University of Bergen
Department of Informatics

Exploration of Linear-Algebra Graph Algorithms on the Graphcore IPU

Author: Yoeri Otten
Supervisors: Fredrik Manne[1] & Johannes Langguth[2]

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

November, 2023

[1]University of Bergen
[2]Simula Research Laboratory

Abstract

More specialized hardware is being created to improve the efficiency of demanding algorithms. Graphcore introduced their Intelligence processors: a specialized true MIMD architecture which specializes in machine learning problems. Earlier work has shown that this architecture can also be used to solve breadth-first search competitively. We explore the possibilities of this machine by implementing several graph algorithms on this machine. Specifically, we look at Sparse-matrix vector multiplication and its usage within an algebraic approach to the breadth-first search algorithm, and show an algebraic solution to Prim's algorithm on the machine. Finally, we discuss the possibilities and problems of the Graphcore IPU.

Acknowledgements

I'm definitely not a researcher in any sense of the word, at least I've figured that out over the past year. Starting this adventure meant diving in a completely new world for me, I knew that but was still surprised with the roadblocks which the project threw at me. Working with a new product, one which is barely being used (I counted!) meant that the usual safety nets which I relied on were useless - and the ones I could rely on I didn't rely on enough. I would like to thank my UiB supervisor Fredrik Manne for listening to me complain one to many times. I haven't been a great student over the past year but he was certainly ready to listen and help when asked, or when I should've asked. I would also like to thank my other supervisor Johannes Langguth and his PhD-student Luk Burchard whom helped me immensely with using the Graphcore IPU, giving great insights into its capabilities.

I do feel truly grateful with what I worked with in the past year: true novel hardware which introduces a load of new possibilities for developers and researchers to work on, with thanks to Graphcore. I hope that these approaches and ideas could spark some new focus on their non artificial intelligence tooling.

Finally, I would like to thank my family, friends and girlfriend whom helped me mentally in the past year.

`Yoeri Otten`

Monday 20$^{\text{th}}$ November, 2023

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In 1996 TomTom introduced their first route planning software for the masses: a simple product in which the user could fill in a start location, destination and a few seconds later a route with instructions would show one exactly where to go. Now, more than 25 years later, it is almost impossible to think of a world where just an analogue map would be used. We have become accustomed to opening up our phones and following the instructions given.

But how do our phones know where to take us? On analogue maps it can be a daunting task planning even a small trip of just a few kilometers, let alone one which would take us to the other side of the globe. Yet, our phone instantly give us a list of highly detailed instructions.

Over the past decades researchers have been been inventing and improving algorithms which calculate these routes. Together with the improvements made to general computers, this has allowed us to perform calculations on even the most detailed and largest data sets. For instance, the OpenStreetMap database which contains almost 10 billion different nodes and connections visualizing the road network of the world [5].

Nevertheless, processing that many possibilities is still a demanding task, and while our demands and usage of this technology keeps growing, we start to see limits in the improvements of general use processors [13].

The clock rate of processors has not improved in the last two decades [12], and most of the speed improvements have come through the use of smaller process dies allowing more transistors on a chip [15]. Moore's law, an observation by Intel co-founder Gordon

Moore in 1965, predicted a doubling of the amount of transistors on a chip every two years. This has mostly been the case over the past decades but is starting to be stopped by constraints when working on an increasingly smaller scale. Recently, CEO of NVIDIA has also pronounced Moore's law to be "dead".

Therefore, instead of relying on generalized hardware like the CPU and GPU, many have now started to shift focus to more specialized hardware which can be optimized for specific problems. This has especially been the case with the rise of machine learning.

In 2018 Graphcore presented their first-of-its-kind Intelligence Processing Unit (IPU)[9], a processor optimized for machine learning tasks like training models and classifying input. However, the capabilities of the processor go beyond that, being a true multiple instruction, multiple data architecture [8].

Or in layman's terms: it allows us to run multiple different programs in parallel each of which able to simultaneously access and write data.

The IPU achieves this by having local memory available to every single core on the machine, thereby also minimizing memory lookup time which is often constrained on traditional hardware.

With any piece of specialized and novel hardware we do see some drawbacks, like the lack of in-use tooling and the strict memory limitations present in the IPU.

Earlier work has mainly been focused on the machine learning capabilities of the IPU, but some work has been done for the implementation of classical algorithms. For instance the breadth-first search algorithm which was first successfully implemented on both a single- and multi-IPU system [4, 3] using a linear-algebra approach.

In this thesis we expand upon this work by exploring a more general approach by implementing Sparse-matrix vector multiplication (SpMV) and adapting it to perform Breadth-first search. SpMV is important building block upon which many other graph algorithms can be built, giving us an insight into how well adapted the IPU is to those kind of problems. Furthermore, we also take a look at an algebraic algorithm outside of this technique: Prim's algorithm which computes the minimum spanning tree of a weighted undirected graph.

Our thesis is ordered as follows: first, we discuss the field of parallel programming and the contemporary hardware used. We then introduce the Graphcore IPU, by explaining its model, capabilities and shortcomings.

We then introduce the three algorithms we implemented on the IPU one by one, starting with Sparse matrix-vector multiplication. In that chapter we also give an introduction to our setup. For each algorithm we discuss the algorithm, its translation to an IPU runnable program, and finally we perform tests and compare those to other results.

Lastly, we discuss our results, and summarize our findings on how effective the IPU is as a solution to graph algorithms.

Chapter 2

Parallel computing

In chapter 1 we talked about working with large data sets. Depending on the type of algorithm used this does not need to be a cause of concern: many algorithms scale linearly, and while the compute time takes longer on larger inputs, these algorithms can still be performed quickly on large amounts of data.

We can denote this scaling using Big O notation. Big O notation shows the time it takes to run an algorithm based upon its input size, or the algorithmic complexity. The exact definition is a bit more complex but has been left out for brevity.

A linear algorithm can be denoted in Big O notation by $\mathcal{O}(n)$ where n is the size of the input. For example the length of an array.

However, not all algorithms scale linearly, and some will instead scale quadratically or even exponentially. This can be denoted by for example $\mathcal{O}(n^2)$. This means that with each unit of extra data to compute, the rate of change of extra compute necessary increases.

Such algorithms quickly become unmanageable if the input size is large enough. Instead we need to find other approaches to get our wanted result. For example by estimation, or decreasing our input data. One possibility is to make parallelize the algorithm, here we split the problem up so that we can work on multiple parts independently and concurrently.

We can call the amount of independent jobs we can run independently of each other p. If we have a quadratic algorithm which we can divide up into p different tasks we might get a new algorithmic complexity of $\mathcal{O}(n^2/p)$, however this depends upon the algorithm.

Parallel computing is thus the act of breaking down a compute problem and computing results for the different parts simultaneously instead of linearly.

So then the question becomes: how do we divide our problem into multiple sub-problems such that we can maximize our usage of a parallel system, and p tasks that can run simultaneously?

In this chapter we discuss the basics of parallel computing, Flynn's taxonomy classifying different types of computer architectures for parallel computing and how these are implemented in modern computing devices.

2.1 Designing parallel algorithms

Designing a parallel algorithm is about finding a way to efficiently split a problem up into sub-problems which can be solved independently and be combined to give an answer on the original problem.

Another problem to keep in mind when designing these algorithms is thread communication. An efficient method needs to be found to assign tasks to different compute instances, preferably maximizing usage of the compute unit.

There are some generalized techniques which are often used when designing these kinds of algorithms.

2.1.1 Divide and Conquer

The Divide and Conquer technique describes the basic approach of looking for ways to split up the input data of the algorithm. If that can be achieved than all split sections can be computed in parallel. The difficulty lies in finding a suitable

A great example of a divide and conquer algorithm is Merge Sort. Merge sort sorts a given input list by splitting the list in two and sorting those independently, and finally merging the two resulting lists together. A synchronous implementation of this algorithm has complexity $\mathcal{O}(n \log n)$ while a parallel implementation can achieve a complexity of $\mathcal{O}(\frac{n \log n}{p})$.

2.1.2 Task parallelism

In task parallelism we look at the different steps performed by an algorithm to try to find steps which can be performed independently of each other.

An example could be finding the highest and lowest value in your input data. In that case the two steps do not depend upon each other and can be performed in parallel.

Modern architectures as discussed in the CPU and GPU section below often implement this as an optimization on low level tasks. A program can for instance continue running while performing a slow memory lookup as long as that data is not yet needed.

2.1.3 Data parallelism

Data parallelism is another method which is often found as an optimization in modern architectures. If the same calculation has to be performed on a range of data, this calculation can be performed in parallel.

2.1.4 Parallel loops

The same idea holds for looping over data, as long as each iteration is not dependent upon another iteration (or is predictably dependent), the loop can be performed in parallel.

2.2 Flynn's taxonomy

In section 2.1 we discussed different approaches to parallelize an algorithm. Which approach should be used is highly dependent on the type of architecture is used as some approaches might work better on certain architectures compared to others.

To generalize over these distinction we introduce Flynn's taxonomy [1]. Flynn's taxonomy tries to classify these different possible computer architectures in four basic categories:

- *Single instruction, single data* (SISD): SISD is the most simple of architectures, allowing only for a single data stream to be worked on sequentially. There is absolutely no parallelism. Memory operations can only be performed on a single element of memory and only one instruction can be executed at the same time.
- *Single instruction, multiple data* (SIMD): These architectures do not allow simultaneous execution of multiple instructions but can perform the same instruction on a (limited) range of data. In cases where the same kind of operations needs to be performed on a sufficiently large amount of data this can already result in significant speedups.
- *Multiple instructions, single data* (MISD): MISD is a very uncommon architecture where multiple instructions can be run simultaneously but only on a single unit of data.
- *Multiple instruction, multiple data* (MIMD): MIMD allows for full parallelism, where multiple instructions can be executed simultaneously all working with different elements of data.

2.3 The CPU and the GPU

Two of the most used architectures are the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU). They can be found in almost any machine around the globe, including the machine that this sentence is written on, and if not printed - the machine that this sentence is read on right now.

2.3.1 Central processing unit (CPU)

The CPU is a core component of the computer and an example of a SISD architecture, but more modern hardware can even be described as having a MIMD architecture. While the CPU can run multiple different programs on its different cores, it is still severely limited by the amount of cores and the shared memory buss. A CPU core has no local memory instead sharing with the other cores, meaning that all cores can normally only perform a single lookup at the same time. Typical CPUs contain a around 8 to 16 cores, while more powerfull hardware exists this is currently limited up to around 128 cores per die [2].

In general, CPUs do not have onboard memory which means that memory lookup can be quite slow. Instead, any information in memory first has to be loaded into a register before an action can be performed. Loading from memory can sped up by the use of multiple layers of caching. This requires that the program is optimized to make use of this, for instance by lying related data together in memory. Most CPUs are also able to use out-of-order execution to to continue running instructions while waiting on data from memory.

More recently limits hit in the process of die-shrinking (the process to decrease the size of transistors on a chip thereby decreasing power usage and allowing for speed improvements) [14], have forced manufacturers to focus `more` on multi-core performance. This was partially achieved by threading, allowing more instructions to be executed simultaneously. And by introducing instructions which perform actions on multiple values in memory (SIMD). It achieves this by using larger registers which can hold multiple values at once, and special operations which can perform operations on these multi-word registers. These are however only modest improvements since these wider registers typically only allow up to four words of memory at a time.

2.3.2 Graphical Processing Unit (GPU)

The GPU was initially introduced to offload rendering tasks from the CPU. Nowadays, the GPU is a highly parallel processor being able to perform calculations on large blocks of memory simultaneously using its high thread-per-core count [11], and high core count. Some machines can contains hundreds to thousands of cores. The architecture is an example of SIMD. Memory is often located near the chip, and connected through a performant memory buss allowing much quicker memory lookup times than is possible without on-die memory like most CPUs.

This parallelizability does come at the drawback of a more limited instruction set, and less precision. For instance 32-bits floating point operations instead of 64-bits which can be achieve on a CPU.

The memory of a GPU is split up into three different kinds: global, shared, and constant memory. Global memory is memory shared globally between all threads on the GPU and behaves similarly compared to memory on a CPU. While shared memory is shared between a thread block, threads which are executing together.

Groups of threads, commonly called Warps execute the same flow-behaviour together to allow executing calculations on vasts amount of memory simultaneously.

The SIMD architecture makes the GPU applicable to problems which are easily parallelizable into many very-similar sub-problems.

2.3.3 Comparison

The CPU is a more versatile processing unit, but does have drawbacks when it comes down to parallelizability. Due to the lack of cores and smaller and slower memory bus it can be difficult to perform many calculations on memory. But, this does bring more versatility, allowing many different programs to be performed simultaneously while a GPU is constrained to performing a single operation per warp.

To overcome these shortcomings CPU's make more use of caching, allowing memory operations to perform more quickly.

Comparing the two architecture we see that both have their strengths for different kind of problems: when a problem is harder to divide into similar blocks a CPU is often preferred, while easily dividable problems are can benefit from the more parellelized GPU.

Chapter 3

The Graphcore IPU

In this chapter we will introduce the Graphcore IPU and take a look at its features, capabilities and shortcomings.

To overcome limitations introduced by generalized hardware one option is to create specialized hardware for a problem. This allows the architecture to be designed around the problem which would otherwise not have been possible.

One field using such specialized hardware is the filed of machine learning, in the form of an AI accelerator. Training models takes a long time and much energy. It is estimated that the training of large-language model GPT-3 took a total 1287 MWh which is equivalent to the yearly energy usage of around 200 households[1].

This created a market for hardware dedicated to machine learning, allowing these extra constraints to save energy and time.

Graphcore in 2017 introduced their solution: the Graphcore IPU, a true MIMD architecture. This IPU was replaced in 2020 by an improved version, allowing higher clock speeds, more memory per tile and a higher tile count. Currently, Graphcore is working on releasing their MK3 IPU which is expected to double the amount of compute power.

[1] Assuming an average daily usage of 17KWh per household

3.1 Architecture

To date Graphcore has introduced multiple architectures based upon the same principles. In this section we will be discussing the IPU-POD64 IPU machine [10] containing 64 of their Mk2 GC200 IPUs, their latest generation of IPUs which came out in 2020. While the general architecture is the same the technical specs like tile count, memory size and band-with differs between generations.

3.2 The IPU

The Intelligence Processing Unit (IPU) is a uniquely highly-parallelizable MIMD chip, allowing to simultaneously perform many different programs at once each with their own local memory.

3.3 Hardware layout

The IPU consists of a central control unit and 1472 cores or tiles as they are called laid out in a grid connected to a central buss for communication between the exchange mesh and the other tiles. A drawing of this layout can be found in figure 3.1.

Each tile is a core capable of running six simultaneous threads running in a round-robin fashion and contains its own local SRAM memory of 624kB. The central control unit also has its own "Streaming Memory" of up-to 64GB which can be used to line-up problems to be run on the tiles and for (cached) communication with the host machine.

The IPU uses its own limited instruction set. The construction set consists of instructions for reading and writing to memory, arithmetic operations including 32-bit and 16-bit float operations, control flow operations and instructions to generate random numbers including random floats [7].

A tile executes in fixed order and has no capability of out-of-order execution. However, since each tile has its own local memory for which it only takes six clock cycles to read and write to it has no use for it either since the data will generally be available when the next instruction of a thread is executed.

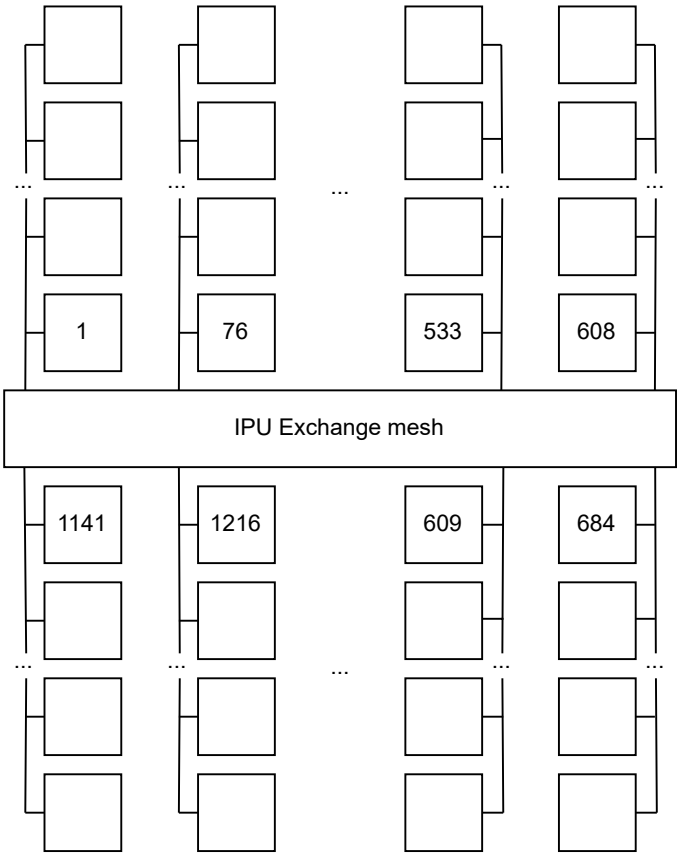


Figure 3.1: Layout of the different cores and their interconnection on the IPU die. The number shown shows the identifier of the core.

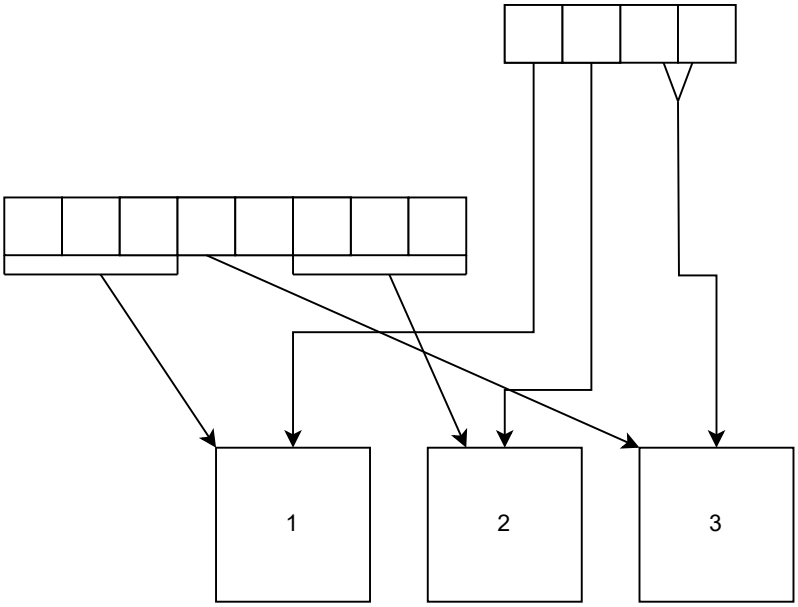


Figure 3.2: A schematic example of how different Tensors can be connected to different tiles during a compute step. The small squares are values in a Tensor, which are assigned to tiles indicated by the big squares.

3.4 Development overview

The IPU has it's own local types and instruction set, independent of that of the host computer.

For programming on the IPU the Poplar Software Development Kit (SDK)[2] was introduced. As a general overview: using this library, code for the IPU can be written using a specialized instruction and type set. These components are called `codelets` and can be compiled using the `popc` compiler which a program uses to build `Vertices`. Vertices are connected to Tensors which contain data. Next, we need to define a compute graph. This consists of coupling tensors to the different vertices and defining in which order they should run.

An example of how tensors can be connected to tiles can be found in figure 3.2.

During runtime a Poplar program compiles `codelets`, and calculates a graph. This generates the programs which can run on the IPU and the instructions necessary for memory movements during global steps.

[2]An SDK is a collection of tools, libraries to facilitate the development process.

3.5 Constraints, limitations and advantages of the architecture

The IPU is one of the first machines to actually give us a true MIDI-architecture, which allows us to perform many *different* calculations divided over its many tiles. This is also possible on a CPU but very limited due to constraints like the amount of tiles available (1472 for our specific instance) and the memory-bandwith which is shared between all the tiles. The GPU-architecture does allow you to perform massive parallel calculations but is constrained in the fact that there's only a single program-flow limiting you in what tasks you can perform simultaneously while also being limited by the memory-bandwidth.

This makes the IPU best focused on applications where a problem is highly-parallelizable with large enough sub-problems or where it is necessary to perform many different tasks to solve a problem. The fact that the IPU does not share control-flow between its processors also makes it possible to divide up work better since tiles do not need to wait upon other tiles' work to be done within a control-flow block to be able to continue with its own workflow.

However, this parallelizability does come at the cost of memory size. Each tile only has less then a megabyte of memory available. While, the memory limit can be increased by using multiple IPU's simultaneously, this makes the memory exchange slower, being only a maximum of 64 Gbps between IPUs and 1 Tbs internally. Thus, it is either necessary for problems to be highly parallelizable - meaning that many parallel tasks can be performed without much communication or very limited in memory size.

3.6 Usage statistics

The IPU is a relatively new device, having first been launched in 2017 and marketed towards an audience of artificial intelligence and machine learning research. This makes real-world usage outside of these domains - like we are doing - very limited. To assess real-world usage we have turned to usage on GitHub. Using the search functionality on the site one can find strings used in codebases over all the publicly available code on GitHub. Using the search term "poplar/Engine.hpp", and filtering out any code written by Graphcore self gives us around 61 different projects [3]. In comparison, the whole of GitHub contains over 300 thousand files written for NVIDIA's CUDA platform.

[3]The exact search on GitHub was done as follows: `"poplar/Engine.hpp -owner:graphcore -owner:graphcore-research NOT is:fork"`

3.7 Developing for the Graphcore IPU

Graphcore has chosen to introduce their own tooling for programming the IPU: the Poplar SDK. Self described as the first toolchain for creating graph software. It contains functionality for writing code which is directly run on the IPU, controlling the IPU hardware, auxiliary helpers to make common steps easier, and bindings for many of the popular Machine Learning and Artificial Intelligence libraries.

For the execution we can differentiate between 3 main ingredients of a Poplar program: the compute tasks, tensors containing our data and the compute graph which connects the execution steps and tensors together to create an executable program.

3.7.1 Tensors

A tensor is a (multi-)dimensional array which lives on one or multiple tiles of the IPU. The tensor type is introduced to create an easy interface for dividing data between the different vertices, and managing variables that live across compute sets.

3.7.2 Codelets

Codelets are the smallest building stones for programming on the IPU. They allow you to write code which directly compiles to machine language on the IPU. Codelets can be written using the Poplar SDK in C++ and compiled using the `popc` compiler. Alternatively they can also be directly written in assembly in cases of shortcomings of the library or when extra speedup can be achieved. Six codelets can be assigned per tile during execution, making use of the multithreading which is discussed below.

Codelets have associated variables which are stored in memory, these can be coupled to tensors during execution.

An example codelet looks as follows:

```
1  class Example : public Vertex
2  {
3  public:
4      Input<Vector<float>> in;
5      Output<Vector<float>> out;
6
7      auto compute() -> bool
8      {
9          // perform calculation
10
11         return true;
12     }
13 };
```

3.7.3 Poplibs library

The Poplibs library contains pre-made codelets and routines with the goal of making common routines easier to implement. They are preferred over implementing a solution yourself since the codelets are highly specialized and optimized for the task.

Some examples of the operations are:

- Reductions
- Simple expressions
- Sorting

3.7.4 Control flow

The SDK contains some functionality to apply control flow to the execution of our codelets and other compute tasks.

In our work we mainly used the following:

- `Sequence`: A sequence combines a number of compute sets which will be executed in a linear order.
- `Copy`: The Copy execution allows to copy between tensors or from/to the host machine.
- `Repeat`: Will repeat its execution an indicated amount of time.
- `RepeatWhileFalse`

3.7.5 IPU model

For debugging a simulator is available called the IPU model. The model allows to run your code in an IPU like environment on any computer. While this makes programming for the IPU more portable this does not take away some of the other constraints as discussed on the pitfalls section.

The IPU model also lacks some features like random number generation.

3.7.6 Multithreading

In general each instruction on a tile takes six clock cycles to execute. Since no out-of-order or speculative execution is possible Poplar opted to execute multiple programs at once on a single tile in a round-robin fashion instead of wasting the remaining cycles.

From a developing perspective there are multiple ways of using this functionality all with their own advantages and drawbacks:

- Allocating multiple independent vertices to a tile
- Using a `MultiVertex`
- Manually running threads using the `SupervisorVertex`

The simplest option for multithreading is to divide the workload up into more vertices. Each tile can execute a total of six vertices at the same time. This does have some drawbacks, mainly the cost of extra memory. Each vertex will take up their own place on the tile, with their own sections for data. Next to that this might also make it more difficult to collect our result which is often a single tile operation. It can also be difficult to divide up the work efficiently over the six times as many spots which need to be filled depending on the algorithm.

Instead we could use a `MultiVertex` to use up the full six threads on the tile. The `MultiVertex` is similar to a `Vertex` with the difference that the `compute` function is called once for each thread, with their thread id (unique number between 0 and 5 identifying the thread on the tile). The data is shared between all of the threads.

The biggest downside to the `MultiVertex` is the lack of collection possibilities. For instance its not possible to combine the results of the different threads into a single result since there is no functionality to sync on a thread level and thus work completely independently from each other. Next, there is no memory safety on a thread level: all Vertex variables are shared between the threads without any locking functionality. This means that a race conditions could occur when reading and writing to the same variable. However, we can use the fact that the execution is predictable to our advantage to for example pre-initialize certain variables by a single thread before any of the other threads would have had a chance to use it. Another way to collect the results is by executing a singe `Vertex` after the execution on the tile is done. An example of a `MultiVertex` can be found in listing A.2.

In cases where we need to collect the results of the different threads at the end of our program a `SupervisorVertex` can be used. A `SupervisorVertex` is more powerful than a common `Vertex` in that it has complete control over the Tile, but is severely limited in its instruction set. For example lacking many common floating point operations. This limitation means it is not possible to perform comparisons on floating point values, something which is commonly done when collecting a set of floating point results. A common technique for multi threading using a `SupervisorVertex` is for the vertex to run threads using the `runall` assembly call. However, it is also possible to run different threads individually. Afterwards, the result of the different threads can be combined. An example of such a program can be found in listing A.9.

Overall this creates a good set of options for pulling the most out of a single tile. However, the way multithreading is currently incorporated does bring some challenges to the end user. For instance the `SupervisorVertex` option requires one to use assembly to spawn the threads, and is not documented in any official documentation from Graphcore. Next to that the `SupervisorVertex` is severely limited in functionality, which while documented [7] is difficult to debug due to the lack of proper debugging capabilities on hardware and in the IPU model.

3.7.7 Pitfalls

The IPU has some pitfalls and problems which one should be wary of during development. This list is by no-way comprehensive but shows the problems we encountered working with the Poplar library.

Debugging

The debugging options on the IPU are limited. There is no way to attach a debugger to code running on the IPU or in the model, however there is an undocumented way of printing to the standard output of the host machine even when running on the IPU:

```
1 #include "print.h"
2
3 printf("%d\n", data[1]);
```

Poplar will include the vertex id and thread id in the output.

The Poplar library also does some type checking to make sure that tensors and codelets share the same types, and that connections to the host machine share the same type. Poplar will also check for initialization of tensors and that they are fully connected to the codelets.

Out of bounds memory

The IPU and the model have only limited protection against reading and writing in memory out of bounds of a tensor. Instead the data will instead be written or read from initialized memory or might be writing to another tensor.

Memory access and multi-threading

There are no memory guarentees when operating in a `MultiVertex` context. Accessing and writing to shared memory will not apply any locking, or other race condition checking. Every memory operation is applied per word, which is a total of 32-bits on the GC200 IPU. This causes memory problems when writing and reading multiple variables in the same word in different threads. Let us take a look at the following example:

```
1  class Example : public MultiVertex
2  {
3  public:
4      Output<Vector<bool>> data;
5
6      auto compute(unsigned threadId) -> bool
7      {
8          for (auto i = threadId; i < data.size(); i+=
               ↪ MultiVertex::numWorkers()) // n. of threads
9          {
10             data[i] = true;
11         }
12
13         return true;
14     }
15 };
```

A boolean is represented as 8-bits in the architecture, thereby having 4 values aligned on each word. This causes undefined behaviour since multiple threads will try to write to the same memory address at the same time. A possible solution would be to divide the work per word, thereby having only one thread operating on a single line of memory.

Usage of a `SupervisorVertex`

As mentioned in the previous section the `SupervisorVertex` is severely limited in instruction set, but also in stack size. This can create problems which are difficult to grasp. Let us for instance take the following codeblock written inside of a `SupervisorVertex`:

```
Vector<float> values; // Pre filled values

if (values[0] > values[1])
{
    // Do something
}
```

As mentioned, a `SupervisorVertex` has no instructions for comparing floating point variables in this context. We should therefore expect either a compiler exception or a runtime error which would indicate that we are performing an illegal instruction. Instead this code will result in a stack overflow on the hardware that we are using. This is due to how the backend compiler handles such cases: it will instead try to replace the instruction with an inbuilt function to compare float variables, however since the stack size is very limited in the supervisor context this will result in a stackoverflow.

Loop optimizations

The architecture contains two instructions meant for reducing the amount of instructions necessary for loops. The `rpt` and `brnzdec` instructions [7].

The `rpt` instruction tells the IPU the amount of blocks that need to be repeated and how many times, thereby allowing repetition with the cost of only a single instruction which is only executed once at the start of a loop.

The `brnzdec` instruction combines three operations:

1. Subtracting one from a register
2. Comparing the register to 0
3. Jumping back to the start of the loop in case of a 0

To make efficient programs it is important that these calls are used to save as many cycles as possible. Especially on long loops every instruction call less saves good time since each instruction is performed in constant time. However, these instructions are hard to optimize for by the compiler due to a few constraints:

- A limit on the number of iterations defined by the size of the internal registers. Especially for a `rpt` instruction where the internal counter is only 16-bits.

- No internal calls to other functions may be made in the loop.
- The loop must be setup in a way such that the compiler can infer its count.

These constraints, while often met are not always entirely obvious the compiler to optimize for. For instance for the use of the `rpt` instruction the compiler needs to infer the maximum amount of iterations. One can circumvent this by using the internal `rptsize_t` size type, which hints the compiler.

Chapter 4

Sparse-Matrix Vector multiplication

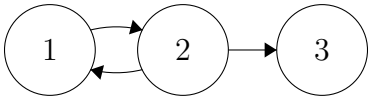
Graph algorithms and linear-algebra have always been tightly connected. A graph can be represented by a matrix in the form of an adjacency matrix. Every row and column in the matrix represent the in and outgoing edges for each vertex. An arbitrary value or 1 and 0 for unweighted graphs and the weight for weighted graphs. So, for a graph $G = (V, E)$ we can define a matrix $A = R^{|V|*|V|}$ with $a_{ij} = 1 \iff (i, j) \in E$ or 0 otherwise.

This representation does bare some limitations. An adjacency graph can for example not easily represent a multi-graph where vertices can be connected through more than one edge. Solutions for this problem exist but will not be discussed here.

Representing a graph as a matrix gives us the ability to apply common matrix operations, which can in turn be useful again to calculate certain properties of graphs. For example the breadth-first search algorithm can be solved using this approach as discussed in the next chapter. One core operation necessary to perform many of these algorithms is Matrix Vector multiplication, and since most graphs are not fully adjacent we can generalize into Sparse Matrix Vector multiplication (SpMV).

4.1 Vector multiplication on an adjecency matrix

So what information can we calculate using Matrix Vector multiplication? Let us first discuss an example. We start by defining a small unweighted example graph G as follows:



Our graph has three vertices and three edges connecting all the vertices together. This gives us the following adjacency matrix according to the above definition:

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Here each column represents all the outgoing edges of a single vertex, and each row represents all incoming edges of a vertex. Let us apply a vector with a single non-zero value over the transform of the matrix:

$$A^T v = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}^T \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Our multiplication has found which rows have non-zero values in our first column. This column signifies all the outgoing edges of our first vertex, thus we have in actuality found out all vertices connected to the first vertex. For a single vertex that is not an impressive feat. However, we can apply this calculation on multiple columns at the same time thereby giving us a list of all vertices adjacent to a sub graph. We use this building stone for our breadth-first search implementation. The full algorithm and proof will be discussed in the next chapter.

4.2 Data structures for a sparse-matrix

As shown we can apply a matrix vector multiplication over the adjacency matrix of a graph to get information about the outgoing edges of the vertices we want to query. However, using a matrix for this application does come at a cost: the memory usage. A graph with m edges needs $\mathcal{O}(m)$ memory to store these edges when stored as an adjacency list. However, when stored as an adjacency matrix we need $\mathcal{O}(m^2)$ elements. An adjacency matrix gives us the advantage that we can lookup any edge given the source and sink of the edge, but does come at a significant higher memory usage.

Since the Graphcore IPU has a severely limited amount of memory, we cannot use this approach. Instead we can use the sparsity of the matrix to our advantage. There are several different data structures specifically meant for sparse matrices:

- Dictionary of keys (DOK): the row and column are used as keys in a dictionary. This is an easy structure to use but more difficult to implement and has longer querying times compared to direct array structures.
- Coordinate list: the matrix is stored as a list of row, column and value triples. Potentially ordered in a specific way meant for processing. This makes it a compact data structure, but does not have the capability to query specific row, column locations without searching through the structure. This comes at a memory cost of $\mathcal{O}(nz)$ (nz is the amount of non-zero values in our sparse-matrix).
- Compressed sparse row (CSR) or compressed sparse column (CSC): compresses the matrix in three lists: a list of values, a list of columns (or rows) belonging to those values and an indexing list pointing to the start and end of every row (or column) in the two other lists. This makes it possible to find either all values in a row or column depending on the type used. This comes at a memory cost of $\mathcal{O}(nz + n)$ or $\mathcal{O}(nz + m)$.

For our uses it turned out that the CSR and CSC structures were a good trade-off between memory usage and functionality.

4.3 Single-threaded sparse-matrix vector multiplication

Before we define a multi-threaded algorithm for SpMV we first want to find a single-threaded solution. Every row is defined as the sum of each nz value multiplicated by the input vector. A single threaded algorithm is thus achieved by iterating through each nz value and keeping track of the sum for each row.

4.4 A parallel algorithm

When designing a parallel algorithms we need to find a method to best split up our problem into multiple sub-problems. If we can divide our problem we have a possibility for concurrent execution, giving us a speedup.

One advantage of matrix vector multiplication is that we can easily split it up. One method is to partition the matrix into multiple submatrices, one such method is using 2d-partitioning.

4.4.1 2d-partitioning

There are multiple ways to divide up our matrix into submatrices. To make our algorithm as efficient as possible we need to find a division in which each problem is roughly of the same size. In the case of a sparse matrix that size is decided by the amount of non-zero values in our submatrix.

We have chosen to create an equal division of the matrix, where each submatrix roughly contains the same amount of rows and columns. To ensure that we divide up the work evenly we perform a (random) permutation on both the matrix and vector. Our result is than permuted back to give us our final result.

Since we have 1472 tiles available we can fit exactly $\lfloor\sqrt{1472}\rfloor = 38$ blocks in a row or column giving us a total of $38 * 38 = 1444$ blocks.

4.5 IPU Algorithm

The full code that is executed can be found in listing A.1. In general the main ingredient is the following sequence:

```
1 Sequence{Repeat(amount_of_loops,  Sequence{spmv, reduce})};
```

4.6 Analysis

4.6.1 Flamegraph

To gain an insight into how efficiently we are using the IPU a flamegraph can be used. A flamegraph shows us how each single tile of the IPU is being used over time. We differentiate over three different states: the tile is executing a program (indicated by the

Figure 4.1: A flame graph of the SpMV algorithm while executing `bfly.mtx` (1-round). Each line represents a single tile on the IPU

color red), the tile is awaiting a synchronization (indicated by the color yellow) or the tile can be exchanging data with other tiles (indicated by the color blue).

An efficient algorithm uses as much of the compute time available to it, and minimizes the exchange and synchronization time.

One caveat, a high execution rate means that the tiles are being used efficiently, it does not indicate that the program that is executed on each tile is efficient.

Figure 4.1 shows the flame graph of a single round of our SpMV algorithm. A single round consists of a single matrix vector multiplication. The steps shown can be summarized as follows:

1. We exchange the previous round's result to be used as a starting point for this round
2. We perform the multiplicative step on each block
3. We reduce the results with addition for each row in each block

We can note that our our usage of the IPU is high, with only a small portion of time used to synchronize the different tiles. We also note a yellow gap neer the bottom of the flame graph. This is caused by the fact that we cannot assign a block to each tile of the IPU, leaving a few tiles empty during the second step of our program.

4.7 Benchmarking

Test instances

To be able to benchmark our algorithm we have chosen a set of matrices from [6]. Most of these matrices have previously been used for [4] to provide us a basis for comparison. All of our matrices are created from graph instances, this is not necessary for this experiment but is for breadth-first search and Prim's algorithm.

We use the following matrices. A complete overview and some properties of the contained graph can be found in table **??**.

- `BFly`: a butterfly graph
- `kron_g500-logn(n)` are Kronecker graphs from the 10th DIMACS implementation challenge.
- `G43` is a sparse uniformly random matrix.
- coAuthorsDBLP and coPapersDBLP are networks of academic collaboration.
- Journals represents shared readership across different research journals .
- Delaunay (n) are planar graphs from the 10th DIMACS implementation challenge.
- loc-Gowalla friendship data from a social network based on location.

The test instances all fit in memory for the SpMV and BFS experiment. For Prim's algorithm not all graphs could be performed due to memory constraints.

4.7.1 Setup

Every test instance is performed ten times, the average result of those ten execution is taken. If any outliers are present the experiment is performed again. An outlier is defined as having a distance $1.5 * IQR$ from either $Q1$ or $Q3$.

Our project is compiled with the Poplar SDK 3.0 and ran on a single GC2 IPU.

The amount of cycles taken to perform the compute set is recorded by the IPU and converted to μs by using $1.35Ghz$ as the clock rate. We also record the graph compilation time, and the memory copy time, these times are recorded on the host.

Name	*Size (n)*	*Non-zero values (2 ∗ m)*	*Min. degree*	*Avg. degree*
Bfly (12)	49k	197k	4	4
coAuthorsDBLP	300k	978k	1	6
CoPapersDBLB	540k	15m	1	56
delaunay_n10	1k	3k	3	5
delaunay_n11	2k	6k	3	5
delaunay_n12	4k	12k	3	5
delaunay_n13	8k	25k	3	5
delaunay_n14	16k	49k	3	5
delaunay_n15	33k	98k	3	5
delaunay_n16	66k	197k	3	5
delaunay_n17	131k	393k	3	5
delaunay_n18	262k	786k	3	5
delaunay_n19	524k	1.5m	3	5
delaunay_n20	1048k	3m	3	5
G43	1k	10k	7	19
Journals	124	6k	18	96
kron_g500-logn16	66k	2m	0	74
kron_g500-logn17	118k	5m	0	78
kron_g500-logn18	236k	11m	0	80
kron_g500-logn19	432k	22m	0	83
loc-Gowalla	197k	950k	1	9
ship_003	122k	4m	17	65

Table 4.1: Overview of the used matrices during testing, and some of their properties. All matrices are made from undirected graphs [6].

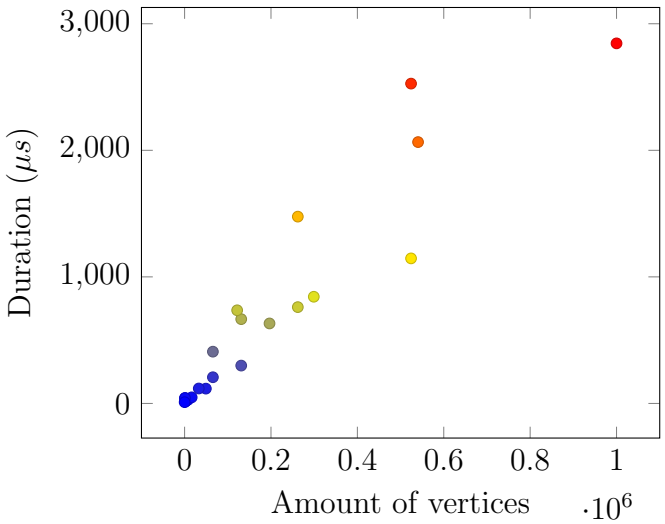


Figure 4.2: Plot of execution duration vs the amount of vertices in the matrix (number of rows) for the SpMV experiment.

Matrix	*Graph compilation (s)*	*Execution (µs)*	*Memory copy (µs)*
BFly	18.91	12	7949
coAuthorsDBLP	19.21	84	50532
coPapersDBLP	19.47	207	517169
delaunay_n10	10.63	4	696
delaunay_n11	16.32	2	958
delaunay_n12	17.52	2	1261
delaunay_n13	16.67	3	2074
delaunay_n14	18.32	5	3323
delaunay_n15	19.03	12	5882
delaunay_n16	20.32	21	12162
delaunay_n17	21.01	30	22551
delaunay_n18	17.81	76	46255
delaunay_n19	18.36	115	148069
delaunay_n20	20.59	285	289355
G43	13.82	4	895
Journals	14.18	1	876
kron_g500-logn16	19.38	41	37403
kron_g500-logn17	20.96	67	78131
kron_g500-logn18	18.11	148	298994
kron_g500-logn19	20.98	253	659572
loc-Gowalla	19.15	63	37190
ship_003	20.81	74	64888

Table 4.2: Execution results of our matrices for the Spare-matrix vector multiplication experiment.

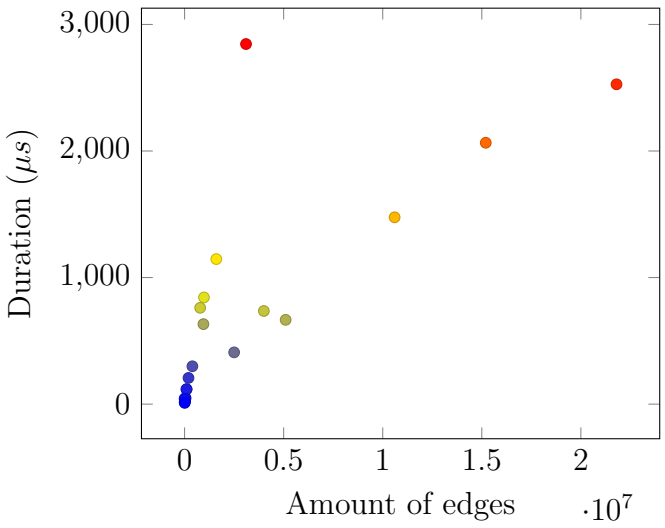


Figure 4.3: Plot of execution duration versus the amount of edges in the matrix for the SpMV experiment.

4.7.2 Results

Our results can be found in table 4.2, figure 4.2 and figure 4.3.

4.8 Discussion

It is difficult to compare our results to any other meaning full data. Results for GPU implementations are severely outdated.

Nevertheless we have shown that it is possible to implement SpMV on the IPU, and we were able to perform calculations on matrices with up to 22 million non-zero values.

We have alsp shown that our implementation shows a linear curve meaning that we have not introduced any extra complexity to the algorithm.

4.9 Possible improvements

We have shown that our approach gives correct results. And that due to the linear nature of instruction time for the IPU, any speedup of the processors clock speed would automatically result in an almost linear speedup of our algorithm.

However, there are definitely other possible improvements to be made to our implementation to achieve higher speeds with the current set-up.

One such improvement would be to optimize the compiled byte code by hand. Currently the IPU code is written in C++ and compiled using the `popc` compiler. Writing the core of the algorithm in assembly instead could improve our results due to limitations of the compiler. But this would come at the cost of readability, and requires intricate knowledge of the IPU instruction set.

Chapter 5

Breadth-first search

A standard problem within the space of graphs is that of pathfinding, often given as the following problem: find the shortest path between two given vertices.

Many algorithms exist to solve this problem, such as Dijkstra and A*. For unweighted graphs the Breadth-first search algorithms finds a solution. In Breadth-first search all neighbours are slowly iterated through with the closest neighbours

5.1 Adapting our Sparse-Matrix Vector code

We have created two implementations The full code that is executed can be found in listing A.4.

5.2 Benchmarking

We have followed the same steps as described in the SpMV chapter.

5.3 Flamegraph

A flamegraph of the full execution of the BFly matrix can be found in figure 5.1, and a single iteration can be found in figure 5.2.

We notice that compared to our SpMV algorithm there is a single big difference in the collection of results. This collection is mostly single threaded, and difficult to parallelize since the extra costs of memory operations are too high.

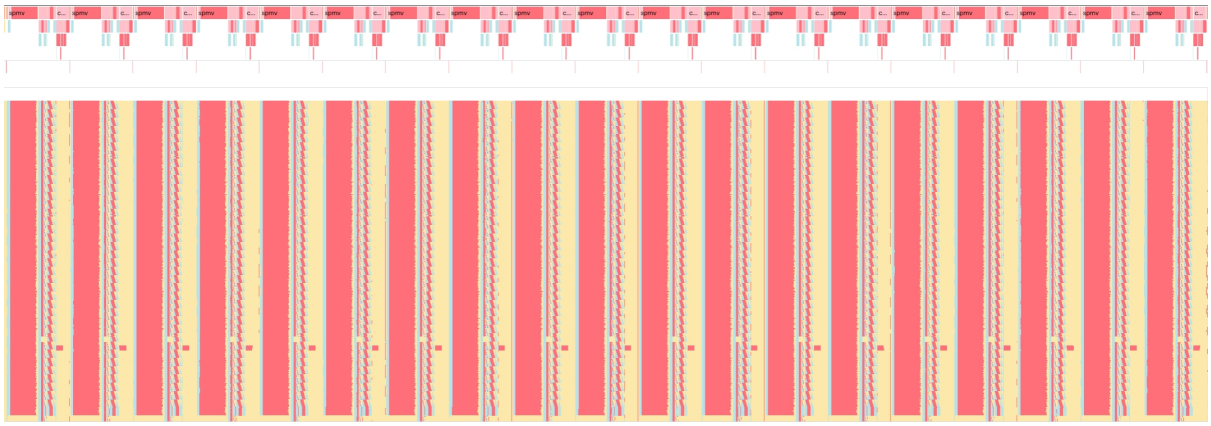


Figure 5.1: A flamegraph of the full execution of the BFly matrix.

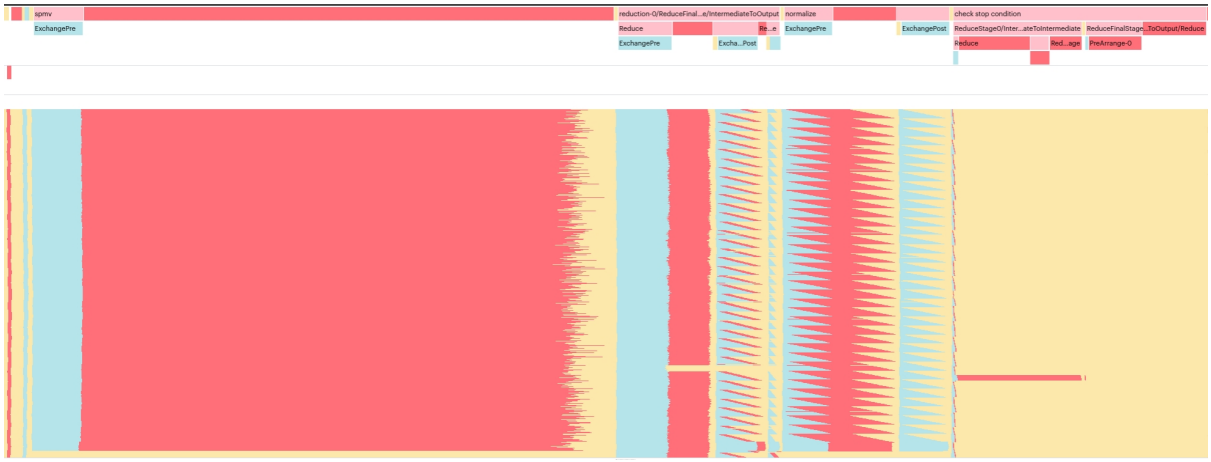


Figure 5.2: A flamegraph of a single step of the execution of the BFly matrix.

Matrix	*Graph compilation (s)*	*Execution (μs)*	*Memory copy (μs)*
bfly	12.63	358	8028
coAuthorsDBLP	8.65	1,664	41570
coPapersDBLP	9.78	3,753	307106
delaunay_n10	11.16	160	2057
delaunay_n11	11.32	154	3444
delaunay_n12	11.47	224	2825
delaunay_n13	11.33	377	4743
delaunay_n14	11.94	623	4594
delaunay_n15	11.97	1,510	12915
delaunay_n16	13.37	3,118	10950
delaunay_n17	11.85	6,349	21501
delaunay_n18	10.06	18,236	34757
delaunay_n19	9.06	43,639	191273
delaunay_n20	11.47	140,669	249283
G43	10.57	45	2106
Journals	10.65	11	1980
kron_g500-logn16	14.29	307	23037
kron_g500-logn17	12.37	503	44923
kron_g500-logn18	7.76	1,032	152977
kron_g500-logn19	10.48	349	375009
loc-Gowalla	8.86	807	29970
ship_003	10.49	5,613	43986

Table 5.1: Execution results of our matrices for the Breadth-first-search.

5.4 Results

Results can be found in table 5.1 and figure 5.3.

5.5 Possible improvements

Our implementation of the breath-first search algorithm lacks many optimizations to further increase the speed. These optimizations have partially not been made due to constraints in the architecture which were difficult to overcome, for instance memory problems in some cases where multiple workers on the same data were used.

Some possible improvements are:

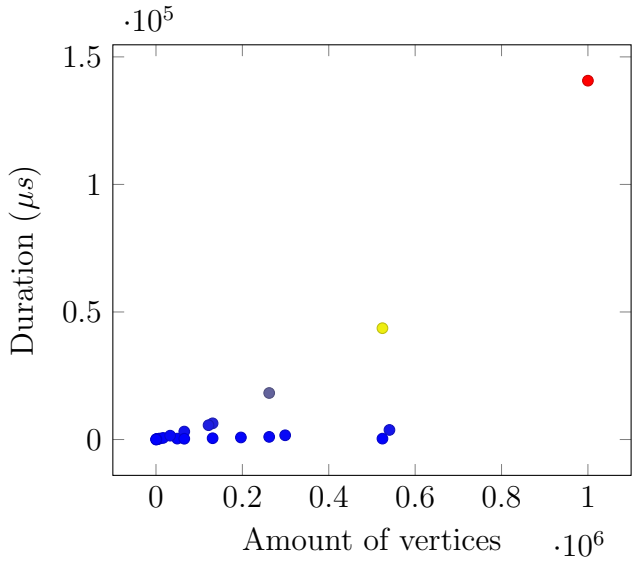


Figure 5.3: Plot of execution duration vs the amount of vertices in the matrix (number of rows) for the BFS experiment.

- Usage of the inbuilt boolean data type. This would improve memory usage. Currently, every row in the matrix and thus vertex in our graph has a 32-bit float allocated for keeping track of the existence in our current iteration, this could be replaced with a boolean
- One could go even further and see if it is worth to implement own boolean semantics. Currently the IPU will allocate a full byte for every boolean, this could be reduced to a single bit.
- Improvements to the finish check step. Currently, this is a single threaded operation, and while this step is not fully parallelizable, it is possible to subdivide further. The biggest challenge there would be to weigh the overhead of a multi step process versus the improved IPU usage. Another possibility is to perform some iterations without this step since the benefit of skipping this step could be bigger than the lost time due to unnecessary steps.
- Writing some of the steps in assembly. This could allow for greater optimizations than the compiler currently optimizes for.

Chapter 6

Prim's algorithm

Our breath-first search algorithm was heavily based upon our earlier implementation of SpMV. However, not all graph algorithms have a necessity for such operations. One such example is Prim's algorithm which we are discussing in this chapter. Prim's algorithm is an algorithm for finding the minimum spanning tree of a weighted undirected graph. A minimum spanning tree is sub-graph of edges of our input graph which form a tree of minimal weight such that all vertices of our graph are connected.

6.1 Classical Prim's

6.2 Algebraic algorithm

An algabreic version of Prim's algorithm can be found in algorithm 1.

6.3 Example

We now give a small example of how the algorithm functions. Let's us take the following graph as our input:

Data: A: a graph in matrix notation
$\mathbf{s} = 0$;
$weight = 0$;
$\mathbf{s}(1) = \inf$;
$\mathbf{d} = \mathbf{A}(1, :)$;
while $\boldsymbol{s} \neq \inf$ **do**
    $u = \text{argmin}\{\mathbf{s} + \mathbf{d}\}$;
    $\mathbf{s}(u) = \inf$;
    $weight = weight + \mathbf{d}(u)$;
    $\mathbf{d} = \mathbf{d}.\min\mathbf{A}(u, :)$;
end

Algorithm 1: An algebraic version of Prim's algorithm

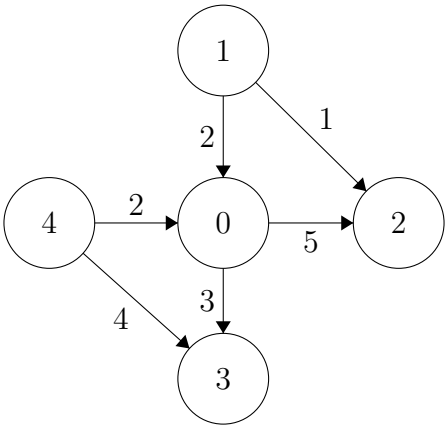


Figure 6.1: Example graph

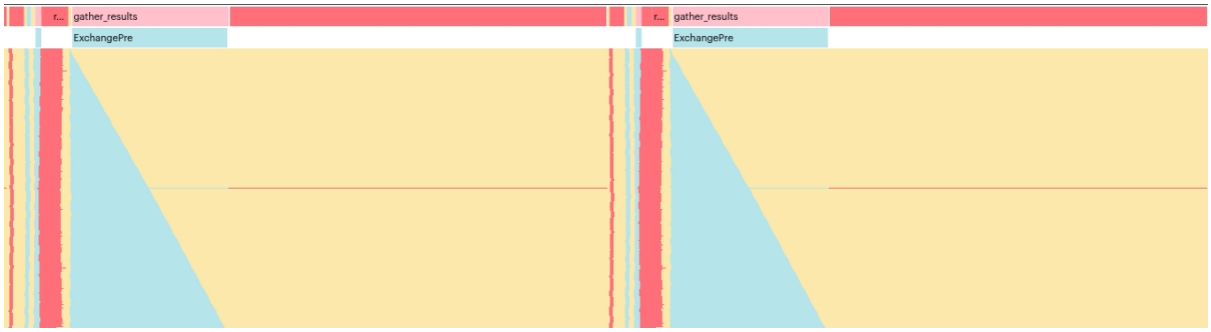


Figure 6.2: A flame graph showing two cycles of the Prim's algorithm while executing `bfly.mtx`

This graph would be converted to our matrix notation as follows (zero values are not shown):

$$A = \begin{bmatrix} & 2 & 5 & 3 & 2 \\ 2 & & 1 & & \\ 5 & 1 & & & \\ 3 & & & & 4 \\ 2 & & & 4 & \end{bmatrix}$$

Each column and row of our matrix is associated with the outbound and inbound edges of a single vertex in our input graph. We note that our matrix is mirrored along the diagonal since our graph is undirected.

6.3.1 Implementation on the IPU

The full code that is executed can be found in listing A.7.

6.4 Results

6.4.1 Flamegraph

Our flamegraph can be found in figure 6.2. We notice that a large amount of time is spend executing the collection of result. This is single threaded and due to hardware limitation posed a constraint which could not be fixed.

Matrix	*Graph compilation (s)*	*Execution (µs)*	*Memory copy (µs)*
delaunay_n10	11.22	9,229	6756
delaunay_n11	6.43	18,458	25741
delaunay_n12	7.6	36,918	22207
delaunay_n13	7.45	73,880	37858
delaunay_n14	8.65	148,151	211273
delaunay_n15	10.92	297,946	315474
G43	11.57	9,011	13392
Journals	9.35	1,120	3978

Table 6.1: Execution results of our matrices for the Prims experiment.

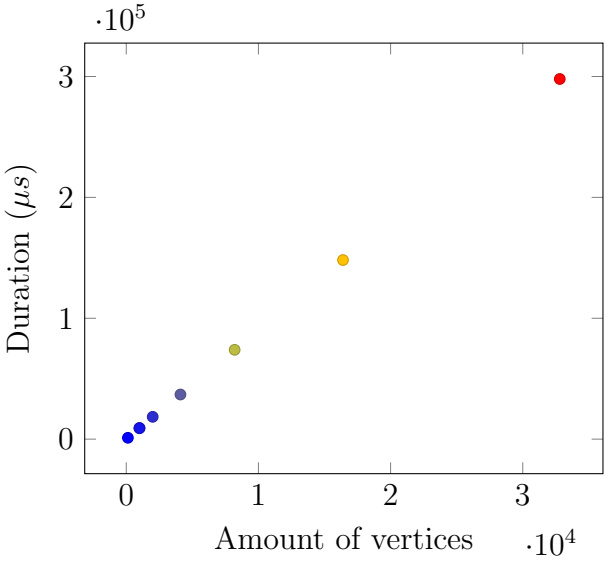


Figure 6.3: Plot of execution duration vs the amount of vertices in the matrix (number of rows) for the prims experiment.

Results

Results can be found in the table 6.1 and figure 6.3.

6.4.2 Possible improvements

A lot more work is required to find an efficient Prims implementation on the IPU. Main work should be focused on figuring out problems which are currently cause by the long time it takes memory to move, and the single threaded-ness of the collection of the results.

Chapter 7

Conclusion

In this thesis we have introduced, explained and used the Graphcore IPU, a first-of-its-kind multiple instruction, multiple data architecture. We successfully implemented three different linear-algebra based algorithms, and tested them.

One thing we noticed throughout the problems we have have mentioned is a severe lack of necessity of one of the main features the Graphcore IPU provides: the ability to run different programs on each tile of the IPU. While it is advantageous to not have to stick to a specific control-flow mechanism compared to a Warp on a GPU, this does not make a huge difference in computation time.

Compared to the GPU there is one other advantage to the architecture: the local memory allows much faster lookup times which can be interesting for problems where memory is accessed in an unpredictable pattern.

7.1 Working with the Graphcore IPU

The IPU and its libraries have many shortcomings which we discussed in section 3.7.7, many of which were at one point encountered during the implementation phase of this thesis. Besides these we often encountered bugs in the provided tooling, like a lack of implementation of some listed features. These pitfalls can make programming for the IPU cumbersome and difficult. Especially the lack of sufficient debugging tooling means that rare bugs are not easily found and solved. During our implementation phase this

lead to multiple times were the decision was take to scrap the current implementation in favour of a simpler alternative.

For instance for our breath-first search implementation this lead to a less efficient implementation which uses more memory per vertex and edge in the input matrix.

As mentioned there is a severe lack of real-world usage of the machine, this increases the difficulty of developing for the IPU since there are very few clear examples. Much of the functionality of The Poplar SDK and the associated libraries remains unused outside of Graphcore's own and other non-public projects.

Finally there also a lack of clear documentation. While functionality is documented, there is currently no cohesive documentation on how different functions can be combined. This has created some situations were the Poplar SDKs source code had to be used to correctly use a feature. And while the source code of the Poplibs functionality of the SDK is freely available, this is not the case of all outwardly facing functionality.

These facts combined make the learning curve for the hardware steep. Developing for the machine efficiently and correctly outside of its main focus domains requires an intricate knowledge about these and other constraints, especially of one does not have a need of a MIMD architecture. Instead for those kind of problems it would be much easier to develop programs for the GPU.

Chapter 8

Future work

We've laid out a groundwork and library upon which other graph algorithms can be implemented. Our work shows a clear foundation of how the Poplar SDK and Graphcore IPU function and contains examples of a range of different techniques which can be used to program on the IPU.

One area which is interesting for further exploration is that of dynamically sized matrices. In our implementation the size of a matrix is fixed for correct linking of tiles to tensors. Theoretically, a computation graph containing a dynamically sized matrix can be used using the same groundwork, but the calculations and implementation for this require more attention. This should not generate more work during runtime, but requires more intricate memory management.

Next to linear algebra based algorithms other classes of algorithms can be explored. One such class could be parallelized divide and conquer algorithms. These class of algorithms are in theory highly parallelizable but can be difficult to implement due to thread communication constraints. Furthermore, they require optimized checks to see when new threads should be spawned.

Overall there are many different directions future work on the Graphcore IPU can be taken.

List of Acronyms and Abbreviations

CPU Central Processing Unit.

GPU Graphics Processing Unit.

IPU Intelligence Processing Unit.

MIMD Multiple instruction, multiple data.

MISD Multiple instruction, single data.

SIMD Single instruction, multiple data.

SISD Single instruction, single data.

Bibliography

[1] ACM, editor. *Proceedings of the 1964 fall joint computer conference, part II: very high speed computer systems, AFIPS 1964 (Fall, part II), San Francisco, California, USA, October 27-29, 1964*, 1964. ACM. ISBN 978-1-4503-7888-8. doi: 10.1145/1464039.
URL: https://doi.org/10.1145/1464039.

[2] *AMD EPYC™ 9754*. AMD, 2023.
URL: https://www.amd.com/en/product/13371.

[3] Luk Burchard, Xing Cai, and Johannes Langguth. ipug for multiple graphcore ipus: Optimizing performance and scalability of parallel breadth-first search. In *28th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2021, Bengaluru, India, December 17-20, 2021*, pages 162–171. IEEE, 2021. doi: 10.1109/HiPC53243.2021.00030.
URL: https://doi.org/10.1109/HiPC53243.2021.00030.

[4] Luk Burchard, Johannes Moe, Daniel Thilo Schroeder, Konstantin Pogorelov, and Johannes Langguth. ipug: Accelerating breadth-first graph traversals using manycore graphcore ipus. In Bradford L. Chamberlain, Ana Lucia Varbanescu, Hatem Ltaief, and Piotr Luszczek, editors, *High Performance Computing - 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 - July 2, 2021, Proceedings*, volume 12728 of *Lecture Notes in Computer Science*, pages 291–309. Springer, 2021. doi: 10.1007/978-3-030-78713-4_16.
URL: https://doi.org/10.1007/978-3-030-78713-4_16.

[5] OpenStreetMap contributors. Openstreetmap stats. Technical report, OSM Foundation, 2023.
URL: https://planet.openstreetmap.org/statistics/data_stats.html.

[6] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011. ISSN 0098-3500. doi:

10.1145/2049662.2049663.
URL: https://doi.org/10.1145/2049662.2049663.

[7] Graphcore. *Graphcore Tile Vertex ISA*. Graphcore Ltd, 2022.
URL: https://docs.graphcore.ai/projects/isa/en/latest/_static/Tile-Vertex-ISA_1.2.3.pdf.

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

[9] Simon Knowles. Graphcore. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–25, 2021. doi: 10.1109/HCS52781.2021.9567075.

[10] Graphcore Ltd. *IPU-POD64 REFERENCE DESIGN DATASHEET*, 2020.
URL: https://docs.graphcore.ai/projects/ipu-pod64-datasheet/en/2.1.0/.

[11] *CUDA C++ Programming Guide*. NVIDIA, 2023.
URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[12] Philip E. Ross. Why cpu frequency stalled. *IEEE Spectrum*, 45(4):72–72, 2008. doi: 10.1109/MSPEC.2008.4476447.

[13] John Shalf. The future of computing beyond moore's law. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 378(2166): 20190061, 2020. doi: 10.1098/rsta.2019.0061.
URL: https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2019.0061.

[14] John Shalf. The future of computing beyond moore's law. *Philosophical Transactions of the Royal Society A*, 378(2166):20190061, 2020.

[15] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David Kaeli. Summarizing cpu and gpu design trends with product data, 2020.

Appendix A

Code written for the Graphcore IPU

This section contains the unnabridged source code for the IPU and the host machine for the different experiments. An archive of the full source code and instructions on how it can be used can be found at `https://github.com/yoeori/thesis`.

Listing A.1: SpMV experiment host execution

```
#include <iostream>
#include <cstdlib>
#include <algorithm>
#include <cmath>
#include <chrono>

#include <poplar/Engine.hpp>
#include <poputil/TileMapping.hpp>
#include <poplar/DeviceManager.hpp>
#include <poplar/Program.hpp>
#include <poplar/CycleCount.hpp>
#include <popops/ElementWise.hpp>
#include <popops/codelets.hpp>
#include <popops/Reduce.hpp>

#include "../matrix.hpp"
#include "../config.cpp"
#include "../ipu.cpp"
#include "../report.cpp"

using ::poplar::Device;
using ::poplar::Engine;
using ::poplar::Graph;
using ::poplar::Tensor;
using ::poplar::OptionFlags;
using ::poplar::SyncType;

using ::poplar::FLOAT;
using ::poplar::INT;

using ::poplar::program::Copy;
using ::poplar::program::Execute;
using ::poplar::program::Program;
using ::poplar::program::Repeat;
using ::poplar::program::Sequence;

using ::popops::SingleReduceOp;
using ::popops::reduceMany;

```

```
40 namespace exp_spmv
41 {
42
43     // Helper functions for experiment
44     namespace
45     {
46         template <typename T, typename = typename
               ↪ std::enable_if<std::is_arithmetic<T>::value, T>::type>
47         struct IPUMatrix
48         {
49         public:
50             IPUMatrix(vector<int> offsets, vector<T> matrix,
                   ↪ vector<int> idx, vector<int> row_idx, int blocks,
                   ↪ int block_height, int m, int n) : offsets(offsets),
                   ↪ matrix(matrix), idx(idx), row_idx(row_idx),
                   ↪ blocks(blocks), block_height(block_height), m(m),
                   ↪ n(n) {}
51
52             vector<int> offsets;
53
54             // actuall data
55             vector<T> matrix;
56             vector<int> idx; // better indexing type? size_t?
57             vector<int> row_idx;
58
59             // matrix data
60             unsigned int blocks;
61             unsigned int block_height;
62             unsigned int m;
63             unsigned int n;
64         };
65
66         template <typename T, typename = typename
               ↪ std::enable_if<std::is_arithmetic<T>::value, T>::type>
67         auto prepare_data(matrix::Matrix<T> matrix, const int
               ↪ num_tiles)
68         {
69             // assumptions at this point in the code: matrix is
                   ↪ shuffled (values are normally divided)
70             // TODO: prepareData currently takes O(n*m), can be done
                   ↪ in O(nz) for SparseMatrix type
71
72             // First we calculate how many blocks we have available.
                   ↪ We need x tiles for summation and x^2 blocks for the
                   ↪ SpMV
73             // In general this _should_ make it possible to execute
                   ↪ SpMV on the same matrix twice with differents
                   ↪ vectors.
74             // const auto blocks = (int)std::floor((-1.0 + std::sqrt(1
                   ↪ + 4 * num_tiles)) / 2.0); // For a standard IPU 37*37
75             const auto blocks = (int) std::floor(std::sqrt(num_tiles));
76             const auto block_size_col = std::max(matrix.cols() /
                   ↪ blocks + (matrix.cols() % blocks != 0), 1);
77             const auto block_size_row = std::max(matrix.rows() /
                   ↪ blocks + (matrix.rows() % blocks != 0), 1);
78
79             vector<T> ipu_matrix(matrix.nonzeroes());
80             vector<int> idx(matrix.nonzeroes());
81
82             // Could be more compact (the last row might need less
                   ↪ space), but this complicates location calculations
                   ↪ _a lot_
83             vector<int> row_idx(blocks * blocks * (block_size_row +
                   ↪ 1));
84
85             // Next we perform summation over the matrix to find exact
                   ↪ length for each block
86             // TODO: we should/could log normality of sparse matrix
```

```
87              // In general the row_idx length is the same for each
                   ↪ block, with the expection of the last row of blocks.
                   ↪ (being ceil(matrix.m / blocks))
88
89              // This will give the offsets for matrix and idx
90              vector<int> offsets(blocks * blocks + 1);
91              offsets[0] = 0;
92
93              for (auto y = 0; y < blocks; y++)
94              {
95                  for (auto x = 0; x < blocks; x++)
96                  {
97                      offsets[y * blocks + x + 1] = offsets[y * blocks +
                           ↪ x];
98
99                      // Search block for non-zero
100                     for (auto mi = block_size_row * y; mi <
                           ↪ std::min(block_size_row * (y + 1),
                           ↪ matrix.rows()); mi++)
101                     {
102                         // Record row offsets
103                         row_idx[(y * blocks + x) * (block_size_row +
                               ↪ 1) + mi - block_size_row * y] =
                               ↪ offsets[y * blocks + x + 1] - offsets[y
                               ↪ * blocks + x];
104
105                         for (auto mj = block_size_col * x; mj <
                               ↪ std::min(block_size_col * (x + 1),
                               ↪ matrix.cols()); mj++)
106                         {
107                             if (matrix.get(mi, mj) != 0)
108                             {
109                                 ipu_matrix[offsets[y * blocks + x +
                                       ↪ 1]] = matrix.get(mj, mi);
110                                 idx[offsets[y * blocks + x + 1]] = mj
                                       ↪ - block_size_col * x;
111
112                                 offsets[y * blocks + x + 1] += 1;
113                             }
114                         }
115                     }
116
117                     row_idx[(y * blocks + x) * (block_size_row + 1) +
                           ↪ std::min(block_size_row * (y + 1),
                           ↪ matrix.rows()) - block_size_row * y] =
                           ↪ offsets[y * blocks + x + 1] - offsets[y *
                           ↪ blocks + x];
118                 }
119             }
120
121             // Final value should be nz (sum of every block)
122             assert(offsets[offsets.size() - 1] == matrix.nonzeroes());
123
124             return IPUMatrix(offsets, ipu_matrix, idx, row_idx,
                   ↪ blocks, block_size_row, matrix.rows(),
                   ↪ matrix.cols());
125         }
126
127         template <typename T, typename = typename
                ↪ std::enable_if<std::is_arithmetic<T>::value, T>::type>
128         auto prepare_data(matrix::SparseMatrix<T> matrix, const int
                ↪ num_tiles)
129         {
130             const auto blocks = (int) std::floor(std::sqrt(num_tiles));
131             const auto block_size_col = std::max(matrix.cols() /
                   ↪ blocks + (matrix.cols() % blocks != 0), 1);
132             const auto block_size_row = std::max(matrix.rows() /
                   ↪ blocks + (matrix.rows() % blocks != 0), 1);
```

```
133
134             // This will give the offsets for matrix and idx
135             vector<int> offsets(blocks * blocks + 1, 0);
136             vector<int> row_idx(blocks * blocks * (block_size_row +
                    ↪ 1), 0);
137
138             // We go through each value in the SpM and update offsets
                    ↪ and row_idx
139             for (auto o = 0; o < matrix.nonzeroes(); o++)
140             {
141                 auto [i, j, v] = matrix.get(o);
142                 (void)v;
143
144                 auto x = j / block_size_col;
145                 auto y = i / block_size_row;
146
147                 offsets[y * blocks + x + 1]++;
148                 row_idx[(y * blocks + x) * (block_size_row + 1) + (i -
                        ↪ (block_size_row * y)) + 1]++;
149             }
150
151             // Stride offsets and row_idx
152             for (size_t i = 2; i < offsets.size(); i++)
153             {
154                 offsets[i] += offsets[i - 1];
155             }
156
157             for (auto block = 0; block < blocks * blocks; block++)
158             {
159                 for (auto i = 0; i < block_size_row; i++)
160                 {
161                     row_idx[block * (block_size_row + 1) + i + 1] +=
                            ↪ row_idx[block * (block_size_row + 1) + i];
162                 }
163             }
164
165             assert(offsets[offsets.size() - 1] == matrix.nonzeroes());
166
167             vector<int> cursor(row_idx); // Cursor inside a block
                    ↪ between rows
168
169             vector<T> ipu_matrix(matrix.nonzeroes());
170             vector<int> idx(matrix.nonzeroes());
171
172             for (auto o = 0; o < matrix.nonzeroes(); o++)
173             {
174                 auto [i, j, v] = matrix.get(o);
175
176                 auto x = j / block_size_col;
177                 auto y = i / block_size_row;
178
179                 size_t value_offset = offsets[y * blocks + x] +
                        ↪ cursor[(y * blocks + x) * (block_size_row + 1) +
                        ↪ (i - (block_size_row * y))];
180
181                 ipu_matrix[value_offset] = v;
182                 idx[value_offset] = j - (block_size_col * x);
183
184                 // Update cursor
185                 cursor[(y * blocks + x) * (block_size_row + 1) + (i -
                        ↪ (block_size_row * y))]++;
186             }
187
188             return IPUMatrix(offsets, ipu_matrix, idx, row_idx,
                    ↪ blocks, block_size_row, matrix.rows(),
                    ↪ matrix.cols());
189         }
190
```

```
191         void build_compute_graph(Graph &graph, map<string, Tensor>
                ↪ &tensors, map<string, Program> &programs, const int
                ↪ num_tiles, IPUMatrix<float> &ipu_matrix, const int loops)
192         {
193
194             // Static Matrix data
195             tensors["matrix"] = graph.addVariable(FLOAT,
                    ↪ {ipu_matrix.matrix.size()}, "matrix");
196             tensors["idx"] = graph.addVariable(INT,
                    ↪ {ipu_matrix.idx.size()}, "idx");
197             tensors["row_idx"] = graph.addVariable(INT,
                    ↪ {ipu_matrix.blocks, ipu_matrix.blocks,
                    ↪ ipu_matrix.block_height + 1}, "row_idx");
198
199             // Input/Output vector
200             tensors["vector"] = graph.addVariable(FLOAT, {(unsigned
                    ↪ int)ipu_matrix.n}, "vector");
201             tensors["res"] = graph.addVariable(FLOAT,
                    ↪ {ipu_matrix.blocks, ipu_matrix.blocks,
                    ↪ ipu_matrix.block_height}, "result");
202
203             // We build the compute set for the MatrixBlock codelet
204             auto spmv_cs = graph.addComputeSet("spmv");
205
206             for (unsigned int y = 0; y < ipu_matrix.blocks; y++)
207             {
208                 for (unsigned int x = 0; x < ipu_matrix.blocks; x++)
209                 {
210                     auto block_id = y * ipu_matrix.blocks + x;
211                     auto v = graph.addVertex(spmv_cs, "MatrixBlock", {
212                         {"matrix", tensors["matrix"].slice(
                                ↪ ipu_matrix.offsets[block_id],
                                ↪ ipu_matrix.offsets[block_id + 1])},
213                         {"idx", tensors["idx"].slice(
                                ↪ ipu_matrix.offsets[block_id],
                                ↪ ipu_matrix.offsets[block_id + 1])},
214                         {"row_idx", tensors["row_idx"][y][x]},
215                         {"vec", tensors["vector"].slice(
                                ↪ std::min(ipu_matrix.m, x *
                                ↪ ipu_matrix.block_height),
                                ↪ std::min(ipu_matrix.m, (x + 1) *
                                ↪ ipu_matrix.block_height))},
216                         {"res", tensors["res"][y][x]}
217                     });
218
219                     // TODO need to be calculated;
220                     graph.setPerfEstimate(v, 100);
221                     graph.setTileMapping(v, block_id);
222
223                     graph.setTileMapping(tensors["matrix"].slice(
                            ↪ ipu_matrix.offsets[block_id],
                            ↪ ipu_matrix.offsets[block_id + 1]), block_id);
224                     graph.setTileMapping(tensors["idx"].slice(
                            ↪ ipu_matrix.offsets[block_id],
                            ↪ ipu_matrix.offsets[block_id + 1]), block_id);
225                     graph.setTileMapping(tensors["row_idx"][y][x],
                            ↪ block_id);
226                     graph.setTileMapping(tensors["vector"].slice(
                            ↪ std::min(ipu_matrix.m, x *
                            ↪ ipu_matrix.block_height),
                            ↪ std::min(ipu_matrix.m, (x + 1) *
                            ↪ ipu_matrix.block_height)), block_id);
227                     graph.setTileMapping(tensors["res"][y][x],
                            ↪ block_id);
228                 }
229             }
230
231             auto program_spmv = Execute(spmv_cs);
```

```
232
233             // We build the compute set for addition
234             auto reducer_cs = graph.addComputeSet("reduce");
235             poplar::program::Program program_reduce;
236
237             if (!Config::get().own_reducer) {
238
239                 auto res_vector_shuffled =
                        ↪ tensors["res"].dimShuffle({0, 2, 1});
240
241                 vector<SingleReduceOp> reductions;
242                 reductions.reserve(ipu_matrix.m); // One reduction for
                        ↪ every row of our matrix
243
244                 vector<Tensor> out;
245                 out.reserve(ipu_matrix.m);
246
247                 for (unsigned int block = 0; block <
                        ↪ ipu_matrix.blocks; block++)
248                 {
249                     for (unsigned int y = 0; y <
                            ↪ ipu_matrix.block_height && block *
                            ↪ ipu_matrix.block_height + y < ipu_matrix.m;
                            ↪ y++)
250                     {
251                         reductions.push_back(SingleReduceOp {
252                             res_vector_shuffled[block][y], {0},
                                    ↪ {popops::Operation::ADD}
253                         });
254
255                         out.push_back(tensors["vector"][block *
                                ↪ ipu_matrix.block_height + y]);
256                     }
257                 }
258
259                 auto p = Sequence{};
260                 popops::reduceMany(graph, reductions, out, p);
261                 program_reduce = p;
262
263             } else {
264                 std::cerr << "Using own reducer, this will lead to a
                        ↪ slower execution." << std::endl;
265
266                 for (unsigned int y = 0; y < ipu_matrix.blocks; y++)
267                 {
268                     auto v = graph.addVertex(reducer_cs,
                            ↪ "ReducerToVector", {
269                         {"res", tensors["res"][y]},
270                         {"vector", tensors["vector"].slice(
                                ↪ std::min(ipu_matrix.m, y *
                                ↪ ipu_matrix.block_height), std::min(
                                ↪ ipu_matrix.m, (y + 1) *
                                ↪ ipu_matrix.block_height))}
271                     });
272
273                     graph.setInitialValue(v["block_length"],
                            ↪ std::min((int)ipu_matrix.block_height,
                            ↪ std::max(0, (int)ipu_matrix.m -
                            ↪ (int)ipu_matrix.block_height * (int)y)));
274                     graph.setInitialValue(v["blocks"],
                            ↪ ipu_matrix.blocks);
275
276                     graph.setPerfEstimate(v, 100);
277                     graph.setTileMapping(v, ipu_matrix.blocks * y);
278                 }
279
280                 program_reduce = Execute(reducer_cs);
281             }
```

```
282
283             auto main_sequence = Sequence{Repeat(
284                 loops,
285                 Sequence{program_spmv, program_reduce})};
286
287             if (!Config::get().model)
288             {
289                 auto timing = poplar::cycleCount(graph, main_sequence,
                        ↪ 0, SyncType::INTERNAL, "timer");
290                 graph.createHostRead("readTimer", timing, true);
291             }
292
293             programs["main"] = main_sequence;
294         }
295
296         auto build_data_streams(Graph &graph, map<string, Tensor>
                ↪ &tensors, map<string, Program> &programs,
                ↪ IPUMatrix<float> &ipu_matrix)
297         {
298             auto toipu_matrix =
                    ↪ graph.addHostToDeviceFIFO("toipu_matrix", FLOAT,
                    ↪ ipu_matrix.matrix.size());
299             auto toipu_idx = graph.addHostToDeviceFIFO("toipu_idx",
                    ↪ INT, ipu_matrix.idx.size());
300             auto toipu_row_idx =
                    ↪ graph.addHostToDeviceFIFO("toipu_row_idx", INT,
                    ↪ ipu_matrix.row_idx.size());
301             auto toipu_vec = graph.addHostToDeviceFIFO("toipu_vec",
                    ↪ FLOAT, ipu_matrix.n);
302
303             auto fromipu_vec =
                    ↪ graph.addDeviceToHostFIFO("fromipu_vec", FLOAT,
                    ↪ ipu_matrix.n);
304
305             auto copyto_matrix = Copy(toipu_matrix, tensors["matrix"]);
306             auto copyto_idx = Copy(toipu_idx, tensors["idx"]);
307             auto copyto_row_idx = Copy(toipu_row_idx,
                    ↪ tensors["row_idx"]);
308             auto copyto_vec = Copy(toipu_vec, tensors["vector"]);
309
310             auto copyhost_vec = Copy(tensors["vector"], fromipu_vec);
311
312             programs["copy_to_ipu_matrix"] = Sequence{copyto_matrix,
                    ↪ copyto_idx, copyto_row_idx};
313             programs["copy_to_ipu_vec"] = copyto_vec;
314
315             programs["copy_to_host"] = copyhost_vec;
316         }
317
318         auto create_graph_add_codelets(const Device &device) -> Graph
319         {
320             auto graph = poplar::Graph(device.getTarget());
321
322             // Add our own codelets
323             graph.addCodelets({"codelets/spmv/MatrixBlock.cpp",
                    ↪ "codelets/spmv/ReducerToVector.cpp"}, "-O3 -I
                    ↪ codelets");
324             popops::addCodelets(graph);
325
326             return graph;
327         }
328     }
329
330     optional<ExperimentReportIPU> execute(const Device &device,
            ↪ matrix::SparseMatrix<float> &matrix, int rounds)
331     {
332         std::cerr << "Executing Sparse Matrix Vector multiplication
                ↪ experiment.." << std::endl;
```

```
333
334         if (rounds != 1 && matrix.rows() != matrix.cols())
335         {
336             std::cerr << "Multi-round was requested, but not supported
                ↪ by matrix." << std::endl;
337             return std::nullopt;
338         }
339
340         Graph graph = create_graph_add_codelets(device);
341
342         auto tensors = map<string, Tensor>{};
343         auto programs = map<string, Program>{};
344
345         auto ipu_matrix = prepare_data(matrix,
            ↪ device.getTarget().getNumTiles());
346
347         std::cerr << "Building programs.." << std::endl;
348
349         build_compute_graph(graph, tensors, programs,
            ↪ device.getTarget().getNumTiles(), ipu_matrix, rounds);
350         build_data_streams(graph, tensors, programs, ipu_matrix);
351
352         auto ENGINE_OPTIONS = OptionFlags{};
353
354         if (Config::get().debug)
355         {
356             ENGINE_OPTIONS = OptionFlags{
357                 {"autoReport.all", "true"}};
358         }
359
360         auto programIds = map<string, int>();
361         auto programsList = vector<Program>(programs.size());
362         int index = 0;
363         for (auto &nameToProgram : programs)
364         {
365             programIds[nameToProgram.first] = index;
366             programsList[index] = nameToProgram.second;
367             index++;
368         }
369
370         std::cerr << "Compiling graph.." << std::endl;
371
372         auto timing_graph_compilation_start =
            ↪ std::chrono::high_resolution_clock::now();
373         auto engine = Engine(graph, programsList, ENGINE_OPTIONS);
374         engine.load(device);
375         auto timing_graph_compilation_end =
            ↪ std::chrono::high_resolution_clock::now();
376         auto timing_graph_compilation =
            ↪ std::chrono::duration_cast<std::chrono::nanoseconds>(
            ↪ timing_graph_compilation_end -
            ↪ timing_graph_compilation_start).count() / 1e3;
377
378         if (Config::get().debug)
379         {
380             engine.enableExecutionProfiling();
381         }
382
383         auto vec = vector<float>(ipu_matrix.n, 1.0);
384
385         // TODO: if we change the input vector we need to apply the
            ↪ matrix mapping to it for a correct result.
386
387         engine.connectStream("toipu_matrix", ipu_matrix.matrix.data());
388         engine.connectStream("toipu_idx", ipu_matrix.idx.data());
389         engine.connectStream("toipu_row_idx",
            ↪ ipu_matrix.row_idx.data());
390         engine.connectStream("toipu_vec", vec.data());
```

```
391
392         auto result_vec = vector<float>(ipu_matrix.n);
393         engine.connectStream("fromipu_vec", result_vec.data());
394
395         // Run all programs in order
396         std::cerr << "Running programs.." << std::endl;
397         std::cerr << "Copy data to IPU\n";
398
399         auto copy_timing_start =
                ↪ std::chrono::high_resolution_clock::now();
400         engine.run(programIds["copy_to_ipu_matrix"], "copy matrix");
401         engine.run(programIds["copy_to_ipu_vec"], "copy vector");
402         auto copy_timing_end =
                ↪ std::chrono::high_resolution_clock::now();
403         auto copy_timing =
                ↪ std::chrono::duration_cast<std::chrono::nanoseconds>(
                ↪ copy_timing_end - copy_timing_start).count() / 1e3;
404
405         std::cerr << "Run main program\n";
406
407         auto execution_start =
                ↪ std::chrono::high_resolution_clock::now();
408         engine.run(programIds["main"], "main loop");
409         auto execution_end = std::chrono::high_resolution_clock::now();
410         auto execution_timing =
                ↪ std::chrono::duration_cast<std::chrono::nanoseconds>(
                ↪ execution_end - execution_start).count() / 1e3;
411
412         vector<unsigned long> ipuTimer(1);
413         if (!Config::get().model)
414         {
415             engine.readTensor("readTimer", ipuTimer.data(),
                    ↪ &*ipuTimer.end());
416             std::cerr << "Timing read: " << ipuTimer[0] << std::endl;
417         }
418
419         std::cerr << "Copying back result\n";
420
421         auto copyback_timing_start =
                ↪ std::chrono::high_resolution_clock::now();
422         engine.run(programIds["copy_to_host"], "copy result");
423         auto copyback_timing_end =
                ↪ std::chrono::high_resolution_clock::now();
424         auto copyback_timing =
                ↪ std::chrono::duration_cast<std::chrono::nanoseconds>(
                ↪ copyback_timing_end - copyback_timing_start).count() /
                ↪ 1e3;
425
426         // std::cout << "Resulting vector:\n";
427         long int res = 0;
428         for (auto v : result_vec)
429         {
430         //     std::cout << v << ", ";
431             res += static_cast<long int>(v);
432         }
433         // std::cout << std::endl;
434
435
436
437         std::cerr << "Sum: " << res << std::endl;
438
439         // setup result report
440         auto report = ExperimentReportIPU(std::move(engine),
                ↪ std::move(graph));
441         report.set_timing("copy", copy_timing);
442         report.set_timing("execution", execution_timing);
443         report.set_timing("copy_back", copybback_timing);
444         report.set_timing("graph_compilation",
```

```
            ↪ timing_graph_compilation);
445
446         if (!Config::get().model)
447         {
448             report.set_timing("ipu_report", ipuTimer[0] /
                ↪ device.getTarget().getTileClockFrequency());
449         }
450
451         return optional(std::move(report));
452     }
453 }
```

Listing A.2: SpMV `MatrixBlock` codelet

```
1  #include <poplar/Vertex.hpp>
2  #include <cstddef>
3  #include <cstdlib>
4  #include <math.h>
5  #include <stdint.h>
6  #include <assert.h>
7  #include <cmath>
8
9  using namespace poplar;
10
11 class MatrixBlock : public MultiVertex
12 {
13 public:
14     // Data structure:
15     // m[i] = M_(E_t where row_idx[t] >= i and row_idx[t + 1] < i ==>
           ↪ t, idx[i])
16     Input<Vector<float>> matrix;
17     Input<Vector<int>> idx;
18     Input<Vector<int>> row_idx;
19
20     Input<Vector<float>> vec;
21     Output<Vector<float>> res;
22
23     auto compute(unsigned workerId) -> bool
24     {
25         // Performs basic matrix * vector mult for block
26         // Go by row
27         for (auto i = workerId; i < row_idx.size() - 1; i+=
               ↪ MultiVertex::numWorkers())
28         {
29             float sum = 0.0;
30             for (auto j = row_idx[i]; j < row_idx[i + 1]; j++)
31             {
32                 sum += vec[idx[j]] * matrix[j];
33             }
34             res[i] = sum;
35         }
36
37         return true;
38     }
39 };
```

Listing A.3: SpMV `ReducerToVector` codelet

```
1  #include <poplar/Vertex.hpp>
2  #include <poplar/Loops.hpp>
3  #include <cstddef>
4  #include <cstdlib>
5  #include <math.h>
6  #include <stdint.h>
7  #include <assert.h>
8  #include <cmath>
```

```
 9
10 using namespace poplar;
11
12 class ReducerToVector : public MultiVertex
13 {
14 public:
15     // We sum 0, n, n*2, n*3 ... to vector[0]
16     // 1, n+1, n*2 + 1 ... to vector[1] etc.
17
18     Vector<Input<Vector<float>>> res;
19     Output<Vector<float>> vector;
20
21     int block_length;
22     int blocks;
23
24     auto compute(unsigned workerId) -> bool
25     {
26         for (int i = workerId; i < block_length; i+=
               ↪ MultiVertex::numWorkers())
27         {
28             auto sum = 0;
29             for (rptsize_t n = 0; n < blocks; n+=1)
30             {
31                 sum += res[n][i];
32             }
33             vector[i] = sum;
34         }
35
36         return true;
37     }
38 };
```

Listing A.4: Breadth-first search experiment host execution

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <algorithm>
 4 #include <cmath>
 5 #include <chrono>
 6 #include <limits.h>
 7
 8 #include <poplar/Engine.hpp>
 9 #include <poputil/TileMapping.hpp>
10 #include <poplar/DeviceManager.hpp>
11 #include <poplar/Program.hpp>
12 #include <poplar/CycleCount.hpp>
13 #include <popops/AllTrue.hpp>
14 #include <popops/ElementWise.hpp>
15 #include <popops/codelets.hpp>
16 #include <popops/Reduce.hpp>
17
18 #include "../matrix.hpp"
19 #include "../config.cpp"
20 #include "../ipu.cpp"
21 #include "../report.cpp"
22
23 using ::poplar::Device;
24 using ::poplar::Engine;
25 using ::poplar::Graph;
26 using ::poplar::Tensor;
27 using ::poplar::OptionFlags;
28 using ::poplar::SyncType;
29
30 using ::poplar::FLOAT;
31 using ::poplar::INT;
32 using ::poplar::UNSIGNED_INT;
33 using ::poplar::BOOL;
34
```

```
35 using ::poplar::program::Copy;
36 using ::poplar::program::RepeatWhileFalse;
37 using ::poplar::program::Execute;
38 using ::poplar::program::Program;
39 using ::poplar::program::Repeat;
40 using ::poplar::program::Sequence;
41
42 using ::popops::SingleReduceOp;
43 using ::popops::reduceMany;
44
45 namespace exp_bfs
46 {
47
48     // Helper functions for experiment
49     namespace
50     {
51         struct IPUMatrix
52         {
53         public:
54             IPUMatrix(vector<int> offsets, vector<int> idx,
                    ↪ vector<int> row_idx, int blocks, int block_height,
                    ↪ int m, int n, unsigned int frontier) :
                    ↪ offsets(offsets), idx(idx), row_idx(row_idx),
                    ↪ blocks(blocks), block_height(block_height), m(m),
                    ↪ n(n), frontier(frontier) {}
55
56             vector<int> offsets;
57
58             // actuall data
59             vector<int> idx; // better indexing type? size_t?
60             vector<int> row_idx;
61
62             // matrix data
63             unsigned int blocks;
64             unsigned int block_height;
65             unsigned int m;
66             unsigned int n;
67
68             unsigned int frontier;
69         };
70
71         auto prepare_data(matrix::SparseMatrix<float> matrix, const
                ↪ int num_tiles)
72         {
73             const auto blocks = (int) std::floor(std::sqrt(num_tiles));
74             const auto block_size_col = std::max(matrix.cols() /
                    ↪ blocks + (matrix.cols() % blocks != 0), 1);
75             const auto block_size_row = std::max(matrix.rows() /
                    ↪ blocks + (matrix.rows() % blocks != 0), 1);
76
77             // This will give the offsets for matrix and idx
78             vector<int> offsets(blocks * blocks + 1, 0);
79             vector<int> row_idx(blocks * blocks * (block_size_row +
                    ↪ 1), 0);
80
81             // We go through each value in the SpM and update offsets
                    ↪ and row_idx
82             for (auto o = 0; o < matrix.nonzeroes(); o++)
83             {
84                 auto [i, j, v] = matrix.get(o);
85                 (void)v;
86
87                 auto x = j / block_size_col;
88                 auto y = i / block_size_row;
89
90                 offsets[y * blocks + x + 1]++;
91                 row_idx[(y * blocks + x) * (block_size_row + 1) + (i -
                        ↪ (block_size_row * y)) + 1]++;
```

```
92              }
93
94              // Stride offsets and row_idx
95              for (size_t i = 2; i < offsets.size(); i++)
96              {
97                  offsets[i] += offsets[i - 1];
98              }
99
100             for (auto block = 0; block < blocks * blocks; block++)
101             {
102                 for (auto i = 0; i < block_size_row; i++)
103                 {
104                     row_idx[block * (block_size_row + 1) + i + 1] +=
                        ↪ row_idx[block * (block_size_row + 1) + i];
105                 }
106             }
107
108             assert(offsets[offsets.size() - 1] == matrix.nonzeroes());
109
110             vector<int> cursor(row_idx); // Cursor inside a block
                ↪ between rows
111
112             vector<int> idx(matrix.nonzeroes());
113
114             for (auto o = 0; o < matrix.nonzeroes(); o++)
115             {
116                 auto [i, j, v] = matrix.get(o);
117                 (void)v;
118
119                 auto x = j / block_size_col;
120                 auto y = i / block_size_row;
121
122                 size_t value_offset = offsets[y * blocks + x] +
                        ↪ cursor[(y * blocks + x) * (block_size_row + 1) +
                        ↪ (i - (block_size_row * y))];
123
124                 idx[value_offset] = j - (block_size_col * x);
125
126                 // Update cursor
127                 cursor[(y * blocks + x) * (block_size_row + 1) + (i -
                        ↪ (block_size_row * y))]++;
128             }
129
130             unsigned int frontier = 0;
131             if (matrix.applied_perm.has_value()) {
132                 frontier = matrix.applied_perm.value().apply(0);
133             }
134
135             return IPUMatrix(offsets, idx, row_idx, blocks,
                    ↪ block_size_row, matrix.rows(), matrix.cols(),
                    ↪ frontier);
136         }
137
138         void build_compute_graph(Graph &graph, map<string, Tensor>
                ↪ &tensors, map<string, Program> &programs, const int
                ↪ num_tiles, IPUMatrix &ipu_matrix)
139         {
140
141             // Static Matrix data
142             tensors["idx"] = graph.addVariable(INT,
                    ↪ {ipu_matrix.idx.size()}, "idx");
143             tensors["row_idx"] = graph.addVariable(INT,
                    ↪ {ipu_matrix.blocks, ipu_matrix.blocks,
                    ↪ ipu_matrix.block_height + 1}, "row_idx");
144
145             // Input/Output vector
146             tensors["vector"] = graph.addVariable(FLOAT, {(unsigned
                    ↪ int)ipu_matrix.n}, "vector");
```

```
147             graph.setInitialValue(tensors["vector"].slice(0,
                    ↪ ipu_matrix.n),
                    ↪ poplar::ArrayRef(vector<float>(ipu_matrix.n, 0.0)));
148             graph.setInitialValue(
                    ↪ tensors["vector"][ipu_matrix.frontier], 1.0); // our
                    ↪ first frontier value is always node 0 (col 0)
149
150             tensors["res"] = graph.addVariable(FLOAT,
                    ↪ {ipu_matrix.blocks, ipu_matrix.blocks,
                    ↪ ipu_matrix.block_height}, "result");
151
152             // We build the compute set for the MatrixBlock codelet
153             auto spmv_cs = graph.addComputeSet("spmv");
154
155             for (unsigned int y = 0; y < ipu_matrix.blocks; y++)
156             {
157                 for (unsigned int x = 0; x < ipu_matrix.blocks; x++)
158                 {
159                     auto block_id = y * ipu_matrix.blocks + x;
160                     auto v = graph.addVertex(spmv_cs, "MatrixBlock", {
161                         {"idx", tensors["idx"].slice(
                                ↪ ipu_matrix.offsets[block_id],
                                ↪ ipu_matrix.offsets[block_id + 1])},
162                         {"row_idx", tensors["row_idx"][y][x]},
163                         {"vec", tensors["vector"].slice(
                                ↪ std::min(ipu_matrix.m, x *
                                ↪ ipu_matrix.block_height),
                                ↪ std::min(ipu_matrix.m, (x + 1) *
                                ↪ ipu_matrix.block_height))},
164                         {"res", tensors["res"][y][x]}
165                     });
166
167                     graph.setPerfEstimate(v, 100);
168                     graph.setTileMapping(v, block_id);
169
170                     graph.setTileMapping(tensors["idx"].slice(
                            ↪ ipu_matrix.offsets[block_id],
                            ↪ ipu_matrix.offsets[block_id + 1]), block_id);
171                     graph.setTileMapping(tensors["row_idx"][y][x],
                            ↪ block_id);
172                     graph.setTileMapping(tensors["vector"].slice(
                            ↪ std::min(ipu_matrix.m, x *
                            ↪ ipu_matrix.block_height),
                            ↪ std::min(ipu_matrix.m, (x + 1) *
                            ↪ ipu_matrix.block_height)), block_id);
173                     graph.setTileMapping(tensors["res"][y][x],
                            ↪ block_id);
174                 }
175             }
176
177             auto program_spmv = Execute(spmv_cs);
178
179             // We build the compute set for addition
180             auto res_vector_shuffled = tensors["res"].dimShuffle({0,
                    ↪ 2, 1});
181
182             vector<SingleReduceOp> reductions;
183             reductions.reserve(ipu_matrix.m); // One reduction for
                    ↪ every row of our matrix
184
185             vector<Tensor> out;
186             out.reserve(ipu_matrix.m);
187
188             for (unsigned int block = 0; block < ipu_matrix.blocks;
                    ↪ block++)
189             {
190                 for (unsigned int y = 0; y < ipu_matrix.block_height
                        ↪ && block * ipu_matrix.block_height + y <
```

```
                ↪ ipu_matrix.m; y++)
191             {
192                 reductions.push_back(SingleReduceOp {
193                     res_vector_shuffled[block][y], {0},
                        ↪ {popops::Operation::ADD}
194                 });
195
196                 out.push_back(tensors["vector"][block *
                    ↪ ipu_matrix.block_height + y]);
197             }
198         }
199
200         auto program_reduce = Sequence{};
201         popops::reduceMany(graph, reductions, out, program_reduce);
202
203         // So far this is a basic copy of SpMV, now the magic:
204         // We need to:
205         // 1. Keep track of a dist, and iteration tensor
206         // 2. Copy over `vector` (if v[i] > 1 ==> dist[i] =
            ↪ min(dist[i], iteration)) to dist
207         // 3. Normalize vector again (ones)
208         // 4. Keep track if we should continue! (We perform some
            ↪ sub-reductions to speed up this process)
209
210         tensors["dist"] = graph.addVariable(UNSIGNED_INT,
            ↪ {(unsigned int)ipu_matrix.n}, "dist");
211         graph.setInitialValue( tensors["dist"], poplar::ArrayRef(
            ↪ vector<unsigned int>(ipu_matrix.n, UINT_MAX)));
212         graph.setInitialValue(tensors["dist"]
            ↪ [ipu_matrix.frontier], 0);
213
214         tensors["iteration"] = graph.addVariable(UNSIGNED_INT,
            ↪ {1}, "iteration");
215         graph.setTileMapping(tensors["iteration"], 0);
216         graph.setInitialValue(tensors["iteration"][0], 1);
217
218         tensors["stop"] = graph.addVariable(BOOL, {(unsigned
            ↪ long)num_tiles}, "stop condition");
219         //graph.setInitialValue(tensors["stop"],
            ↪ poplar::ArrayRef(vector<char>(num_tiles, false)));
            ↪ // We use char here because a bool is a 1-bit value
            ↪ in cpp
220
221         auto normalize_cs = graph.addComputeSet("normalize");
222
223         unsigned int rows_per_tile = std::max(ipu_matrix.m /
            ↪ num_tiles + (ipu_matrix.m % num_tiles != 0),
            ↪ (unsigned int) 1);
224         for (unsigned int i = 0; i < static_cast<unsigned
            ↪ int>(num_tiles); i++) {
225             unsigned int row_start = std::min(i * rows_per_tile,
                ↪ ipu_matrix.m);
226             unsigned int row_end = std::min((i + 1) *
                ↪ rows_per_tile, ipu_matrix.m);
227
228             auto v = graph.addVertex(normalize_cs, "Normalize", {
229                 {"vec", tensors["vector"].slice(row_start,
                    ↪ row_end)},
230                 {"dist", tensors["dist"].slice(row_start,
                    ↪ row_end)},
231                 {"iteration", tensors["iteration"][0]},
232                 {"stop", tensors["stop"][i]}
233             });
234
235             graph.setPerfEstimate(v, 100);
236             graph.setTileMapping(v, i);
237
238             graph.setTileMapping(tensors["stop"][i], i);
```

```
239                 graph.setTileMapping(tensors["dist"].slice(row_start,
                        ↪ row_end), i);
240             }
241
242             auto program_normalize = Execute(normalize_cs);
243
244             auto program_sequence = Sequence{
245                 program_spmv, program_reduce, program_normalize
246             };
247
248             tensors["should_stop"] = popops::allTrue(graph,
                    ↪ tensors["stop"], program_sequence, "check stop
                    ↪ condition");
249             popops::mapInPlace(graph, popops::expr::_1 +
                    ↪ popops::expr::Const(1), {tensors["iteration"][0]},
                    ↪ program_sequence, "add 1 to iteration");
250
251             auto main_sequence = Sequence{RepeatWhileFalse(Sequence(),
                    ↪ tensors["should_stop"], program_sequence)};
252
253             if (!Config::get().model)
254             {
255                 auto timing = poplar::cycleCount(graph, main_sequence,
                        ↪ 0, SyncType::INTERNAL, "timer");
256                 graph.createHostRead("readTimer", timing, true);
257             }
258
259             programs["main"] = main_sequence;
260         }
261
262         auto build_data_streams(Graph &graph, map<string, Tensor>
                ↪ &tensors, map<string, Program> &programs, IPUMatrix
                ↪ &ipu_matrix)
263         {
264             auto toipu_idx = graph.addHostToDeviceFIFO("toipu_idx",
                    ↪ INT, ipu_matrix.idx.size());
265             auto toipu_row_idx =
                    ↪ graph.addHostToDeviceFIFO("toipu_row_idx", INT,
                    ↪ ipu_matrix.row_idx.size());
266
267             auto fromipu_dist =
                    ↪ graph.addDeviceToHostFIFO("fromipu_dist",
                    ↪ UNSIGNED_INT, ipu_matrix.n);
268
269             auto copyto_idx = Copy(toipu_idx, tensors["idx"]);
270             auto copyto_row_idx = Copy(toipu_row_idx,
                    ↪ tensors["row_idx"]);
271
272             auto copyhost_vec = Copy(tensors["dist"], fromipu_dist);
273
274             programs["copy_to_ipu_matrix"] = Sequence{copyto_idx,
                    ↪ copyto_row_idx};
275             programs["copy_to_host"] = copyhost_vec;
276         }
277
278         auto create_graph_add_codelets(const Device &device) -> Graph
279         {
280             auto graph = poplar::Graph(device.getTarget());
281
282             // Add our own codelets
283             graph.addCodelets({"codelets/bfs/MatrixBlock.cpp",
                    ↪ "codelets/bfs/Normalize.cpp"}, "-O3 -I codelets");
284             popops::addCodelets(graph);
285
286             return graph;
287         }
288     }
289
```

```
290     optional<ExperimentReportIPU> execute(const Device &device,
            ↪ matrix::SparseMatrix<float> &matrix)
291     {
292         std::cerr << "Executing BFS experiment.." << std::endl;
293
294         Graph graph = create_graph_add_codelets(device);
295
296         auto tensors = map<string, Tensor>{};
297         auto programs = map<string, Program>{};
298
299         auto ipu_matrix = prepare_data(matrix,
                ↪ device.getTarget().getNumTiles());
300
301         std::cerr << "Building programs.." << std::endl;
302
303         build_compute_graph(graph, tensors, programs,
                ↪ device.getTarget().getNumTiles(), ipu_matrix);
304         build_data_streams(graph, tensors, programs, ipu_matrix);
305
306         auto ENGINE_OPTIONS = OptionFlags{};
307
308         if (Config::get().debug)
309         {
310             ENGINE_OPTIONS = OptionFlags{
311                 {"autoReport.all", "true"}};
312         }
313
314         auto programIds = map<string, int>();
315         auto programsList = vector<Program>(programs.size());
316         int index = 0;
317         for (auto &nameToProgram : programs)
318         {
319             programIds[nameToProgram.first] = index;
320             programsList[index] = nameToProgram.second;
321             index++;
322         }
323
324         std::cerr << "Compiling graph.." << std::endl;
325
326         auto timing_graph_compilation_start =
                ↪ std::chrono::high_resolution_clock::now();
327         auto engine = Engine(graph, programsList, ENGINE_OPTIONS);
328         engine.load(device);
329         auto timing_graph_compilation_end =
                ↪ std::chrono::high_resolution_clock::now();
330         auto timing_graph_compilation =
                ↪ std::chrono::duration_cast<std::chrono::nanoseconds>(
                ↪ timing_graph_compilation_end -
                ↪ timing_graph_compilation_start).count() / 1e3;
331
332         if (Config::get().debug)
333         {
334             engine.enableExecutionProfiling();
335         }
336
337         auto vec = vector<float>(ipu_matrix.n, 1.0);
338
339         // TODO: if we change the input vector we need to apply the
                ↪ matrix mapping to it for a correct result.
340
341         engine.connectStream("toipu_idx", ipu_matrix.idx.data());
342         engine.connectStream("toipu_row_idx",
                ↪ ipu_matrix.row_idx.data());
343
344         auto result_dist = vector<unsigned int>(ipu_matrix.n);
345         engine.connectStream("fromipu_dist", result_dist.data());
346
347         // Run all programs in order
```

```
348         std::cerr << "Running programs.." << std::endl;
349         std::cerr << "Copy data to IPU\n";
350
351         auto copy_timing_start =
                ↪ std::chrono::high_resolution_clock::now();
352         engine.run(programIds["copy_to_ipu_matrix"], "copy matrix");
353         auto copy_timing_end =
                ↪ std::chrono::high_resolution_clock::now();
354         auto copy_timing =
                ↪ std::chrono::duration_cast<std::chrono::nanoseconds>(
                ↪ copy_timing_end - copy_timing_start).count() / 1e3;
355
356         std::cerr << "Run main program\n";
357
358         auto execution_start =
                ↪ std::chrono::high_resolution_clock::now();
359         engine.run(programIds["main"], "main loop");
360         auto execution_end = std::chrono::high_resolution_clock::now();
361         auto execution_timing =
                ↪ std::chrono::duration_cast<std::chrono::nanoseconds>(
                ↪ execution_end - execution_start).count() / 1e3;
362
363         vector<unsigned long> ipuTimer(1);
364         if (!Config::get().model)
365         {
366             engine.readTensor("readTimer", ipuTimer.data(),
                    ↪ &*ipuTimer.end());
367             std::cerr << "Timing read: " << ipuTimer[0] << std::endl;
368         }
369
370         std::cerr << "Copying back result\n";
371
372         auto copyback_timing_start =
                ↪ std::chrono::high_resolution_clock::now();
373         engine.run(programIds["copy_to_host"], "copy result");
374         auto copyback_timing_end =
                ↪ std::chrono::high_resolution_clock::now();
375         auto copyback_timing =
                ↪ std::chrono::duration_cast<std::chrono::nanoseconds>(
                ↪ copyback_timing_end - copyback_timing_start).count() /
                ↪ 1e3;
376
377         std::cerr << "Resulting vector:\n";
378         long int res = 0;
379         for (auto v : result_dist)
380         {
381             std::cerr << v << ", ";
382             res += static_cast<long int>(v);
383         }
384         std::cerr << std::endl;
385         std::cerr << "Sum: " << res << std::endl;
386
387         // setup result report
388         auto report = ExperimentReportIPU(std::move(engine),
                ↪ std::move(graph));
389         report.set_timing("copy", copy_timing);
390         report.set_timing("execution", execution_timing);
391         report.set_timing("copy_back", copyback_timing);
392         report.set_timing("graph_compilation",
                ↪ timing_graph_compilation);
393
394         if (!Config::get().model)
395         {
396             report.set_timing("ipu_report", ipuTimer[0] /
                    ↪ device.getTarget().getTileClockFrequency());
397         }
398
399         return optional(std::move(report));
```

```
400     }
401 }
```

Listing A.5: BFS `MatrixBlock` codelet

```
#include <poplar/Vertex.hpp>
#include <cstddef>
#include <cstdlib>
#include <math.h>
#include <stdint.h>
#include <assert.h>
#include <cmath>

using namespace poplar;

class MatrixBlock : public MultiVertex
{
public:
    // Data structure:
    // m[i] = 1 <==> M_(E_t where row_idx[t] >= i and row_idx[t + 1] <
       ↪ i ==> t, idx[i])
    Input<Vector<int>> idx;
    Input<Vector<int>> row_idx;

    Input<Vector<float>> vec;
    Output<Vector<float>> res;

    auto compute(unsigned workerId) -> bool
    {
        // Performs basic matrix * vector mult for block
        // Go by row
        for (auto i = workerId; i < row_idx.size() - 1; i+=
           ↪ MultiVertex::numWorkers())
        {
            res[i] = 0.0;

            for (auto j = row_idx[i]; j < row_idx[i + 1]; j++)
            {
                if (vec[idx[j]] > 0) {
                    res[i] = 1.0;
                    goto cnt;
                }
            }

            cnt:;
        }

        return true;
    }
};
```

Listing A.6: BFS `Normalize` codelet

```
#include <poplar/Vertex.hpp>
#include <poplar/Loops.hpp>
#include <cstddef>
#include <cstdlib>
#include <math.h>
#include <stdint.h>
#include <assert.h>
#include <cmath>
#include <limits.h>

using namespace poplar;

class Normalize : public Vertex
{
```

```
15 public:
16     InOut<Vector<unsigned int>> dist;
17     InOut<Vector<float>> vec;
18
19     Input<unsigned int> iteration;
20     Output<bool> stop;
21
22     auto compute() -> bool
23     {
24         *stop = true;
25         for (rptsize_t i = 0; i < vec.size(); i += 1)
26         {
27             if (vec[i] >= 1.0)
28             {
29                 if (dist[i] == UINT_MAX)
30                 {
31                     dist[i] = *iteration;
32                     vec[i] = 1.0;
33                     *stop = false;
34                 }
35                 else
36                 {
37                     vec[i] = 0.0;
38                 }
39             }
40         }
41
42         return true;
43     }
44 };
```

Listing A.7: Prim's algorithm experiment host execution

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <algorithm>
4  #include <cmath>
5  #include <chrono>
6  #include <float.h>
7  #include <limits.h>
8
9  #include <poplar/Engine.hpp>
10 #include <poplar/DeviceManager.hpp>
11 #include <poplar/Program.hpp>
12 #include <popops/codelets.hpp>
13 #include <poplar/CycleCount.hpp>
14 #include <poplar/PrintTensor.hpp>
15
16 #include "../matrix.hpp"
17 #include "../config.cpp"
18 #include "../ipu.cpp"
19 #include "../report.cpp"
20
21 using ::poplar::Device;
22 using ::poplar::Engine;
23 using ::poplar::Graph;
24 using ::poplar::OptionFlags;
25 using ::poplar::Tensor;
26 using ::poplar::SyncType;
27
28 using ::poplar::program::Program;
29 using ::poplar::program::Copy;
30 using ::poplar::program::Sequence;
31 using ::poplar::program::Execute;
32 using ::poplar::program::RepeatWhileTrue;
33
34 using ::poplar::FLOAT;
35 using ::poplar::INT;
```

```
36 using ::poplar::UNSIGNED_INT;
37 using ::poplar::BOOL;
38
39 namespace exp_prims
40 {
41     // Private namespace
42     namespace
43     {
44         struct PrimsIPU
45         {
46             PrimsIPU(vector<int> values, vector<unsigned int> idx_row,
                ↪ vector<unsigned int> idx_col, unsigned int n,
                ↪ vector<size_t> block_indices, vector<size_t>
                ↪ block_row_lt, unsigned int blocks) : values(values),
                ↪ idx_row(idx_row), idx_col(idx_col), n(n),
                ↪ block_indices(block_indices),
                ↪ block_row_lt(block_row_lt), blocks(blocks){};
47
48             vector<int> values;
49             vector<unsigned int> idx_row;
50             vector<unsigned int> idx_col;
51             unsigned int n;
52
53             vector<size_t> block_indices;
54             vector<size_t> block_row_lt;
55             unsigned int blocks;
56         };
57
58         PrimsIPU prepare_data(matrix::SparseMatrix<float> &matrix,
            ↪ const int num_tiles)
59         {
60             unsigned amount_of_blocks = (unsigned)num_tiles;
61
62             // We are building up a CSC data structure, divided over
                ↪ num_tiles blocks vertically
63
64             // First we need to divide our matrix up over the blocks,
                ↪ so we need to go through all nz-values and make a
                ↪ prefix
65             // lookup for the rows. Then we can divide evenly with
                ↪ around matrix.nonzeroes() / num_tiles values per
                ↪ block
66
67             vector<int> rows_size = vector(matrix.rows(), 0);
68             for (size_t o = 0; o < (unsigned)matrix.nonzeroes(); o++)
69             {
70                 rows_size[get<0>(matrix.get(o))]++;
71             }
72
73             int values_per_block =
                ↪ std::max((unsigned)matrix.nonzeroes() /
                ↪ amount_of_blocks + (matrix.nonzeroes() %
                ↪ amount_of_blocks != 0), (unsigned)1);
74
75             // Our 3 main data structures
76             vector<int> ipu_values(matrix.nonzeroes(), 0.0);
77             vector<unsigned> ipu_row(matrix.nonzeroes(), 0.0);
78             vector<unsigned> ipu_column((matrix.cols() + 1) *
                ↪ amount_of_blocks, 0.0);
79
80             // pointer structure for where each block lies in values &
                ↪ row
81             vector<size_t> blocks(amount_of_blocks + 1, 0);
82             vector<size_t> block_row_lt(amount_of_blocks + 1, 0);
83             vector<size_t> block_lt(matrix.rows(), 0);
84
85             // Now we divide up the rows
86             // Logic: we count non-zero values until it's equal or
```

```
            ↪ above the tresshold (block_cursor * values_per_block)
87          unsigned int block_cursor = 0;
88          unsigned int total = 0;
89          for (size_t row = 0; row < (unsigned)matrix.rows(); row++)
90          {
91              total += rows_size[row];
92              block_lt[row] = block_cursor;
93
94              if (total >= (block_cursor + 1) * values_per_block)
95              {
96                  blocks[block_cursor + 1] = total;
97                  block_row_lt[block_cursor + 1] = row + 1;
98                  block_cursor++;
99              }
100         }
101
102         for (; block_cursor < (unsigned)blocks.size() - 1;
                ↪ block_cursor++)
103         {
104             blocks[block_cursor + 1] = total;
105             block_row_lt[block_cursor + 1] = matrix.rows();
106         }
107
108         // Now we can start calculating ipu_column
109         for (size_t o = 0; o < (unsigned)matrix.nonzeroes(); o++)
110         {
111             auto [i, j, v] = matrix.get(o);
112             (void)v;
113
114             auto block = block_lt[i];
115
116             ipu_column[(block * (matrix.cols() + 1)) + j + 1]++;
117         }
118
119         // stride ipu_col for each block
120         for (size_t block = 0; block < amount_of_blocks; block++)
121         {
122             for (size_t idx = 2; idx < (unsigned)matrix.cols();
                    ↪ idx++)
123             {
124                 ipu_column[(block * (matrix.cols() + 1)) + idx] +=
                        ↪ ipu_column[(block * (matrix.cols() + 1)) +
                        ↪ idx - 1];
125             }
126         }
127
128         // populate ipu_values / ipu_row
129         vector<unsigned> col_cursor(ipu_column);
130
131         for (size_t o = 0; o < (unsigned)matrix.nonzeroes(); o++)
132         {
133             auto [i, j, v] = matrix.get(o);
134             auto block = block_lt[i];
135
136             auto offset = blocks[block] + col_cursor[(block *
                    ↪ (matrix.cols() + 1)) + j];
137             ipu_values[offset] = (int)v;
138             ipu_row[offset] = i - block_row_lt[block];
139
140             col_cursor[(block * (matrix.cols() + 1)) + j]++;
141         }
142
143         return PrimsIPU(ipu_values, ipu_row, ipu_column,
                ↪ matrix.cols(), blocks, block_row_lt,
                ↪ amount_of_blocks);
144     }
145
146     auto create_graph_add_codelets(const Device &device) -> Graph
```

```
147         {
148             auto graph = poplar::Graph(device.getTarget());
149 
150             // Add our own codelets
151             graph.addCodelets({"codelets/prims/PrimsBlock.cpp",
                    ↪ "codelets/prims/ReduceBlock.cpp",
                    ↪ "codelets/prims/GatherResult.cpp"}, "-I codelets
                    ↪ -O3");
152             popops::addCodelets(graph);
153 
154             return graph;
155         }
156 
157         void build_compute_graph(Graph &graph, map<string, Tensor>
                ↪ &tensors, map<string, Program> &programs, const int
                ↪ num_tiles, PrimsIPU &ipu_matrix, const int loops)
158         {
159             // The algorithm step by step:
160             // 1. we select a vertex (initial 0, current is the
                    ↪ previously added vertex)
161             // 2. we update dist/dist_prev with that specific vertex
                    ↪ (computeset update_dist)
162             // 2.1 min of dist to that vertex and current dist
163             // 2.2 or remove of current vertex
164             // 3. we select the minimal dist (computeset reduce_dist +
                    ↪ single reduction over the result)
165             // 4. update connection for that vertex and repeat with
                    ↪ new vertex (computeset update)
166 
167             // Static matrix data
168             tensors["weights"] = graph.addVariable(INT,
                    ↪ {ipu_matrix.values.size()});
169             tensors["idx_row"] = graph.addVariable(UNSIGNED_INT,
                    ↪ {ipu_matrix.idx_row.size()});
170             tensors["idx_col"] = graph.addVariable(UNSIGNED_INT,
                    ↪ {ipu_matrix.blocks, ipu_matrix.n + 1});
171 
172             // Result structs
173             tensors["connection"] = graph.addVariable(UNSIGNED_INT,
                    ↪ {ipu_matrix.n});
174             graph.setInitialValue(tensors["connection"][0], 0);
175 
176             tensors["dist"] = graph.addVariable(INT, {ipu_matrix.n});
177             graph.setInitialValue(tensors["dist"].slice(0,
                    ↪ ipu_matrix.n),
                    ↪ poplar::ArrayRef(vector<int>(ipu_matrix.n,
                    ↪ INT_MAX)));
178             tensors["dist_prev"] = graph.addVariable(UNSIGNED_INT,
                    ↪ {ipu_matrix.n});
179 
180             // Sub-results
181             // We store results in dist then reduce for each block,
                    ↪ finally find min in reduction (single-threaded)
182             tensors["block_dist"] = graph.addVariable(INT,
                    ↪ {ipu_matrix.blocks});
183             tensors["block_dist_from"] =
                    ↪ graph.addVariable(UNSIGNED_INT, {ipu_matrix.blocks});
184             tensors["block_dist_to"] = graph.addVariable(UNSIGNED_INT,
                    ↪ {ipu_matrix.blocks});
185 
186             // Cursor(s)
187             tensors["current"] = graph.addVariable(UNSIGNED_INT, {1});
188             graph.setInitialValue(tensors["current"][0], 0);
189 
190             // Loop variable
191             tensors["should_continue"] = graph.addVariable(BOOL, {1});
192             graph.setInitialValue(tensors["should_continue"][0], 1);
193 
```

```
194             auto update_d_cs = graph.addComputeSet("update_dist");
195
196             for (unsigned int block = 0; block < ipu_matrix.blocks;
                    ↪ block++)
197             {
198                 auto v = graph.addVertex(update_d_cs, "PrimsBlock", {
199                     {"weights", tensors["weights"].slice(
                            ↪ ipu_matrix.block_indices[block],
                            ↪ ipu_matrix.block_indices[block + 1])},
200                     {"rows", tensors["idx_row"].slice(
                            ↪ ipu_matrix.block_indices[block],
                            ↪ ipu_matrix.block_indices[block + 1])},
201                     {"columns", tensors["idx_col"][block]},
202                     {"dist", tensors["dist"].slice(
                            ↪ ipu_matrix.block_row_lt[block],
                            ↪ ipu_matrix.block_row_lt[block + 1])},
203                     {"dist_prev", tensors["dist_prev"].slice(
                            ↪ ipu_matrix.block_row_lt[block],
                            ↪ ipu_matrix.block_row_lt[block + 1])},
204                     {"current", tensors["current"][0]}
205                 });
206
207                 graph.setTileMapping(tensors["weights"].slice(
                        ↪ ipu_matrix.block_indices[block],
                        ↪ ipu_matrix.block_indices[block + 1]), block);
208                 graph.setTileMapping(tensors["idx_row"].slice(
                        ↪ ipu_matrix.block_indices[block],
                        ↪ ipu_matrix.block_indices[block + 1]), block);
209                 graph.setTileMapping(tensors["idx_col"][block], block);
210
211                 graph.setTileMapping(tensors["dist"].slice(
                        ↪ ipu_matrix.block_row_lt[block],
                        ↪ ipu_matrix.block_row_lt[block + 1]), block);
212                 graph.setTileMapping(tensors["dist_prev"].slice(
                        ↪ ipu_matrix.block_row_lt[block],
                        ↪ ipu_matrix.block_row_lt[block + 1]), block);
213
214                 graph.setInitialValue(v["row_offset"],
                        ↪ ipu_matrix.block_row_lt[block]);
215
216                 graph.setPerfEstimate(v, 100); // Needed for simulator
217                 graph.setTileMapping(v, block);
218             }
219
220             auto program_update_d = Execute(update_d_cs);
221
222             auto reduce_d_cs = graph.addComputeSet("reduce_dist");
223
224             for (unsigned int block = 0; block < ipu_matrix.blocks;
                    ↪ block++)
225             {
226                 auto v = graph.addVertex(reduce_d_cs,
                        ↪ "ReduceBlockSupervisor", {
227                     {"dist", tensors["dist"].slice(
                            ↪ ipu_matrix.block_row_lt[block],
                            ↪ ipu_matrix.block_row_lt[block + 1])},
228                     {"dist_prev", tensors["dist_prev"].slice(
                            ↪ ipu_matrix.block_row_lt[block],
                            ↪ ipu_matrix.block_row_lt[block + 1])},
229                     {"block_dist", tensors["block_dist"][block]},
230                     {"block_dist_from",
                            ↪ tensors["block_dist_from"][block]},
231                     {"block_dist_to", tensors["block_dist_to"][block]},
232                 });
233
234                 graph.setFieldSize(v["tmp1"], 6);
235                 graph.setFieldSize(v["tmp2"], 6);
236                 graph.setFieldSize(v["tmp3"], 6);
```

```
237
238                 graph.setPerfEstimate(v, 100); // Needed for simulator
239                 graph.setTileMapping(v, block);
240
241                 graph.setTileMapping(tensors["block_dist"][block],
                        ↪ block);
242                 graph.setTileMapping(
                        ↪ tensors["block_dist_from"][block], block);
243                 graph.setTileMapping( tensors["block_dist_to"][block],
                        ↪ block);
244
245                 graph.setInitialValue(v["row_offset"],
                        ↪ ipu_matrix.block_row_lt[block]);
246             }
247
248             auto program_reduce_d = Execute(reduce_d_cs);
249
250             auto gather_result_cs =
                    ↪ graph.addComputeSet("gather_results");
251             auto v = graph.addVertex(gather_result_cs,
                    ↪ "GatherResultSupervisor", {
252                 {"block_dist", tensors["block_dist"]},
253                 {"block_dist_from", tensors["block_dist_from"]},
254                 {"block_dist_to", tensors["block_dist_to"]},
255                 {"current", tensors["current"][0]},
256                 {"connection", tensors["connection"]},
257                 {"should_continue", tensors["should_continue"][0]}
258             });
259
260             graph.setPerfEstimate(v, 100); // Needed for simulator
261             graph.setTileMapping(v, ipu_matrix.blocks >> 1);
262
263             graph.setTileMapping(tensors["should_continue"][0],
                    ↪ ipu_matrix.blocks >> 1);
264             graph.setTileMapping(tensors["current"][0],
                    ↪ ipu_matrix.blocks >> 1);
265             graph.setTileMapping(tensors["connection"],
                    ↪ ipu_matrix.blocks >> 1);
266
267             graph.setFieldSize(v["tmp1"], 6);
268             graph.setFieldSize(v["tmp2"], 6);
269             graph.setFieldSize(v["tmp3"], 6);
270
271             auto program_gather_result = Execute(gather_result_cs);
272
273             auto main_sequence = Sequence{RepeatWhileTrue(Sequence{},
                    ↪ tensors["should_continue"][0],
                    ↪ Sequence{program_update_d, program_reduce_d,
                    ↪ program_gather_result})};
274
275             if (!Config::get().model)
276             {
277                 auto timing = poplar::cycleCount(graph, main_sequence,
                        ↪ 0, SyncType::INTERNAL, "timer");
278                 graph.createHostRead("readTimer", timing, true);
279             }
280
281             programs["main"] = main_sequence;
282         }
283
284         auto build_data_streams(Graph &graph, map<string, Tensor>
                ↪ &tensors, map<string, Program> &programs, PrimsIPU
                ↪ &ipu_matrix)
285         {
286             auto toipu_weights =
                    ↪ graph.addHostToDeviceFIFO("toipu_weights", INT,
                    ↪ ipu_matrix.values.size());
287             auto toipu_idx_row =
```

```
            ↪ graph.addHostToDeviceFIFO("toipu_idx_row",
            ↪ UNSIGNED_INT, ipu_matrix.idx_row.size());
288         auto toipu_idx_col =
            ↪ graph.addHostToDeviceFIFO("toipu_idx_col",
            ↪ UNSIGNED_INT, ipu_matrix.idx_col.size());
289
290         auto fromipu_connection =
            ↪ graph.addDeviceToHostFIFO("fromipu_connection",
            ↪ UNSIGNED_INT, ipu_matrix.n);
291
292         auto copyto_weights = Copy(toipu_weights,
            ↪ tensors["weights"]);
293         auto copyto_idx_row = Copy(toipu_idx_row,
            ↪ tensors["idx_row"]);
294         auto copyto_idx_col = Copy(toipu_idx_col,
            ↪ tensors["idx_col"]);
295
296         auto copyhost_connection = Copy(tensors["connection"],
            ↪ fromipu_connection);
297
298         programs["copy_to_ipu_matrix"] = Sequence{copyto_weights,
            ↪ copyto_idx_row, copyto_idx_col};
299         programs["copy_to_host"] = copyhost_connection;
300     }
301
302 }
303
304 optional<ExperimentReportIPU> execute(const Device &device,
    ↪ matrix::SparseMatrix<float> &matrix, int rounds)
305 {
306     std::cerr << "Executing Prims experiment.." << std::endl;
307
308     if (Config::get().model)
309     {
310         std::cerr << "Using the simulator is not supported by this
            ↪ experiment" << std::endl;
311         return std::nullopt;
312     }
313
314     auto ipu_matrix = prepare_data(matrix,
        ↪ device.getTarget().getNumTiles());
315
316     auto graph = create_graph_add_codelets(device);
317     auto tensors = map<string, Tensor>{};
318     auto programs = map<string, Program>{};
319
320     build_compute_graph(graph, tensors, programs,
        ↪ device.getTarget().getNumTiles(), ipu_matrix, rounds);
321     build_data_streams(graph, tensors, programs, ipu_matrix);
322
323     auto ENGINE_OPTIONS = OptionFlags{};
324
325     if (Config::get().debug)
326     {
327         ENGINE_OPTIONS = OptionFlags{
328             {"autoReport.all", "true"}};
329     }
330
331     auto programIds = map<string, int>();
332     auto programsList = vector<Program>(programs.size());
333     int index = 0;
334     for (auto &nameToProgram : programs)
335     {
336         programIds[nameToProgram.first] = index;
337         programsList[index] = nameToProgram.second;
338         index++;
339     }
340
```

```
341         std::cerr << "Compiling graph.." << std::endl;
342
343         auto timing_graph_compilation_start =
                ↪ std::chrono::high_resolution_clock::now();
344         auto engine = Engine(graph, programsList, ENGINE_OPTIONS);
345         engine.load(device);
346         auto timing_graph_compilation_end =
                ↪ std::chrono::high_resolution_clock::now();
347         auto timing_graph_compilation =
                ↪ std::chrono::duration_cast<std::chrono::nanoseconds>(
                ↪ timing_graph_compilation_end -
                ↪ timing_graph_compilation_start).count() / 1e3;
348
349         if (Config::get().debug)
350         {
351             engine.enableExecutionProfiling();
352         }
353
354         engine.connectStream("toipu_weights",
                ↪ ipu_matrix.values.data());
355         engine.connectStream("toipu_idx_row",
                ↪ ipu_matrix.idx_row.data());
356         engine.connectStream("toipu_idx_col",
                ↪ ipu_matrix.idx_col.data());
357
358         auto result_connection_vec = vector<unsigned>(ipu_matrix.n);
359         engine.connectStream("fromipu_connection",
                ↪ result_connection_vec.data());
360
361         // Run all programs in order
362         std::cerr << "Running programs.." << std::endl;
363         std::cerr << "Copy data to IPU\n";
364
365         auto copy_timing_start =
                ↪ std::chrono::high_resolution_clock::now();
366         engine.run(programIds["copy_to_ipu_matrix"], "copy matrix");
367         auto copy_timing_end =
                ↪ std::chrono::high_resolution_clock::now();
368         auto copy_timing =
                ↪ std::chrono::duration_cast<std::chrono::nanoseconds>(
                ↪ copy_timing_end - copy_timing_start).count() / 1e3;
369
370         std::cerr << "Run main program\n";
371
372         auto execution_start =
                ↪ std::chrono::high_resolution_clock::now();
373         engine.run(programIds["main"], "main loop");
374         auto execution_end = std::chrono::high_resolution_clock::now();
375         auto execution_timing =
                ↪ std::chrono::duration_cast<std::chrono::nanoseconds>(
                ↪ execution_end - execution_start).count() / 1e3;
376
377         vector<unsigned long> ipuTimer(1);
378         if (!Config::get().model)
379         {
380             engine.readTensor("readTimer", ipuTimer.data(),
                    ↪ &*ipuTimer.end());
381         }
382
383         std::cerr << "Copying back result\n";
384
385         auto copyback_timing_start =
                ↪ std::chrono::high_resolution_clock::now();
386         engine.run(programIds["copy_to_host"], "copy result");
387         auto copyback_timing_end =
                ↪ std::chrono::high_resolution_clock::now();
388         auto copyback_timing =
                ↪ std::chrono::duration_cast<std::chrono::nanoseconds>(
```

```
            ↪ copyback_timing_end - copyback_timing_start).count() /
            ↪ 1e3;

        // Create report
        auto report = ExperimentReportIPU(std::move(engine),
            ↪ std::move(graph));
        report.set_timing("copy", copy_timing);
        report.set_timing("execution", execution_timing);
        report.set_timing("copy_back", copyback_timing);
        report.set_timing("graph_compilation",
            ↪ timing_graph_compilation);

        if (!Config::get().model)
        {
            report.set_timing("ipu_report", ipuTimer[0] /
                ↪ device.getTarget().getTileClockFrequency());
        }

        return optional(std::move(report));
    }
}
```

Listing A.8: Prim's `PrimsBlock` codelet

```
#include <poplar/Vertex.hpp>
#include <cstddef>
#include <cstdlib>
#include <math.h>
#include <stdint.h>
#include <assert.h>
#include <cmath>
#include <float.h>
#include <limits.h>
#include <print.h>

using namespace poplar;

class PrimsBlock : public MultiVertex
{
public:
    // Our matrix in CSC form
    Input<Vector<int>> weights;
    Input<Vector<unsigned>> rows;
    Input<Vector<unsigned>> columns;

    // The current column over which we are updating dist
    // TODO row offset, setting dist to infinity
    Input<unsigned> current;
    unsigned row_offset;

    InOut<Vector<int>> dist;
    InOut<Vector<unsigned>> dist_prev;

    auto compute(unsigned workerId) -> bool
    {
        if (workerId == 0 && current >= row_offset && current -
            ↪ row_offset < dist.size())
        {
            dist[current - row_offset] = INT_MAX;
            dist_prev[current - row_offset] = UINT_MAX;
        }

        // MultiVertex safety: dist and dist_prev are aligned per row
            ↪ (32-bit word), each column only contains a row once max.
        for (size_t i = columns[current] + workerId; i <
            ↪ columns[current + 1]; i+= MultiVertex::numWorkers()) {
            unsigned row = rows[i];

```

```
42              if (weights[i] < dist[row] && dist_prev[row] != UINT_MAX) {
43                  // printf("Found: %d at dist %d\n", row + row_offset,
                        ↪ weights[i]);
44                  dist[row] = weights[i];
45                  dist_prev[row] = *current;
46              }
47          }
48
49          return true;
50      }
51  };
```

Listing A.9: Prim's `ReduceBlock` codelet

```
1  #ifndef __IPU_ARCH_VERSION__
2  #define __IPU_ARCH_VERSION__ 2
3  #endif
4
5  #include <poplar/Vertex.hpp>
6  #include <arch/gc_tile_defines.h>
7
8  #include <cstddef>
9  #include <cstdlib>
10 #include <math.h>
11 #include <stdint.h>
12 #include <assert.h>
13 #include <cmath>
14 #include <limits.h>
15 #include "print.h"
16
17 using namespace poplar;
18
19 // This class uses as SupervisorVertex to control multiple vertices,
       ↪ instead of using MultiVertex
20 // MultiVertex does not have a way to collect results from different
       ↪ threads
21 class ReduceBlock : public Vertex
22 {
23 public:
24     Input<Vector<int>> dist;
25     Input<Vector<unsigned>> dist_prev;
26
27     unsigned row_offset;
28
29     Output<int> block_dist;
30     Output<unsigned> block_dist_from;
31     Output<unsigned> block_dist_to;
32
33     Vector<int> tmp1;       // block_dist result for each thread
34     Vector<unsigned> tmp2; // block_dist_from result for each thread
35     Vector<unsigned> tmp3; // block_dist_to result for each thread
36
37     bool compute()
38     {
39         unsigned workerId = __builtin_ipu_get(CSR_W_WSR__INDEX) &
               ↪ CSR_W_WSR__CTXTID_M1__MASK;
40         unsigned workers = CTXT_WORKERS;
41
42         int best_dist = INT_MAX;
43         unsigned from = 0;
44         unsigned to = 0;
45
46         for (size_t i = workerId; i < dist.size(); i += workers)
47         {
48             if (dist[i] < best_dist)
49             {
50                 best_dist = dist[i];
51                 from = dist_prev[i];
```

```
                to = i + row_offset;
            }
        }

        tmp1[workerId] = best_dist;
        tmp2[workerId] = from;
        tmp3[workerId] = to;

        return true;
    }
};

class ReduceBlockSupervisor : public SupervisorVertex
{
public:
    Input<Vector<int>> dist;
    Input<Vector<unsigned>> dist_prev;

    unsigned row_offset;

    Output<int> block_dist;
    Output<unsigned> block_dist_from;
    Output<unsigned> block_dist_to;

    Vector<int> tmp1;      // block_dist result for each thread
    Vector<unsigned> tmp2; // block_dist_from result for each thread
    Vector<unsigned> tmp3; // block_dist_to result for each thread

    __attribute__((target("supervisor"))) void collect()
    {
        // use tmp to write back out value;
        unsigned res1 = tmp1[0] < tmp1[1] ? 0 : 1;
        unsigned res2 = tmp1[2] < tmp1[3] ? 2 : 3;
        unsigned res3 = tmp1[4] < tmp1[5] ? 4 : 5;

        res1 = tmp1[res1] < tmp1[res2] ? res1 : res2;
        res1 = tmp1[res1] < tmp1[res3] ? res1 : res3;

        *block_dist = tmp1[res1];
        *block_dist_from = tmp2[res1];
        *block_dist_to = tmp3[res1];
    }

    __attribute__((target("supervisor"))) bool compute()
    {
        __asm__ volatile(
            "setzi   $m1, __runCodelet_ReduceBlock\n"
            "runall  $m1, $m0 , 0 \n"
            "sync    %[sync_zone]\n" ::[sync_zone]
                ↪ "i"(TEXCH_SYNCZONE_LOCAL));

        collect();
        return true;
    }
};
```

Listing A.10: Prim's `GatherResult` codelet

```
#ifndef __IPU_ARCH_VERSION__
#define __IPU_ARCH_VERSION__ 2
#endif

#include <poplar/Vertex.hpp>
#include <arch/gc_tile_defines.h>

#include <cstddef>
#include <cstdlib>
#include <math.h>
```

```
11 #include <stdint.h>
12 #include <assert.h>
13 #include <cmath>
14 #include <limits.h>
15 #include "print.h"
16
17 using namespace poplar;
18
19 // This class uses as SupervisorVertex to control multiple vertices,
   ↪ instead of using MultiVertex
20 // MultiVertex does not have a way to collect results from different
   ↪ threads
21 class GatherResult : public Vertex
22 {
23 public:
24     Input<Vector<int>> block_dist;
25     Input<Vector<unsigned>> block_dist_from;
26     Input<Vector<unsigned>> block_dist_to;
27
28     Output<unsigned> current;
29     InOut<Vector<unsigned>> connection;
30
31     Output<bool> should_continue;
32
33     Vector<int> tmp1;      // block_dist result for each thread
34     Vector<unsigned> tmp2; // block_dist_from result for each thread
35     Vector<unsigned> tmp3; // block_dist_to result for each thread
36
37     bool compute()
38     {
39         unsigned workerId = __builtin_ipu_get(CSR_W_WSR__INDEX) &
               ↪ CSR_W_WSR__CTXTID_M1__MASK;
40         unsigned workers = CTXT_WORKERS;
41
42         int best_dist = INT_MAX;
43         unsigned from = 0;
44         unsigned to = 0;
45
46         for (size_t i = workerId; i < block_dist.size(); i += workers)
47         {
48             if (block_dist[i] < best_dist)
49             {
50                 best_dist = block_dist[i];
51                 from = block_dist_from[i];
52                 to = block_dist_to[i];
53             }
54         }
55
56         tmp1[workerId] = best_dist;
57         tmp2[workerId] = from;
58         tmp3[workerId] = to;
59
60         return true;
61     }
62 };
63
64 class GatherResultSupervisor : public SupervisorVertex
65 {
66 public:
67     Input<Vector<int>> block_dist;
68     Input<Vector<unsigned>> block_dist_from;
69     Input<Vector<unsigned>> block_dist_to;
70
71     Output<unsigned> current;
72     InOut<Vector<unsigned>> connection;
73
74     Output<bool> should_continue;
75
76     Vector<int> tmp1;      // block_dist result for each thread
```

```
77      Vector<unsigned> tmp2; // block_dist_from result for each thread
78      Vector<unsigned> tmp3; // block_dist_to result for each thread
79
80      __attribute__((target("supervisor"))) void collect()
81      {
82          // use tmp to write back out value;
83          unsigned res1 = tmp1[0] < tmp1[1] ? 0 : 1;
84          unsigned res2 = tmp1[2] < tmp1[3] ? 2 : 3;
85          unsigned res3 = tmp1[4] < tmp1[5] ? 4 : 5;
86
87          res1 = tmp1[res1] < tmp1[res2] ? res1 : res2;
88          res1 = tmp1[res1] < tmp1[res3] ? res1 : res3;
89
90          if (tmp1[res1] == INT_MAX)
91          {
92              *should_continue = false;
93          }
94          else
95          {
96              *current = tmp3[res1];
97              connection[tmp3[res1]] = tmp2[res1];
98              *should_continue = true;
99          }
100     }
101
102     __attribute__((target("supervisor"))) bool compute()
103     {
104         __asm__ volatile(
105             "setzi   $m1, __runCodelet_GatherResult\n"
106             "runall  $m1, $m0 , 0 \n"
107             "sync    %[sync_zone]\n" ::[sync_zone]
                    ↪ "i"(TEXCH_SYNCZONE_LOCAL));
108
109         collect();
110         return true;
111     }
112 };
```