the help of DSR, the WSE can address 4-dimensional tensors. This is not possible on the IPU, and indexing functions are often needed to address higher-order objects. Furthermore, the WSE instruction supports AXPY operations $\mathbf{y} = \mathbf{y} + a \times \mathbf{x}$, where $\mathbf{y}$ and $\mathbf{x}$ are vectors while a is a scalar. The IPU does not support such an instruction and cannot perform this operation atomically. The WSE is therefore more capable of handling higher-dimensional stencils than an IPU. In terms of concurrency within each tile/PE, the WSE provides support for a background asynchronous thread which can operate on data as soon as it arrives on the tile, whereas the IPU features 2 operation pipelines, both of which are synchronous.

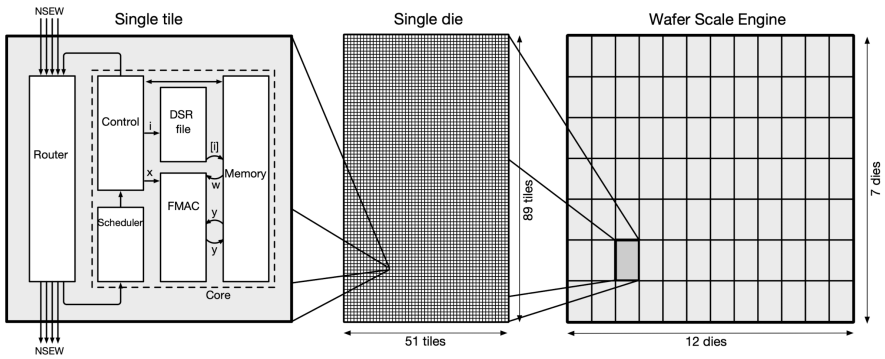


Figure 3.2: An overview of the Wafer Scale Engine (WSE) Hardware Architecture. Figure from [4]

3.1.2 Programming Model

In all three programming models, *host* is used to refer to the main CPU process, whereas *device* the specialised hardware.

GPU Programming Model Comparison

The GPU provides a heterogeneous programming model with the CUDA library implemented in the C language. Each processor can be assigned a thread, and a set of threads forms a block, while a set of blocks forms a grid. A *Kernel* represents a grid that performs a specific function. Programmers have the freedom to assign a set of blocks and threads to a given grid implemented in a *Kernel*. A *Kernel* can be initiated by the host, and the program immediately returns to the host. Blocking functions can be used to wait for the completion of GPU execution. Optimization techniques in GPU programming often follow those of multi-core CPU programming, with a focus on more efficient caching.

While CUDA uses the C language, Poplar utilizes the C++ language. The concepts of *thread*, *block*, *grid*, and *Kernel* in CUDA can be compared to the *Vertex*, *Compute Set*, *Graph*, and *Program* in Poplar. However, Poplar introduces an additional notion of a *Tensor*, which tracks data dependencies when compiling a graph. This dependency check is not built into the programming model in CUDA. Moreover, Poplar differs from CUDA in that it uses the Bulk Synchronous Parallel (BSP) model, where Vertices in Compute Sets are not allowed to operate on the same data. This difference implies that concurrency issues such as race conditions seldom occur in Poplar but can be a significant problem in CUDA. Since BSP separates data exchange and computation, many existing optimization techniques that take advantage of overlapping communication and computation do not apply to the IPU. While GPU optimization focuses on optimizing communication, specifically hiding memory bandwidth associated with data fetching, the IPU emphasizes compute-side optimization through the utilization of vectorized instructions. Lastly, while CUDA allows a single CPU to spawn multiple Kernels to run concurrently on the GPU, the BSP model restricts *Programs* to run sequentially on the IPU.

WSE Programming Model Comparison

The WSE uses the Cerebras Software Language (CSL) to program the device, and Python to program the *host* CSL employs static typing and shares similarities with C programming. It was referred to as the medium level language in the The Cerebras Software Development Kit [12], as it contains features from both high and low level programming languages. It exposes the capabilities of the instruction set and hardware structures it controls, yet retain high level programming features such as structured control flow, hides processor registers and performs automatic register allocation. The CSL introduces the concept of a *task*, which is akin to a function performing tensor operations asynchronously and reacting to incoming data of a specified *color*. Data movement is statically compiled by assigning a *color*, a 5-bit tag that determines the transmission channel among the 5 duplex channels in a given PE.

In contrast, the IPU employs graphs to compile programs, while CSL utilizes data-flow. The Poplar library allows a process to access data beyond its tile memory by specifying data requirements beyond in-processor memory, and the compiler automatically generates exchange logic to facilitate the necessary data movement. In CSL, data movement is controlled through the concept of *color*. Programmers configure routing by assigning a specific *color* to data during compilation, enabling dynamic reconfiguration of virtual channels during runtime. Additionally, the notion of a *task*, performing asynchronous operations, is exclusive to the CSL language. The IPU offers a similar, albeit different, functionality through the utilization of the *aux* pipeline, which operates independently from the *main* pipeline. However, asynchronous instructions are not available in the IPU.

3.2 IPU on HPC

3.2.1 IPU in 3D Heat Equation

Problem - 3D Heat Equation

In the work of Simon Håpnes[9], the 3D Heat equation had been implemented on the IPU using the finite difference method. Equation 3.1 shows the discretisation of the heat equation using *forward difference in time* and *central difference in space*:

$$
\begin{aligned}
u_{ijk}^{t+\delta} =& (1-6\alpha)u_i^t jk + \\
& \alpha(u_{i+1,j,k}^t + u_{i-1,j,k}^t + u_{i,j+1,k}^t + u_{i,j-1,k}^t + u_{i,j,k+1}^t + u_{i,j,k-1}^t)
\end{aligned} \tag{3.1}
$$

Result

Table 3.1 presents the run-time benchmark from computing solution for the discretised heat equation 3.1 on a 3D mesh comprising approximately 46.6 million elements (360 × 360 × 360), over a duration of 1000 time steps. Where *HeatEquationSimple* and *HeatEquationOptimized* refers to a simple and optimised version of the problem respectively. *HeatEquationOptimized* differs to *HeatEquationSimple* in that it uses vectorised instructions to reduce the number of floating point operations needed to compute the stencil

Result	**Processor**	**Vertex**	**Time**	**Throughput**	**Minimal Bandwidth**
Reproduced	IPU	*HeatEquationSimple*	0.36 s	1.03 TFLOPS	3.81 TB/s
	IPU	*HeatEquationOptimized*	0.26 s	1.45 TFLOPS	5.36 TB/s
Original	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 3.1: HeatEquation 3D runtime stats

Discussion

Table 3.1 shows that the reproduced throughput and minimal bandwidth had an 18% increase for the *HeatEquationSimple* implementation, whilst stayed approximately the same for the *HeatEquationOptimized* implementation.

The increase performance for the *HeatEquationSimple* implementation is presumably due to improvements in the updated poplar library. As it was not recorded which version of the poplar library is used, it is difficult to pinpoint the exact improvements that causes the improvement. Nonetheless, it could be established that improvements had been made to the IPU compiler to enhance of performance of IPU even prior deliberate efforts of optimisation.

More importantly, the reproduced results indicate that the performance of the IPU in 3D stencil computation is estimated to be approximately 1.0 TFLOPS in throughput and 3.11TB/s in minimal bandwidth.

The study also noted the co-issue of instructions to the *main* and *aux* pipeline allowed clock cycle per iteration required by the 3D heat equation be reduced from 27 to 16, which points presents a direction for optimisation in the following chapters.

3.2.2 IPU in Gaussian blur image filter / 2D Lattice Boltzmann fluid simulation

Problem - Gaussian blur image filter / 2D Lattice Boltzmann fluid simulation

In the works of Louw and McIntosh-Smith, two stencil-based computations has been employed: a Gaussian blur image filter and a 2D Lattice Boltzmann fluid simulation. While the Gaussian blur image filter experiment demonstrated IPU's capability in smaller sized problem, the 2D Lattice Boltzmann fluid simulation unveiled some of IPU's limitation in larger sized problem.

Result

Figure 3.3 showed a roofline model for the IPU, where the performance of a non-vectorised Gaussian blur image filter, vectorised Gaussian blue image filter and optimised 2D Lattice Boltzmann fluid

simulation were represented by a blue star, red diamond and black dot respectively. The roofline was plotted based on empirical results from the study.

Figure 3.4 showed the relative performance of 1 IPU, 48 Skylake CPU CPU and 1 NVIDIA V100 GPU for the Gaussian blur image filter. While figure 3.5 compared the same set of hardware but for the 2D Lattice Boltzmann fluid simulation.

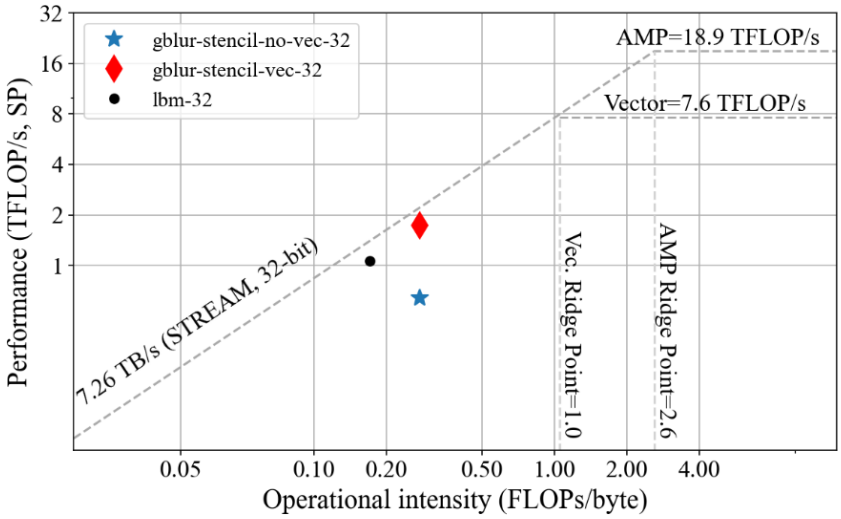


Figure 3.3: Roofline models for the IPU. Figure from [5]

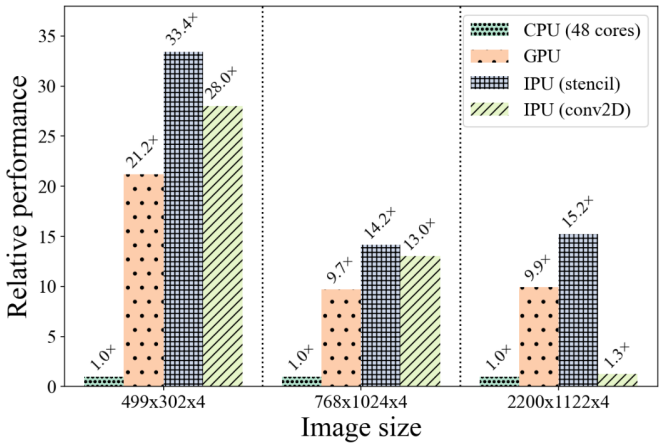


Figure 3.4: Relative performance of Gaussian Blur implementations on 1 IPU (stencil and convolution) vs 48 Skylake CPU cores and NVIDIA V100 GPU (32-bit, vectorised). Figure from [5]

Discussion

Figure 3.3 showed that both vectorised implementation for the Gaussian blur image filter and 2D Lattice Boltzmann fluid simulation was able to achieve performance close to the theoretical limit. The result suggest that vectorisation is a very key optimisation in the context of IPU programming.

Figure 3.4 and 3.5 showed that the IPU outperforms GPU and CPU in Gaussian Blur application, as well as smaller problem sizes of Lattice-Boltzmann Fluid Simulation. However, GPU outperforms the IPU for as the problem size grew larger for the Lattice-Boltzmann Fluid Simulation.

The literature points out that the performance gap between the IPU and GPU arises due to data rearrangements induced by the IPU's halo exchange implementation, outweighing the benefits of the fast IPU exchange. This results in the GPU outperforming the IPU for larger problem

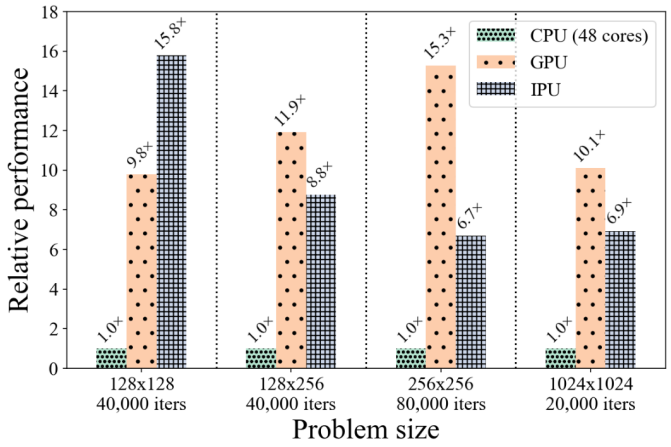


Figure 3.5: Relative performance of LBM D2Q9 implementations on 1 IPU vs 48 CPU cores and NVIDIA V100 GPU. Figure from [5]

sizes as the cost of data rearrangements grew larger[5]. The result suggested that the IPU's high-bandwidth, low-latency on-chip memory can potentially benefit HPC computations, but this advantage is limited to specific problem sizes.

On the other hand, the study also demonstrated the importance of reducing the volume of halo exchange, as it was shown that for the 2D Lattice Boltzmann fluid simulation 27% of execution time was spent in exchange and data rearrangement activities for 1 IPU. This number rises to more than 50% of execution time on 16 IPUs. Henceforth in the following chapters, special attentions should be paid to minimise the amount of halo exchange.

3.3 Conclusion

From the examination of relevant studies, the following can be established:

Expectation of IPU performance in stencil computation The performance of the IPU in 3D stencil computation is estimated to be approximately 1.0 TFLOPS in throughput and 3.11TB/s in minimal bandwidth.

Common IPU optimization techniques Vectorisation a common optimisation direction in IPU programming. If possible, the co-issue of instructions to the *main* and *aux* pipeline also presents a major optimisation opportunity.

IPU performance in relation to other hardware Although IPU outperforms GPU on problem of smaller size, the result may not generalise to larger problems due to extra cost in data arrangement. It showed that the relative performance of IPU may not apply uniformly across all problem sizes.

Chapter 4

The 3D Diffusion Equation

In the works of Håpnes in 2021 [9], the performance of the IPU (Intelligent Processing Unit) was evaluated by benchmarking its capabilities 3D Diffusion equations. This section investigates the same Diffusion Equation, but space order is also varied to investigate the effect of increased bandwidth on the IPU performance. In addition, this section also walks through the modifications needed to increase the space order of a problem.

The primary objective of this chapter is to answer the following questions:

1. What are the modifications needed to implement problems of different space order?
2. What are the programming challenges associated with IPU machines?

In the following sections, 4.1 defines the diffusion problem mathematically. 4.2 describes the coding process using the *Devito* compiler framework as well as the applied optimisations. 4.3 describes the process of porting the stencil equation onto the IPU.

4.1 The Diffusion Equation

The diffusion equation investigated in this chapter is presented below:

$$\frac{\partial u}{\partial t} = \alpha(\frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y} + \frac{\partial^2 u}{\partial^2 z}) \quad , \quad 0 < x, y, z < 1, t > 0 \tag{4.1}$$

Boundary Conditions: $u(0,0,0,t) = 0 \;\; u(1,1,1,t) = 0$ (4.2)

Initial conditions: $u(x,y,z,t) = f(x,y,z)$ (4.3)

(4.4)

The *Forward Difference in Time* and *Central Difference in Space* is applied to create a stencil equation that can be used to update the heat mesh at every time iteration. Fortunately, the *Devito* compiler framework performs this operation automatically, so this step, traditionally done by hand, could be performed automatically by the *Devito* compiler.

4.2 Implementation - Devito

The problem then transforms into a much simpler problem: how to express the equation using the *Devito* DSL. The representation of the diffusion equation 4.2 is illustrated in Listing 4.1.

In Line 1 to 10, the variables are initialised from the command line arguments. A `grid` describing the domain of the diffusion is then created on line 12 based on the variables initialised. This `grid` is then mapped into a `TimeFunction` on line 13. Line 15-16 initialises the data. After the problem space has been set up, the `Eq` object is used to express equation 4.2 by equating the partial derivative of u against the laplacian matrix of u multiplied by a constant α. At last, the stencil equation can

be obtained by solving the equation using the `solve` function. The stencil code for the equation is automatically generated in Line 26-27 by passing the `stencil` to the `Operator`. The resulting stencil equation is illustrated on equation 4.5, while the control flow in listing 4.5.

```
1  nx, ny, nz = args.shape
2  nt = args.nt
3  nu = .5
4  dx = 2. / (nx - 1)
5  dy = 2. / (ny - 1)
6  dz = 2. / (nz - 1)
7  sigma = .25
8  dt = sigma * dx * dz * dy / nu
9  so = args.space_order
10 to = 1
11
12 grid = Grid(shape=(nx, ny, nz), extent=(2., 2., 2.))
13 u = TimeFunction(name='u', grid=grid, space_order=so)
14
15 u.data[:, :, :, :] = 0
16 u.data[:, int(3*nx/4), int(ny/2), int(nz/2)] = 1
17
18 a = Constant(name='a')
19 # Create an equation with second-order derivatives
20 eq = Eq(u.dt, a * u.laplace, subdomain=grid.interior)
21 stencil = solve(eq, u.forward)
22 eq_stencil = Eq(u.forward, stencil)
23
24 op = Operator([eq_stencil], subdomain=grid.interior)
25 op(time=nt, dt=dt, a=nu)
```

Listing 4.1: Expressing the Diffusion Equation on Devito

Equation 4.5 shows the equation produced by the *Devito* compiler for a Space Order 4 problem. The 13-point stencil displayed on the right-hand side of Equation 4.5 represents the operation that will be sent to the IPU for computation. It is important to note that this stencil is solely defined by the inner nodes of the three-dimensional mesh, while the Dirichlet boundary condition has been applied along the edges. This means that only the inner part of the 3D mesh is updated, whereas the outermost 2 layers are set to 0 and kept constant for a space order 4 diffusion.

$$
\begin{aligned}
u_{ijk}^{t+\delta} = dt \cdot (\alpha \cdot (& \\
& r_1 \cdot \left(r_4 - 8.\dot{3} \times 10^{-2} \cdot (u_{i-2,j,k}^t + u_{i+2,j,k}^t) + 1.\dot{3} \cdot (u_{i-1,j,k}^t + u_{i+1,j,k}^t)\right) \\
& + r_2 \cdot \left(r_4 - 8.\dot{3} \times 10^{-2} \cdot (u_{i,j-2,k}^t + u_{i,j+2,k}^t) + 1.\dot{3} \cdot (u_{i,j-1,k}^t + u_{i,j+1,k}^t)\right) \\
& + r_3 \cdot \left(r_4 - 8.\dot{3} \times 10^{-2} \cdot (u_{i,j,k-2}^t + u_{i,j,k+2}^t) + 1.\dot{3} \cdot (u_{i,j,k-1}^t + u_{i,j,k+1}^t)\right) \\
) + r_0 \cdot u_{ijk}^t & \big) \\
r_0 = 1.0/dt, r_1 = 1.0/h_x^2, r_2 = 1.0/h_y^2, r_3 = 1.0/h_z^2, & r_4 = -2.5 \cdot u_{ijk}^t \\
\text{where} \quad u \text{ is a regular 3D mesh,} \ \alpha \text{ is a constant property} &
\end{aligned}
\tag{4.5}
$$

Listing 4.2 shows a snippet of the c code generated by the *Devito* compiler to run the stencil equation 4.5. An important observation of this snippet of code is the application of optimization techniques such as time buffering, blocking, multi-threading, and vectorisation. On line 4, one may observe that time buffering is applied successively to interchange between two-time values: *t0* and *t1*. So instead of saving the whole grid of the mesh, only two copies of the mesh are present at each iteration. The time values *t0* and *t1* represent the grid of the present time step and the next time step, respectively. Then from lines 7-10, blocking is applied to divide the work into blocks, which defines the workload assigned to each thread. By assigning blocks of data to individual threads, the code takes advantage of the spatial locality of data. So when data is fetched, a cache hit is more likely as the cache still retains data fetched from the previous iteration. Line 11 then instructs the `openmp` library to spawn multiple threads that perform SIMD instructions. The concurrency

of the problem is leveraged by utilising both the threading and vectorisation provided by a SIMD multi-core CPU.

```
float r0 = 1.0F/dt, r1 = 1.0F/(h_x*h_x);
float r2 = 1.0F/(h_y*h_y), r3 = 1.0F/(h_z*h_z);

for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M; time += 1, t0 = (time)%(2), t1 = (time + 1)%(2)){
/* Begin section0 */
START_TIMER(section0)
for (int x0_blk0 = x_m; x0_blk0 <= x_M; x0_blk0 += x0_blk0_size){
  for (int y0_blk0 = y_m; y0_blk0 <= y_M; y0_blk0 += y0_blk0_size){
    for (int x = x0_blk0; x <= MIN(x_M, x0_blk0 + x0_blk0_size - 1); x += 1){
      for (int y = y0_blk0; y <= MIN(y_M, y0_blk0 + y0_blk0_size - 1); y += 1){
        #pragma omp simd aligned(u:32)
        ... (the stencil equation)
}}}}}
STOP_TIMER(section0,timers)
/* End section0 */
}
```

Listing 4.2: The c code stencil produced by *devito*

4.3 Implementation - IPU

After discretised 3D diffusion equation was implemented on the CPU and GPU using the *Devito* compiler, the problem was then ported to IPU using the Poplar framework and implemented in C++. The process of development could be explained in three main parts, Memory Allocation, Work Division, and Compute.

4.3.1 Memory Allocation

As the space order of 4 is used, extra ghost cells [13] is added around the edges following the *Dirichlet condition*, which initialises the boundary to 0. This means more memory than the problem size is needed to compute the diffusion equation. These cells are padded for ease of execution and are considered when evaluating the performance of the IPU. The actual result obtained, however, discards these ghost cells and returns only the problem space specified. figure 4.1 illustrates padded ghost cells in a 2D problem with space order of 4. The added ghost cells are represented in the colour grey.

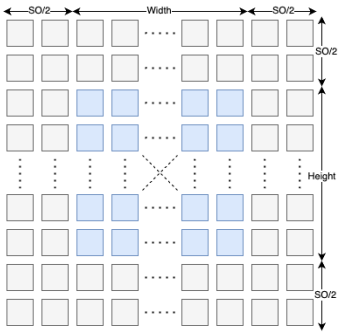


Figure 4.1: Added ghost cells for a space order of 4 (grey cells)

4.3.2 Work Division

Similar to the blocking optimisation applied in the CPU implementation in listing 4.2, the work can also be divided efficiently in the IPU using the blocking method. The work division solution devised by Håpnes in 2021 [9] follows the blocking principle and generalises to the diffusion problem.

This section documents the changes needed to adapt the solution to the context of the isotropic diffusion problem of variable space order.

The mesh, represented by a 3D tensor within the IPU, is divided and allocated among the available tiles. As the BSP model dictates that the execution phase is limited by the longest working thread, the goal of the work division is to balance work among the tiles while maintaining minimum communication volume. This is achieved by using a program to iteratively search for the suitable number of partitions along each of the dimensions on the grid.

Figure 4.2 shows the process of finding the optimal work division visually. By iterating through configurations a to e, the surface area of each configuration is calculated. The configuration with the least surface area is chosen to minimise communication volume. From empirical evidence, the optimal configuration tends to take an intermediate form between configurations b to d.

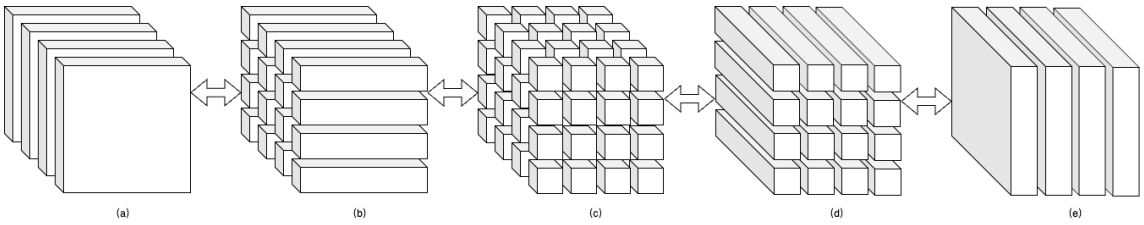


Figure 4.2: Visualisation of searching in the Work Division Space

The body of the program is shown in Listing 4.3. As work is only performed on the inner grid while the outer grid stays constant under the *Dirichlet conditions*, the work division only needed to be performed in the inner grid. A variable, `options.padding`, is used to reduce the dimensions to reflect the inner grid, where update is being performed. A *space order* of 4, for example, would give a *padding* of 2.

The program then follows these steps to determine the optimal partitions.

1. factorize the tile count into three integers that, when multiplied together, yield the given tile count.
2. iterate through combinations of the factors to divide the mesh
3. calculate the surface area from the given partition.
4. choose the partition with the smallest surface area

Note that the surface area of a given partition does not represent the total communication volume, but it provides a good estimation. Using this method, the work partition varies in each dimension at most by one unit. The largest block for a 342^3 problem in space order of 4 is $15 \times 43 \times 43$, whiles the smallest is $14 \times 42 \times 42$. The maximum work imbalance is 12% in a 342^3 grid.

```
float smallest_surface_area = std::numeric_limits<float>::max();
std::size_t height = (options.height - 2*options.padding);
std::size_t width = (options.width - 2*options.padding);
std::size_t depth = (options.depth - 2*options.padding) / options.
  ↪ num_ipus;
std::size_t tile_count = options.num_tiles_available / options.num_ipus;
for (std::size_t i = 1; 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};
        if (i*j*k != tile_count) {
          throw std::runtime_error("workDivision(), factorization does
  ↪ not work.");
        }
        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;
                  options.splits = splits;
    }}}}}}}}}}
```

Listing 4.3: Work Division. Body from [9]

4.3.3 Compute

After distributing the workload among the tiles, the vertices are assigned to perform the computations. The communication between tiles is automatically handled by the Poplar framework. To assign the input to require data beyond the tile's memory, the necessary code for fetching the data is generated. This is achieved by connecting a larger slice of data to the vertex's input, as depicted in Figure 4.3, where different colours represent data allocated to different tiles. The input and output data required by the vertex are bounded by the black boxes.

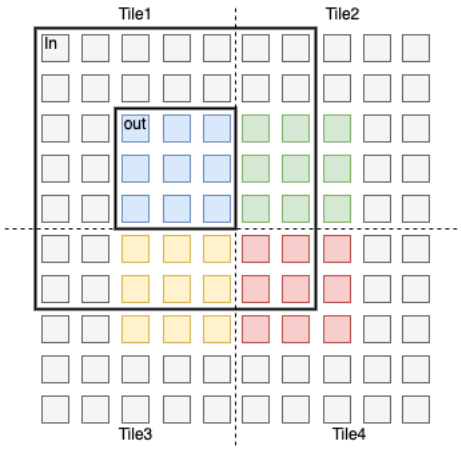


Figure 4.3: Input and output assignment illustration for Space order 4

Listing 4.4 shows the overall *codelet* structure used to compute the diffusion equation. Where line 5-6 shows the *input* and *output* of the vertex. `in` and `out` refers to the previous and current time-step of the mesh, respectively. The `Input` and `Output` type indicate to the *popc* compiler that these vectors form the input and output of the vertices. `Vector` is a *poplar* representation of vectors, while the four parameters `<float , VectorLayout :: SPAN , 8 , true>` determines how the Vectors are located in the IPU. `float` defines the type of the elements; `VectorLayout::SPAN` specifies that a pointer to the first element and size of the `Vector` is stored and used to access the `Vector`; `8` refers to the memory alignment, which allows for concurrent access when variables are stored in separate banks; `true` specifies that the elements are stored in the interleaved region of the IPU memory. Refer to section 2.1.2 for more details of interleave memory and concurrent access.

Line 7 shows the necessary parameters needed by the vertex to navigate the assigned workload. Line 8 contains the required parameters to complete the computation. These values are pre-computed and saved on the vertex prior to runtime; line 12-14 shows the indexing function used to address a 3D mesh in a flattened 2D array. The indexing function is used so that the *grid* can still be accessed using 3D array coordinates but on a flattened 2D array. The reason why the array was flattened to 2D was due to the poplar framework constraints, where `Input` and `Output` could, at most, be a 2D array. Line 15 shows the `compute()` function, which specifies the operations to be performed on the vertex. This is where the code for equation 4.5 is placed.

```
1  class DiffusionEquationSimple : public Vertex {
2      public:
3        DiffusionEquationSimple();
4
5        Vector<Input<Vector<float, VectorLayout::SPAN, 8, false>>> in;
6        Vector<Output<Vector<float, VectorLayout::SPAN, 4, false>>> out;
7        const unsigned worker_height,worker_width,worker_depth,padding;
8        const float alpha,hx,hy,hz,dt,r0,r1,r2,r3;
9
10     /* The index corresponding to [x,y] in for a row-wise flattened
11     2D variable*/
12     unsigned idx(unsigned x, unsigned y, unsigned w) {
13         return y + x*w;
14       }
15     bool compute () {
16         ...
17     }
18 }
```

Listing 4.4: codelete structure

4.3.4 IPU performance estimate

The duration of the execution was measured using the `chrono` tools in C++. The wall time is then used to calculate the computational throughput. The computational throughput was measured using the $Gpts/s$. This was calculated by multiplying the inner volume of elements in the mesh and the number of iterations, then dividing by the wall time measured.

$$throughput = \frac{(inner\ volume)(no.\ time\ steps)}{measured\ wall\ time}(Gpts/s) \tag{4.6}$$

4.4 Programming Challenges

When programming for IPU, a few hurdles have been observed. First, the compilation time could take very long. This time is observed to be positively correlated with the space order and problem size. Space order has a greater effect on compile time than problem size. This is likely to be because the number of exchange codes needed to be generated increases with space order.

It is also observed that a large amount of memory is needed to compile the IPU program within a reasonable amount of time. Around 100GB of memory and several hours are needed to compile a program that utilises the memory of four IPUs and employs a stencil of space order 8.

Chapter 5

The 3D Wave Equation

5.1 Motivation

In this chapter, the wave equation will be examined. This investigation serves to extend the complexity of the exploration in two significant ways.

First, it explores the performance of the IPU under increased time order. It is worth noting that the previously examined equations only involved a time order of 1, where solely the magnitudes of the previous time step were considered. Conversely, the wave equation incorporates a time order of 2, which requires the retention of an additional layer of data throughout the computation process.

Furthermore, in addition to the increased time order, the wave equation introduces additional layers of constant data, such as velocity profiles and damping factors. Each of these layers possesses unique values that correspond to specific positions. Consequently, the wave equation places more stress on the computation bandwidth compared to the diffusion equation. More importantly, It also increases the memory density of the problem, meaning more memory is needed to store data for a given grid space compared to that of the diffusion equation.

5.2 The Wave Equation

The wave equation investigated in this chapter is presented below:

$$\frac{\partial^2 u}{\partial t^2} = c^2\left(\frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y} + \frac{\partial^2 u}{\partial^2 z}\right) \quad , \quad 0 < x, y, z < 1, t > 0 \tag{5.1}$$

Boundary Conditions: $u(0,0,0,t) = 0 \;\; u(L,L,L,t) = 0$ (5.2)

Initial conditions: $u(x,y,z,t) = f(x,y,z)$ (5.3)

$$\frac{\partial u}{\partial t}(x,y,z,0) = g(x) \tag{5.4}$$

Similar to Chapter 4, the *Forward Difference in Time* and *Central Difference in Space* are applied to create a stencil equation that updates the wave mesh at every time iteration. Only this time, it requires mash data from 2-time steps due to the increased time order.

The wave propagation is often accompanied by a source which produces the wave. This will be injected using the `RickerSource` provided by the *Devito* DSL.

5.3 Implementation - Devito

The problem, when expressed in the *Devito* DSL, is shown in listing 5.1.

Prior to line 2, the `spacing`, `origin`, and `shape` of the problem have been defined. `spacing` defines the extent of space the grid occupies, `origin` defines the origin of the coordinate system, and `shape`

defines the number of grid points used to represent the space. In line 2, these variables are used to create a seismic model. `vp` defines the velocity profile of the spacing, and `nbl` defines the thickness of the layer applied at the boundaries for damping the waves in this model.

The `TimeAxis` is then constructed from `t0`, `nt`, and `dt` in line 5-9. `nt` is user-defined and controls how long in milliseconds the model will be run. `dt` is obtained from `model.critical_dt`, which provides the maximum `dt` and the minimum time steps required for the model to be stable.

From lines 11-13, the stencil used to update the wave mesh is constructed in a similar fashion to the diffusion problem in the previous chapter. By constructing `u` as a `Time Function`, the memory needed for constructing `grid` with a space order `so` and a time order `to` is reserved. The Partial Differential Equation (PDE) representing equation 5.1 is constructed in line 12, and the stencil is obtained in line 13 by solving the PDE.

Lines 15-19 define the source of the wave using the `devito RickerSource` object. The source is propagated alongside the wave stencil for 50 ms in lines 21-22 to produce a significant enough wave. This wave forms the initial condition on which experiments will be run. This mesh is then exported on lines 24-34 alongside other essential parameters for the IPU. Once the source is generated, it is removed from the stencil for benchmarking. This process is shown in lines 36-37.

```
1  # shape, spacing and origin defined from command line arguements
2  model = Model(vp=v, origin=origin, shape=shape, spacing=spacing,
3                space_order=so, nbl=10, bcs="damp")
4
5  t0 = 0.  # Simulation starts a t=0
6  tn = nt  # Simulation last 1 second (1000 ms)
7  dt = model.critical_dt  # Time step from model grid spacing
8
9  time_range = TimeAxis(start=t0, stop=tn, step=dt)
10
11 u = TimeFunction(name="u", grid=model.grid, time_order=to, space_order=so)
12 pde = model.m * u.dt2 - u.laplace + model.damp * u.dt
13 stencil = Eq(u.forward, solve(pde, u.forward))
14
15 f0 = 0.010  # Source peak frequency is 10Hz (0.010 kHz)
16 src = RickerSource(name='src', grid=model.grid, f0=f0,
17                    npoint=1, time_range=time_range)
18 src.coordinates.data[0, :] = np.array(model.domain_size) * .5
19 src_term = src.inject(field=u.forward, expr=src * dt**2 / model.m)
20
21 op = Operator([stencil] + src_term, subs=model.spacing_map)
22 op.apply(time=50, dt=model.critical_dt)
23
24 parameters = {
25   "dt": float(dt),
26   "nt": int(nt),
27   "steps": time_range.num ,
28   "shape": [float(model.damp.shape[0]+so),float(model.damp.shape[1]+so),
       ↪ float(model.damp.shape[2]+so)],
29   "damp": model.damp.data.tolist(),
30   "vp": model.vp.data.tolist(),
31   "u0": u.data[0].tolist(),
32   "u1": u.data[1].tolist(),
33   "u2": u.data[2].tolist()
34 }
35 # code for saving parameters in json
36 op = Operator([stencil], subs=model.spacing_map)
37 op.apply(time=time_range.num, dt=model.critical_dt)
```

Listing 5.1: Wave Equation expressed on Devito

5.4 Implementation - IPU

After implementing the 3D wave equation on the `devito`, This section documents the process of porting the solution to IPU using the Poplar framework in C++. The development process is divided into 4 parts: Memory allocation, Work Division, Control Flow, and Compute.

5.4.1 Memory allocation

Similar to the diffusion problem discussed in the previous chapter, the wave equation demands more memory than the actual problem size. Aside from the ghost cells used for completing the inner grid in various space order stencil computations, an additional damping layer is incorporated to handle boundary conditions in wave problems. The number of damping layers added is controlled by the `nbl` parameter in the `model` described in Section 5.3. The boundary condition is visualised in Figure 5.1, where the dark greyed cells represent this padded damping layer, and light grey represents the added ghost cells.

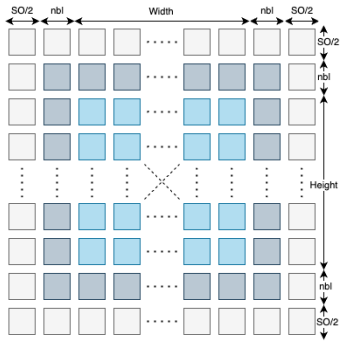


Figure 5.1: Padded Damp layers (dark grey) and ghost cells for space order of 2 (light grey) in a 2D stencil problem

Moreover, the wave equation necessitates additional memory beyond the defined problem size. Increased time order requires extra memory buffering, leading to the need for additional copies of the mesh grid. For instance, a time order of 2 would require three copies of the mesh grid for time buffering. Additionally, more memory is necessary to store the coefficients for dampening and velocity profiles associated with each cell on the grid. While they have the same size as the wave mesh, these coefficients are read-only.

5.4.2 Work Division

In a BSP programming model, throughput is limited by the longest-running worker. As the exchange phase in the BSP is dependent on the amount of data transfer, the communication is also desirable to be minimised. In the diffusion problem, work division is performed by a program to search for the work partition for each side of the grid. Although the wave equation requires additional mesh, the surface area of the mesh scales linearly with the number of mesh, given their size is the same. As all the mesh required in calculating the wave equation are the same in size, the same algorithm could be used to divide the work among the tiles. Figure 4.2 shows visually how the work division is performed, while listing 4.3 shows the corresponding code. For more details on how work division is performed, refer to section 4.3.2.

5.4.3 Control Flow

There are three main steps involved when programming an IPU: streaming data to the IPU, execution, and streaming data back to the host. While the control flow is straightforward with a time order of 2, it becomes more complex as the time order increases. Therefore, this chapter includes an additional section to document workflow changes. Since the execution determines where data should be streamed to and from, it will be explained before the other steps.

Execute

To iterate the stencil equation over multiple cycles within the IPU, the `poplar` object `poplar::program ↪ ::Repeat` is used. As mentioned in Section 5.4.1, three memory buffers are needed to compute a stencil with a time order of 2. These three memory locations are referred to as `a`, `b`, and `c` for ease of discussion. In `devito`, `t0`, `t1`, and `t2` are designated as the meshes for the previous, next, and current time steps, respectively. Depending on the iteration, `a`, `b`, and `c` could represent any of `t0`, `t1`, and `t2`. This is illustrated in Figure 5.2. For example, in one iteration, `b` and `a` could be used to compute `c`, and then in the next iteration, `c` and `b` would be used to compute `a`, and so on. Due to this memory rotation, iterations are repeated in sets of three to complete the memory rotation. Therefore additional iterations are prepended to make up for the remaining iterations not divisible by 3. Extra logic is therefore needed to stream data `t0`, `t1`, and `t2` to the correct memory location, and the final iteration will always be available at memory allocation `c` after execution.

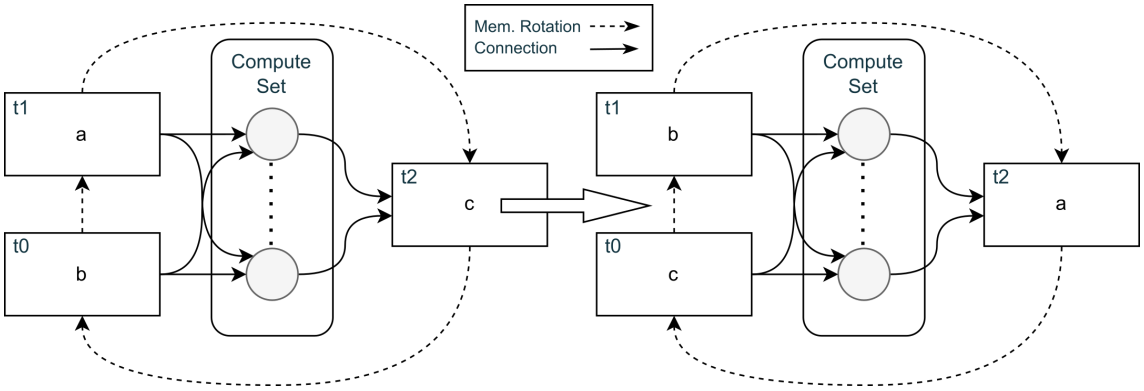


Figure 5.2: Memory `a`,`b` and `c` rotates to represent time `t0`, `t1` and `t2`

Streaming to device

In IPU, constants and variables are typically transferred in a different manner. While constants such as the `damp` and `velocity profile` mesh are saved in the program in compile time, variables are streamed at runtime. Depending on the total number of iterations, the corresponding data of `t0`, `t1`, and `t2` are streamed to different memory locations. The stream logic is illustrated in Table 5.1.

Iterations	t0	t1	t2
iteration\%3==0	c	b	a
iteration\%3==1	b	a	c
iteration\%3==2	a	c	b

Table 5.1: Memory allocation Logic

Streaming to host

As the final iteration is always stored in memory location `c`, the program always streams the data from memory location `c` to obtain the latest iteration of the mesh.

5.4.4 Compute

Listing 5.2 demonstrates the general *codelet* structure used for programming the *vertex* responsible for computing the wave equation. The structure closely follows that of the diffusion equation but with more data connected to the *vertex*.

In lines 4-8, the inputs to the *vertex* are specified. The wave equation requires two-time steps of the mesh, represented by `in1` and `in2`, as well as the damping coefficients (`damp`) and the velocity profile (`vp`). All the inputs are configured in the same way: they are set to be a 2D array of floats;

the `VectorLayout::SPAN` specifies the pointer and size of the array to be stored in the *vertex*; the memory alignment is set to 8 bytes; and `true` indicates that the memory is interleaved to allow concurrent memory access.

Line 9 specifies the output of the *vertex*, `out`. It is defined to be a 2D float array. Since only one output is computed at a time, it has a memory alignment of 4 bytes and is stored in the non-interleaved memory region.

Lines 10-11 define the dimension and padding of the workload assigned to a worker thread, which is necessary for traversing the assigned data.

Line 12 contains the coefficients r0, and r1, which are computed at compile time to reduce operations performed in runtime.

Lines 13-17 present the indexing function required to traverse a 3D mesh using a 2D array. This is often necessary when working with higher-order tensors in the IPU, as the input and output are limited to a maximum of 2D in the poplar programming model.

Finally, lines 19-32 depict the compute function, which includes the stencil equation shown in Equation 5.5. In lines 21-23, the aliases `t0`, `t1`, and `t2` are used for `in1`, `in2`, and `out`, respectively. This alignment allows the stencil equation to better correspond to the mathematical equation. Consequently, programming within *vertex* is made agnostic to the memory rotation discussed in Section 5.4.3. The memory assignment is dealt with in the previous control flow section, 5.4.3, when connecting the computed set to the tensors.

```
1  class WaveEquationSimple : public Vertex
2  {
3  public:
4    WaveEquationSimple();
5    Vector<Input<Vector<float, VectorLayout::SPAN, 8, true>>> in1;
6    Vector<Input<Vector<float, VectorLayout::SPAN, 8, true>>> in2;
7    Vector<Input<Vector<float, VectorLayout::SPAN, 8, true>>> damp;
8    Vector<Input<Vector<float, VectorLayout::SPAN, 8, true>>> vp;
9    Vector<Output<Vector<float, VectorLayout::SPAN, 4, false>>> out;
10   const unsigned worker_height, worker_width, worker_depth;
11   const unsigned padding;
12   const float r0, r1;
13   unsigned idx(unsigned x, unsigned y, unsigned w)
14   {
15     /* The index corresponding to [x,y] in for a row-wise flattened 2D
       ↪ variable*/
16     return y + x * w;
17   }
18
19   bool compute()
20   {
21     auto& t0 = in1;
22     auto& t1 = in2;
23     auto& t2 = out;
24     const unsigned padded_width = worker_width + 2*padding;
25     for (std::size_t x = padding; x < worker_height + padding; ++x){
26         for (std::size_t y = padding; y < worker_width + padding; ++y){
27             for (std::size_t z = padding; z < worker_depth + padding; ++z){
28            \\ Stencil Eqaution
29         }
30       }
31     }
32     return true;
33   }
34 }
```

Listing 5.2: codelete structure

$$
\begin{aligned}
u_{ijk}^{t+\delta} = \big(\quad & r_1 \cdot damp_{ijk} \cdot u_{ijk}^{t} \\
& + r_2 \cdot (-r_0 \cdot (-2.0 \cdot u_{ijk}^{t}) - r_0 \cdot u_{ijk}^{t+2\delta}) \\
& + 8.33333315 \times 10^{-4} \cdot (-u_{i-2,j,k}^{t} - u_{i,j-2,k}^{t} - u_{i,j,k-2}^{t} - u_{i,j,k+2}^{t} - u_{i,j+2,k}^{t} - u_{i+2,j,k}^{t}) \\
& + 1.33333333 \times 10^{-2} \cdot \big(\quad u_{i+1,j,k}^{t} + u_{i,j+1,k}^{t} + u_{i,j,k+1}^{t} + u_{i,j,k-1}^{t} + u_{i,j-1,k}^{t} + u_{i-1,j,k}^{t}) \\
& - 7.49999983 \times 10^{-2} \cdot u_{ijk}^{t} \\
& \big) \,/\, (r_0 \cdot r_2 + r_1 \cdot damp_{ijk}) \\
where \quad & r_0 = dt^{-2}, r_1 = dt^{-1}, r_2 = vp_{ijk}^{-2}
\end{aligned}
\tag{5.5}
$$

5.5 Programming Challenges

While the problem compilation time and data usage are mentioned in Section 4.4, this section documents other challenges in programming the IPU related to the Wave Equation.

When time order is increased for a stencil equation, additional logic is needed to handle the control flow, thus where data has to be streamed. This additional complication could be prone to errors, and care must be taken to design the control flow prior to coding, especially in understanding where data has to be streamed for the first iteration of the program.

The limitation of the IPU in supporting only 2D input and output also becomes apparent as more data of different sizes are needed to be addressed within the *vertex*. Data are connected to the *vertex* based on its access patterns. Using the example of Equation 5.5, u^t requires 2 neighbouring data at each direction, whereas $u^{t+2\delta}$ only requires data at position i, j, k. The size of the u^t array connected to the vertex would therefore be different to $u^{t+2\delta}$, and they will be addressed differently. As this is not natively supported by the programming model, programmers must take care to ensure the indexing logic is correct.

Chapter 6

Evaluation

After the wave and diffusion equations have been implemented both on the *Devito* compiler and the IPU, this chapter utilises the equations to evaluate the performance of the IPU.

The primary objectives of this section are to benchmark both the wave and diffusion equations on the IPU and address the following questions:

1. How does the IPU perform compared to other hardware?
2. How are the strong and weak scaling capabilities of the IPU?
3. To what extent increased space order affects throughput on the IPU?

In the following sections, Section 6.1 outlines the hardware and software experimental setup, which emphasises the reproducibility of the experimental results in the following section. Section 6.2 then describes the experiments performed on the equations and presents the result of the experiment to your reader

6.1 Experiment Setup

This section aims to provide the reader with the necessary information to reproduce the experiments. 6.1.1 describes the hardware used for the experiment; 6.1.2 summarizes the key software versions. These setups are used to evaluate both the diffusion equation and wave equation.

6.1.1 Hardware Setup

The problem was run on the M2000 machines with four *Colosus MK2 GC200 IPU*.

The performance of the IPU is compared to four commercially available hardware options, including 2 CPUs and 3 GPU architectures.

The *Intel(R) Core(TM) i7-10700KF* CPU and *NVIDIA GeForce RTX 3070 Ti* are included in the benchmark to represent the performance of HPC applications on common CPU and GPU hardware. *Archer2, NVIDIA Tesla V100*, and *NVIDIA RTX A6000*, on the other hand, represent stronger candidates in their respective categories and are designed for HPC applications. Table 6.1 provides a summary of the hardware benchmarked against the IPU.

CPU	**GPU**
Intel(R) Core(TM) i7-10700KF CPU	NVIDIA GeForce RTX 3070 Ti NVIDIA TRX A6000
AMD EPYCTM 7742	Tesla V100

Table 6.1: Benchmarking Hardware

Since archer 2 is quite distinct from a typical CPU, its hardware characteristics are summarised in a different manner compared to the CPU. Other wise, an overview of the hardware specification are included in table 6.2, 6.3 AND 6.4.

Intel(R) Core(TM) i7-10700KF CPU					
CPU(s)	16	**Socket(s)**	1	**L2 cache**	2 MiB
thread(s) per core	2	**NUMA node(s)**	8	**L1d cache**	256 KiB
Core(s) per socket	8	**L3 cache**	16 MiB	**L1i cache**	256 KiB

Table 6.2: Intel(R) Core(TM) i7-10700KF CPU Specification

Archer 2	
Nodes	5,860 nodes: 5,276 standard memory, 584 high memory
Processor	2× AMD EPYC^{TM} 7742, 2.25 GHz, 64-core
Cores per node	128 (2× 64-core processors)
NUMA structure	8 NUMA regions per node (16 cores per NUMA region)
Memory per node	256 GiB (standard memory), 512 GiB (high memory)
Memory per core	2 GiB (standard memory), 4 GiB (high memory)
Interconnect	HPE Cray Slingshot, 2× 100 Gbps bi-directional per node

Table 6.3: Archer2 Processor Specification

GeForce RTX 3070 Ti			
GPU Architecture	Ampere	**Memory Size**	8 GB
NVIDIA Tensor Cores	192	**Memory Bandwidth**	600 GB/sec
NVIDIA CUDA® Cores	6144	**Memory Interface**	256-bit
NVIDIA Tesla V100			
GPU Architecture	Volta	**Memory Size**	32GB/16GB HBM2
NVIDIA Tensor Cores	640	**Memory Bandwidth**	900 GB/sec
NVIDIA CUDA® Cores	5120	**Memory Interface**	4096-bit
NVIDIA RTX A6000			
GPU Architecture	Ampere	**Memory Size**	48 GB GDDR6
NVIDIA Tensor Cores	336	**Memory Bandwidth**	768 GB/sec
NVIDIA CUDA® Cores	10,752	**Memory Interface**	384-bit

Table 6.4: NVIDIA GeForce RTX 3070 Ti and NVIDIA GPU Specification

6.1.2 Software Setup

The experiment was performed with key software versions listed on table 6.5. An older version of python 3.10.8 is selected based on the requirement of running *Devito 4.8.1*.

Software Versions			
IPU Overall Software Version	2.5.0	**POPLAR version**	3.2.0
ICU firmware Version	2.4.4	**clang version**	15.0.0
Python	3.10.8	**Devito**	4.8.1

Table 6.5: Summary of Software Versions

6.2 Evaluation

Three separate experiments are conducted to measure various aspects of IPU performance.

6.2.1 Experiment Setup

The first experiment runs a fixed size problem with a space order of 4 on the IPU and the hardware listed in Section 6.1.1. The grid size is set to maximise the memory usage of a single IPU, which is 342^3 and 225^3 for the diffusion equation and wave equation, respectively. This experiment intends to compare the performance of IPU in comparison to other hardware alternatives. The norms of the resulting matrix from this experiment are compared to ensure the same operations are performed on all hardware platforms.

The second experiment also runs a fixed-size problem but varies the number of IPU used to compute the problem. This experiment is intended to measure the strong scaling capability of the IPU.

The last experiment varies both the number of IPU and problem size, where problem sizes increase almost linearly with the number of IPUs. This experiment measures the weak scaling capability of the IPU.

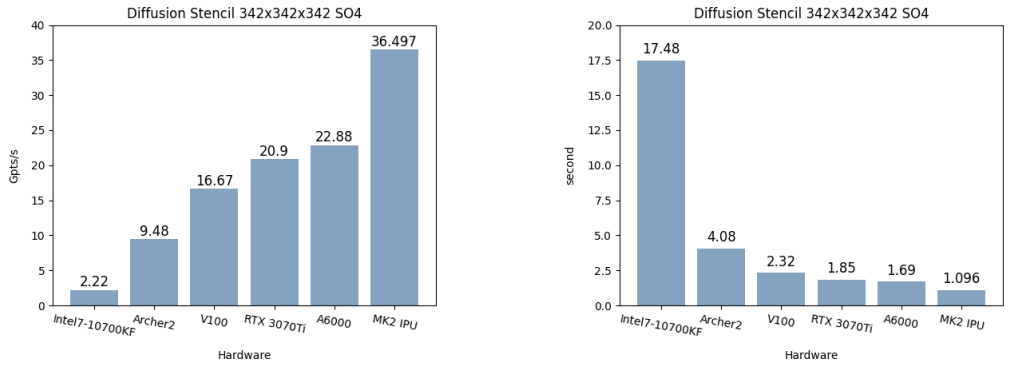
6.2.2 Hardware Performance Benchmark

This section presents and analyses the hardware performance of different computing platforms for solving the diffusion and wave equation. Notably, the IPU demonstrated a substantial performance advantage over all other alternatives in the context of the diffusion equation. However, when considering the wave equation, the IPU slightly underperformed in comparison to the NVIDIA A6000.

Diffusion Equation

Figure 6.1 shows the relative performance of the IPU against other hardware for the diffusion equation. This is shown through two figures. Figure 6.1a shows the throughput comparison, while Figure 6.1b shows the execution time comparison among the hardware.

The IPU was able to outperform all other hardware listed in this specific problem size and configuration. The closest contender, *NVIDIA RTX A6000*, was only able to achieve 62.7% of the IPU throughput. While other hardware namely, *NVIDIA RTX 3070Ti NVIDIA V100* , *Archer2*, and *Intel 7-10700KF* achieved 57.3, 49.6%, 28.2%, 6.6% of the IPU throughput performance respectively.



(a) Diffusion Stencil Throughput for listed hardware (b) Diffusion Stencil Time for listed hardware

Figure 6.1: Diffusion Stencil Performance BenchMark for listed hardware

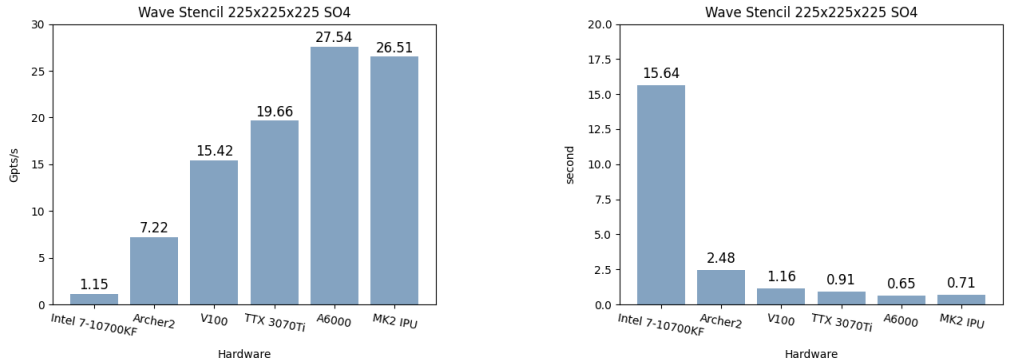
An interesting comparison could be made between *archer2*, *A6000*, and the *IPU*. While these hardware all contain large amounts of cores, they differ in their memory architecture. Although Archer2 contains a large memory, it has a relatively small in-processor memory, and the distance between the processor and memory is large. The *NVIDIA V100* has a relatively larger SRAM shared among

its clusters of processors, but it still needs to fetch from the global shared memory, which presents a memory fetch latency. IPU takes the other extreme; with 624kB of In-Processor-Memory, its processor has very close proximity to its memory, allowing for frequent fast memory access. All this means that the different hardware has different trade-offs between processor computing power and memory proximity, and they are suitable for different levels of *operational intensity (OI)*. This experiment shows that for a diffusion problem of size 342^3, space order 4 and with an OI of 3.38, the IPU is able to outperform similar hardware by a large margin.

While the IPU exhibited remarkable efficacy in calculating the diffusion equation, its performance was comparatively inferior when tasked with computing the wave equation.

Wave Equation

Figure 6.2 shows the relative performance of various hardware compared to the IPU in computing the wave equation. While the IPU still outperforms most other hardware, its performance is slightly worse than that of the *NVIDIA RTX A6000*. In fact, the *NVIDIA RTX A6000* has 103.89% the throughput of the IPU in the wave equation.



(a) Wave Stencil Throughput for listed hardware (b) Wave Stencil Exec Time for listed hardware

Figure 6.2: Wave Stencil Time for listed hardware

To better understand the performance of hardware across the diffusion and wave equation, Figure 6.3 shows the performance comparison of hardware in the wave equation against that of the diffusion equation. As shown, the IPU and CPU architectured hardware performance was greatly impacted when changing from computing the diffusion equation to the wave equation. While the GPU-architectured hardware mainly maintained its performance or even improved in the case of the NVIDIA RTX A6000.

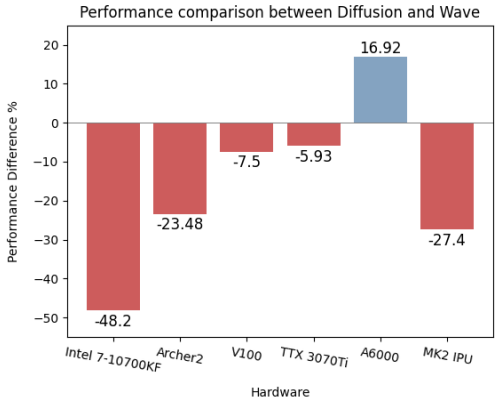


Figure 6.3: Throughput comparison between wave and diffusion equation

When comparing the stencil equations between the diffusion and wave equations. They differ mainly in the data access pattern and Operational Intensity. The wave equation retrieves data

from multiple meshes, while diffusion only retrieves from one. The Operational Intensity (OI) of the wave equation, 1.89, is also lower compared to the diffusion equation, 3.38. Since much more memory is needed for the wave equation, a plausible explanation would be that the GPU cache per block is big enough to accommodate enough data for it to exhibit the same performance.

This case cannot be said about the IPU. Examining the smallest partition in both problems, the wave equation necessitated a memory partition size of $9 \times 27 \times 27$, while the diffusion equation required a partition of $14 \times 42 \times 42$. In the wave equation, the surface area to volume ratio of the partition amounts to 37.06%, whereas in the diffusion equation, it is 23.8%. This discrepancy implies that the IPU must communicate much more with its neighbouring tiles when dealing with the wave equation. This increased communication likely contributes to the deterioration of the IPU's performance. To validate this hypothesis, more complex wave equations, requiring more meshes and hence more memory, should be tested. According to the theory, the IPU's performance would further decline in such cases. Nonetheless, despite the need for more frequent communication, the IPU still achieved comparable results to the *NVIDIA A6000* when computing the wave equation.

6.2.3 Strong Scaling Performance Benchmark

Table 6.6 shows the performance of a varied number of IPUs on running the diffusion problem of size 342^3 and the wave problem of size 225^3, both with a space order of 4.

For the diffusion equation, the speed up from one to two IPU was 81.1%, while from two to four was 63.6%. While for the wave equation, the speed up from one to two IPU was 87.36%, two to four IPU was 38.27%.

From these numbers, the IPU has exhibited a desirable strong scaling capability for the diffusion equation, as the performance increase is still substantial when the number of IPUs increases from two to four. This observation does not apply to the wave equation, as the performance increase from two to four IPUs is less than half the performance increase from one to two IPUs.

Equation	**Size**	**Space Order**	**Processor**	**No. of Unit**	**Time**	**Throughput**
Diffusion	342^3	4	MK2 IPU	1	1.096	36.497 GPts/s
Diffusion	342^3	4	MK2 IPU	2	0.605	66.108 GPts/s
Diffusion	342^3	4	MK2 IPU	4	0.370	108.140 GPts/s
Wave	225^3	4	MK2 IPU	1	0.71	26.51 GPts/s
Wave	225^3	4	MK2 IPU	2	0.38	49.67 GPts/s
Wave	225^3	4	MK2 IPU	4	0.27	68.68 GPts/s

Table 6.6: Throughput of 1, 2 and 4 IPUs with fixed problem size and space order

Problem	**No. of IPU**	**Smallest Partition**	**Surface Area to Volume Ratio%**
Diffusion	1	$14 \times 42 \times 42$	23.80%
Diffusion	2	$14 \times 21 \times 42$	28.57%
Diffusion	4	$14 \times 21 \times 21$	33.33%
Wave	1	$9 \times 27 \times 27$	37.04%
Wave	2	$9 \times 13 \times 27$	45.01%
Wave	4	$9 \times 27 \times 13$	52.99%

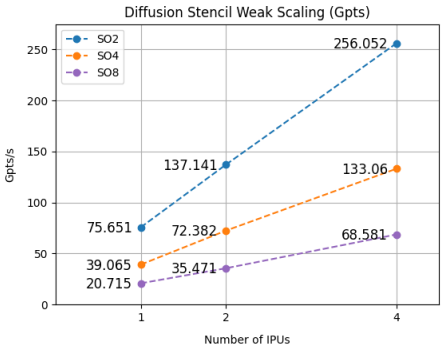
Table 6.7: Surface Area to Volume Ratio of Strong Scaling Experiment runs

The surface area to volume ratio (SATVR) of the partition also provides an adequate explanation in this case. Table 6.7 displays the partitions and SATVR values for various strong scaling experiments conducted. Since the SATVR for the wave equation is high to start with, it does not scale well in the strong scaling experiment. It appears that, in general, the IPU can only demonstrate desirable strong scaling capabilities when the SATVR of the partitions is initially low.

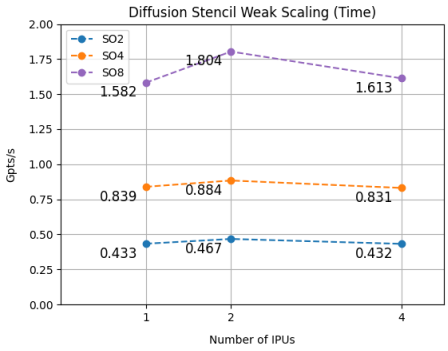
6.2.4 Weak Scaling Performance Benchmark

Figure 6.4 and 6.5 show the result of the weak scaling experiment for diffusion and wave equation, respectively. The problem size corresponding to 1,2 and 4 IPU is 320^3, 400^3, 480^3 for the diffusion equation, and 225^3, 280^3, 350^3 for the wave equation.

In the weak scaling experiment for the Diffusion equation, space order was varied alongside problem size. Its results showed a correlation between the space order of the problem and execution time. The problem size, on the other hand, does not appear to affect the execution time when supported with a proportional amount of IPUs. This shows that, on one hand, the IPU is able to scale well with the problem size, but also, the diffusion stencil computation is I/O bound.



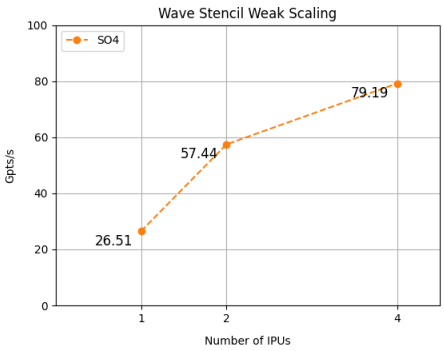
(a) Diffusion Stencil Throughput Weak Scaling



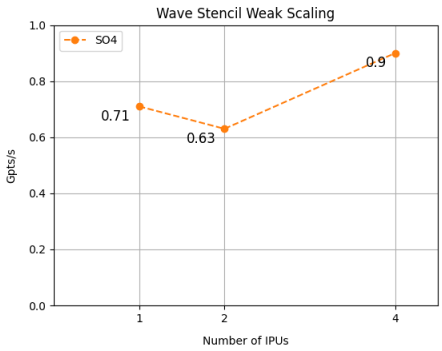
(b) Diffusion Stencil Execution Time Weak Scaling

Figure 6.4: Diffusion Stencil Weak Scaling Experiment Result

In the weak scaling experiment for the wave equation, the IPU exhibited decent weak scaling capability. When the problem size increases, the throughput per core is 28.72 Gpts/s and 19.80 Gpts/s for 2 IPUs and 4 IPUs. The throughput-per-core increased from one to two IPUs and remained reasonably close to the original throughput-per-core when it increased to four IPUs. It should be noted that when the number of IPU doubles, the problem size they support does not double. In fact, the problem size will be slightly less than double since there will be extra exchange codes generated to facilitate the extra data transfer.



(a) Wave stencil throughput with varied IPUs and problem size



(b) Wave stencil execution time with varied IPUs and problem size

Figure 6.5: Wave Stencil Weak Scaling Experiment Result

6.3 Conclusion

From the result of the 3 experiments performed. The following results are obtained.

In terms of the diffusion equation, the IPU surpasses other hardware by a significant margin. The performance throughput of the IPU is 62.7% more compared to the next best candidate *NVIDIA RTX A6000*. However, when it comes to the wave equation, its performance is slightly worse than the *NVIDIA RTX A6000*. The performance throughput of the *NVIDIA RTX A6000* was 3.89% more compared to that of the IPU.

The performance of the IPU depends on the memory density of the problem. In cases where greater memory is required to be stored in a tile, and the surface area to volume ratio of the partition is high, the performance of the IPU tends to diminish. This suggests that memory demands and partition characteristics play a crucial role in influencing the IPU's performance.

In general, the IPU exhibits decent weak scaling capability, and throughput-per-core is maintained when the number of IPU increases. However, the weak scaling capabilities of the IPU deteriorate when confronted with high memory density scenarios that necessitate increased communication between tiles.

The weak scaling experiment also revealed that the space order has a more substantial impact on performance throughput than the size of the problem itself.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Promising outcomes have emerged from the investigation of the IPU's performance in stencil computation. Notably, the IPU has outperformed the NVIDIA A6000 by 59.5% in the diffusion equation while performing similarly, with 3.7% worse performance, to the NVIDIA A6000 in the wave equation.

The IPU demonstrates commendable weak and strong scaling capabilities. In the weak scaling experiment, doubling the problem size and the number of IPUs resulted in a comparable execution time, highlighting the IPU's proficiency in weak scaling. In the strong scaling experiment, the IPU shows better scalability with stencil equations of higher Operational Intensity. For the diffusion equation, the performance throughput increased by 82.5% from one to two IPUs and by 69.2% from two to four IPUs. However, the performance gains were less significant for the wave equation, with an 87.4% increase from one to two IPUs and a 38.3% increase from two to four IPUs.

It has also been observed that the execution time of a problem positively correlates with its spatial order. For instance, when considering the same diffusion problem size, the average execution time increased by 105.3% as the spatial order rose from 2 to 4 and by 51.3% when it increased from 4 to 8.

Furthermore, the time order of a stencil equation contributes to the overhead of the IPU's control flow. To accommodate higher time orders, multiple memory buffers are utilised, and a rotation of the memory buffer occurs at each iteration to represent different time steps in the stencil equation. Optimising the IPU control flow favours a complete memory rotation, necessitating batch iteration in the program. Consequently, additional program logic is required to handle scenarios where the number of iterations is not divisible by the batch size.

Certain challenges arise during the compilation stage. Lengthy compilation times and substantial memory usage are observed, especially when compiling a diffusion problem size equivalent to that handled by 4 IPUs, which demands several hours and approximately 100GB of memory. Hence, an adequately equipped host system with ample memory and computing power becomes essential for effective IPU programming. Additionally, the absence of an efficient method to estimate the exchange code volume and the total memory required before compilation poses a significant problem. Programmers are incentivised to over-underestimate the maximum memory capacity that an IPU can accommodate to avoid OOM errors.

7.2 Limitation

The investigation in this project primarily focused on analysing the impact of space and time order on IPU performance and programming. However, there exist intriguing aspects within the scope of this project that warrant further exploration. These aspects are outlined below.

7.2.1 Holistic Measurement

While this project primarily emphasised the execution time of the program, it is equally crucial to consider other factors such as stream, load, and compile time. These factors contribute to a more comprehensive understanding of the overall programming experience. Stream time refers to the time required to transfer data to the IPU, and it is the most efficient approach for data transfer between host and device[14]. Load time, in the context of this project, pertains to the time needed to load an IPU executable from the host to the device. Given the substantial compilation time involved with the IPU, it is common practice to save and load executables when the same program for the device could be reused.

As mentioned in section 7.1, the compilation time is significant. A more holistic measurement should consider times other than execution time, which will better represent the overall time needed to solve a problem using the IPU.

7.2.2 Optimisation

While the *Devito* compiler inherently applies automatic optimisation techniques specific to the target hardware, optimisation was not implemented for the IPU program in this project. Consequently, it is anticipated that the performance disparity between the IPU and the hardware mentioned in Chapter 6 would be more pronounced for the diffusion equation. The IPU is also anticipated to perform better than the benchmarked hardware for the wave equation after optimisation.

One potential avenue for optimisation in the IPU program is vectorisation. Empirical findings by Simen Hopnås [9] have demonstrated a notable 65.5% improvement in performance by employing vectorisation on a single IPU for a 3D Heat Equation, resulting in a throughput increase from 0.87 to 1.44 TFLOPS, a 65.5% improvement in throughput. It would be interesting to investigate the impact of vectorisation on the IPU's performance within the scope of this project.

7.2.3 Scaling

The investigation in this project focused on exploring the weak and strong scaling capabilities of one to four IPUs. However, it would be intriguing to expand the experiment to include 8 and 16 IPUs to observe how the IPU scales beyond 4 IPUs. It should be noted that the host setup provided by Graphcore for 16 IPUs includes 400 GB of RAM. Therefore, readers should anticipate a similar configuration to ensure a reasonable compilation time when utilising 16 IPUs.

7.2.4 Comprehensive Wave Simulation

In Section 5.3, the `RickerSource` from the *Devito* compiler was used to perform source injection. Once the process of source injection was completed, it was removed, and the wave stencil kernel was evaluated. Ideally, a more advanced setup would also include a benchmarking of the source injection alongside the wave stencil in the IPU. The set-up would have more resemblance to an application-level wave propagation simulation, and the result obtained from the set-up would also generalise better to HPC applications. This was left as future work due to the additional complexity it involves.

7.3 Future Work

This section outlines potential areas for future research that extend beyond the scope of this project but are useful for the application of the IPU in the HPC context.

7.3.1 IPU Code Memory Estimation

As highlighted in Section 7.1, there is currently no established method for accurately estimating generated code memory usage within the HPC domain [15]. The suggested approach is to start with a small problem size and observe memory usage as the problem size increases. It would be valuable to investigate how the exchange code volume varies with communication volumes, and vertex code volume varies with the *codelet* used to program the vertex. Such analysis would ease future developments that aim to utilise the full extent of the IPU memory.

7.3.2 Benchmarking More Complex Wave Equations

In addition to the Isotropic wave equation examined in this project, it is worthwhile to benchmark more complex equations on the IPU that represent application-level scenarios. Examples of such equations include the anisotropic acoustic [16] and isotropic elastic equations [17]. These equations pose more intricate memory access patterns and arithmetic intensity, thus offering a more realistic assessment of the IPU's performance in HPC problems encountered in real-world applications.

Chapter 8

Ethical Issues

This projects uses IPUs to optimise processes in high performance computing, and in particular, stencil computation. The projects revolves around optimising such computation, and reducing computational latency. There is little ethical issues revolving the improvement of Devito.

This projects aims to maintain a high level of transparency to the best of the author's ability. Proper citations are included for reference to support claims and uphold academic integrity, while information regarding experiments are disclosed to ensure reproducibility.

The project prioritizes open science by providing details regarding the procedures and necessary equipment to reproduce the results. This approach maximizes the reproducibility of the experimental findings, and aims to facilitate meaningful academic discussions between parties.

Bibliography

[1] "Graphcore ipu hardware overview." https://docs.graphcore.ai/projects/ipu-overview/en/latest/about_ipu.html#ipu-hardware-overview.

[2] G. Bisbas *Automated cache optimisations of stencil computations for partial differential equations*, p. 1–208, Aug 2022.

[3] P. Micikevicius, "3d finite difference computation on gpus using cuda," in *Proceedings of 2nd workshop on general purpose processing on graphics processing units*, pp. 79–84, 2009.

[4] K. Rocki, D. Van Essendelft, I. Sharapov, R. Schreiber, M. Morrison, V. Kibardin, A. Portnoy, J. F. Dietiker, M. Syamlal, and M. James, "Fast stencil-code computation on a wafer-scale processor," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, IEEE, 2020.

[5] T. Louw and S. McIntosh-Smith, "Using the graphcore ipu for traditional hpc applications," in *3rd Workshop on Accelerated Machine Learning (AccML)*, 2021.

[6] G. Ltd, "Ipu technology." https://www.graphcore.ai/products/ipu.

[7] R. J. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations*. 2007.

[8] T. D. C. Group and I. C. London, "Devito." https://www.devitoproject.org/, 2016.

[9] S. Håpnes, "Solving partial differential equations by the finite difference method on a specialized processor," Master's thesis, 2021.

[10] Cerebras, "Cerebras - wafer scale engine." https://www.cerebras.net/product-chip/, 2023.

[11] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, *et al.*, "The landscape of parallel computing research: A view from berkeley," 2006.

[12] https://f.hubspotusercontent30.net/hubfs/8968533/Cerebras%20SDK%20Technical%20Overview%20White%20Paper.pdf.

[13] G. Bachler, A. Bregant, and H. Schiffermüller, "A parallel fully implicit sliding mesh method for industrial cfd applications," Sep 2007.

[14] Graphcore, "Graphcore tutorial 1." https://github.com/graphcore/examples/tree/master/tutorials/tutorials/poplar/tut1_variables, Mar 2023.

[15] "Mapping a model to an ipu system - memory and performance optimisation on the ipu," *3. Mapping a model to an IPU system - Memory and Performance Optimisation on the IPU.*

[16] Y. Zhang, H. Zhang, and G. Zhang, "A stable tti reverse time migration and its implementation," *GEOPHYSICS*, vol. 76, p. WA3–WA11, May 2011.

[17] J. Virieux, "Elastic wave," *GEOPHYSICS*, vol. 51, p. 889–901, Apr 1986.