Faculty of Science and Technology
Department of Computer Science

Investigating the effects of dynamic approximation methods on machine learning (ML) algorithms running on ML-specialized platforms

Eirik Haugen

INF-3990 Master’s Thesis in Computer Science - May 2021

/1 Abstract

This thesis discusses the application of optimizations to machine learning algorithms. In particular, we look at implementing these algorithms on specialized hardware, I.e. a Graphcore Intelligence Processing Unit, while also applying software optimizations that have been shown to improve performance of traditional workloads on general purpose CPUs. We discuss the feasibility of using these techniques when performing Matrix Factorization using Stochastic Gradient Descent on an IPU. We implement a program doing this, and show the results of changing different parameters during the running of SGD. We demonstrate that while machine learning is inherently approximate this does not mean that all approximate computation techniques are applicable, and that indeed some of these techniques require a more measurable level of approximation that is given by there being a correct answer, I.e. that the algorithm being approximated is not inherently approximate from the start. We also show that other techniques can be applied to reduce the time it takes for SGD to converge.

Contents

List of Figures

2 Introduction

Machine learning has become a prevalent branch of Computer Science. As the usage of Machine Learning techniques grows, interest in hardware specifically optimized for that type of workload will grow. When new types of hardware emerges, it is important to quantify which existing techniques work or can be adapted to these novel architectures. This thesis looks at combinations of machine learning and Approximate Computing techniques, and investigates the applicability of some techniques that have been applied to conventional programs to increase their efficiency. We then attempt to leverage these techniques to improve the convergence speed of Stochastic Gradient Descent based Matrix Factorization implemented for Graphcore Intelligence Processing Units.

In particular we look at JouleGuard, a runtime control system shown by Hoffmann[Hof15]. With JouleGuard Hoffmann builds a method for achieving computations under an energy budget by creating a runtime control system that is energy and system configuration aware, and is capable of applying approximate computing techniques to a problem. By doing this he breaks the task of achieving an energy budget target into two sub-tasks, and uses two different solutions for these problems. The first of these problems is in identifying an optimal system configuration for the system computing the problem, and the second is in finding the greatest possible accuracy possible such that the energy budget will be met. For solving the system configuration problem, Hoffman uses a machine learning technique called Value-Difference Based Exploration (VDBE), while for the latter he imagines the problem as a control theoretic one, and implements a proportional integral controller to find the required speed up of the computation. We try to map the problem of optimizing SGD-based matrix factorization running on an IPU onto both of these sub-problems, and show that only the first of these sub-problems apply in this case. We then create an experimental implementation that uses VDBE to find the best set of configurations for improving convergence speed.

We show that while this mapping of problems is not the best fit, we still produce results that show there is an increase in performance by using automated adjustment of certain parameters during runtime.

3 Dynamic Approximation and Machine Learning

3.1 Loop Perforation

Loop perforation aims to speed up computations by removing the least critical iterations of specific loops. This changes the output of the program in nearly every case tested in the original paper, however there are a wide range of computations where this can be acceptable. The loop perforation technique consists of two distinct phases. In the first phase, it tries to identify which loops of the client program respond the best to tuning. This is done by identifying every loop of the program, and trying to run it with a selective sample of inputs for each loop, and identifying which loops cause poor results, such as very large effects on the accuracy of the output, little decrease in run-time, crashes, or even increases. When these have been filtered out, it is left with a set of loops that are referred to as "tunable".
After the tunable loops of the client program have been identified, the second phase aims to find the optimal amount of tuning of these loops that produce the highest speed-up with the least accuracy sacrificed.
Sidiroglou et. al. demonstrate that this technique typically only changes the result of the application by 10%, while speeding up execution by a factor of 2. We mention Loop Perforation in this thesis to demonstrate the power of Approximate computing on traditional, sequential workloads.

3.2 JouleGuard

JouleGuard, described in [Hof15], is an energy-aware system for optimizing the performance of a program while operating under a power budget. The system breaks this problem into two sub-problems: that of finding an optimal system configuration delivering the best possible performance to power consumption ratio for a specific problem, and then speeding up the running of the program further by leveraging approximate computing techniques.

3.2.1 Value-Difference Based Exploration

When searching for an optimal configuration of the client system, JouleGuard applies a machine learning technique called Value-Difference Based Exploration (VDBE). This technique involves defining a set of system configurations Sys, and creating an estimate of a given configurations power usage and performance using exponentially weighted moving averages. The performance estimate of a given configuration in timestep t is denoted as $\hat{p}_{sys}(t)$, and the power estimate as $\hat{r}_{sys}(t)$. JouleGuard then lets the client program execute for timestep t, and gets the real measured values for the performance and power, denoted as $\bar{p}_{sys}(t)$ and $\bar{r}_{sys}(t)$ respectively. The Estimates are then updated using Eq. 3.1. JouleGuard uses $\alpha = .85$[Hof15]. The key feature of

$$\begin{aligned}\hat{r}_{sys}(t) &= (1-\alpha)\times\hat{r}_{sys}(t-1)+\alpha\times\bar{r}_{sys}(t)\\ \hat{t}_{sys}(t) &= (1-\alpha)\times\hat{p}_{sys}(t-1)+\alpha\times\bar{p}_{sys}(t)\end{aligned} \tag{3.1}$$

3.1: Estimate update in VDBE, from [Hof15]

VDBE is in deciding when to try other system configurations, I.e. explore the configuration space, and when to use the estimated best configuration. The way VDBE does this decision is in calculating the value $\epsilon(t)$ based on the relationship between the estimated and measured values. $\epsilon(t)$ is given by Eq. 3.2.

$\epsilon(t)$ is then used to decide whether to explore the configuration space by generating a random number r such that $0 < r < 1$. If $r < \epsilon(t)$, we select the system configuration that delivers the best estimated performance, otherwise we pick a random configuration. Eq. 3.2 has the properties that the closer our estimates are to the measured values, the closer $\epsilon(t)$ is to 1, with $\epsilon(t) = 1$ when $\hat{r}_{conf} = \bar{r}_{conf}$ and $\hat{p}_{conf} = \bar{p}_{conf}$. This gives the desirable effect of never causing configuration space exploration when the estimates are correct, instead sticking with the best configuration. So the system is stable when the correct configuration is found, but the system might be vulnerable to unforeseeable events that cause the performance or power usage to change for some or all

$$\begin{aligned} x(e) &= e^{\frac{-|\frac{\bar{r}sys(e)}{\bar{t}sys(e)} - \frac{\hat{r}sys(e)}{\hat{t}sys(e)}|}{5}} \\ p(e) &= \frac{1 - x(e)}{1 + x(e)} \\ \epsilon(e) &= \frac{1}{|Sys|} \times p(e) + (1 - \frac{1}{|Sys|}) \times \epsilon(e-1) \end{aligned} \tag{3.2}$$

3.2: Calculate ϵ update in VDBE, from [Hof15]

configurations. In an event like this, the values of the estimates will diverge from the measured values, which in turn will lower $\epsilon(t)$, and cause configuration space exploration to become more likely. Via this technique JouleGuard achieves a high degree of efficiency even when circumstances change during runtime[Hof15].

3.2.2 Proportional Integral Controller

While finding the optimal configuration of the system can already mean that a computation will fall under an energy budget, this is not sufficient to guarantee that it will. In cases where the computation will not fall under the energy budget in the optimal configuration, JouleGuard can find how much the program execution must be sped up such that the budget will be kept. JouleGuard now has a target speed, and a way to alter the performance of the computation; approximate computing. The runtime works with any approximate computing technique which can order configurations, i.e. it can know that configuration A delivers more accuracy than configuration B.

With this knowledge and this way of altering the speed of computation, JouleGuard can model this issue as a control theoretic problem. It does this by creating using a Proportional Integral (PI) Controller to minimize the error between the required speed-up and the measured speed-up. This means that this controller attempts to find the amount of approximation that delivers the needed speed up and no more.

This PI controller also includes an adaptive pole which defines how much inaccuracy the controller can tolerate while still providing an energy guarantee, I.e this pole defines how inaccurate the estimates delivered by VDBE can be while the budget is kept. This pole is then defined by the measured inaccuracies in the estimates such that when the measured inaccuracy of the estimates is high, the controller is more conservative in how much it acts, while when the VDBE portion is confident in its estimates the PI controller can be more aggressive in its actions.

3.3 Matrix Factorization

Matrix factorization is a concept wherein you take a matrix of values and decompose it into two smaller matrices. If we have original matrix $M \in \mathbb{R}^{m \times n}$, the goal is to Produce two matrices $W \in \mathbb{R}^{m \times k}$ and $H \in \mathbb{R}^{n \times k}$, where k is much smaller than both m and n, and $M \approx WH$.
Already this results in a savings in storage space required to store the information contained in M, as $(m \times k) + (n \times k) < m \times n$.

Another use for matrix factorization is to use it to perform Matrix Completion. If we do not know every value in M, I.e. M is a sparse matrix, if we can still factor M into W and H, we can use the product WH as an estimate of what a complete M would look like. In this thesis, we look at using Matrix Factorization to perform Matrix Completion in the context of recommender systems. The power of matrix Factorization in this context is that what the Matrix Factorization algorithm is doing is detecting a set of attributes that different items in the M share in different ratios, and building W and H such that each row or column of the smaller matrices have these sets of attributes in ratios that can be used to build the original matrix M. This translates very well to situations wherein a system is trying to find qualities of a certain product, and then find out what qualities different users like. In this thesis we will look at a case where M is a matrix of user-given ratings of movies. In a case like this, for a matrix factorization-powered recommender system, H might be a matrix where each column represents a set fo features or qualities the movie has, and W a matrix of users where each row is a set of qualities that user enjoys.
What we end up with then is a matrix W of users and matrix H of movies, where if we take two vectors W_i and H_j, we can find the value of $M_{i,j}$ by calculating the dot product of the two vectors. The dot product is given by

$$\langle W_i, H_j \rangle = \sum_{n \in W_i, H_j} W_{i,n} \times H_{j,n}$$

This means that after Matrix Factorization, we have H and W such that we can build M, and by doing so create a predicted rating for what a user would give a movie, based on the qualities the algorithm has assigned the two.

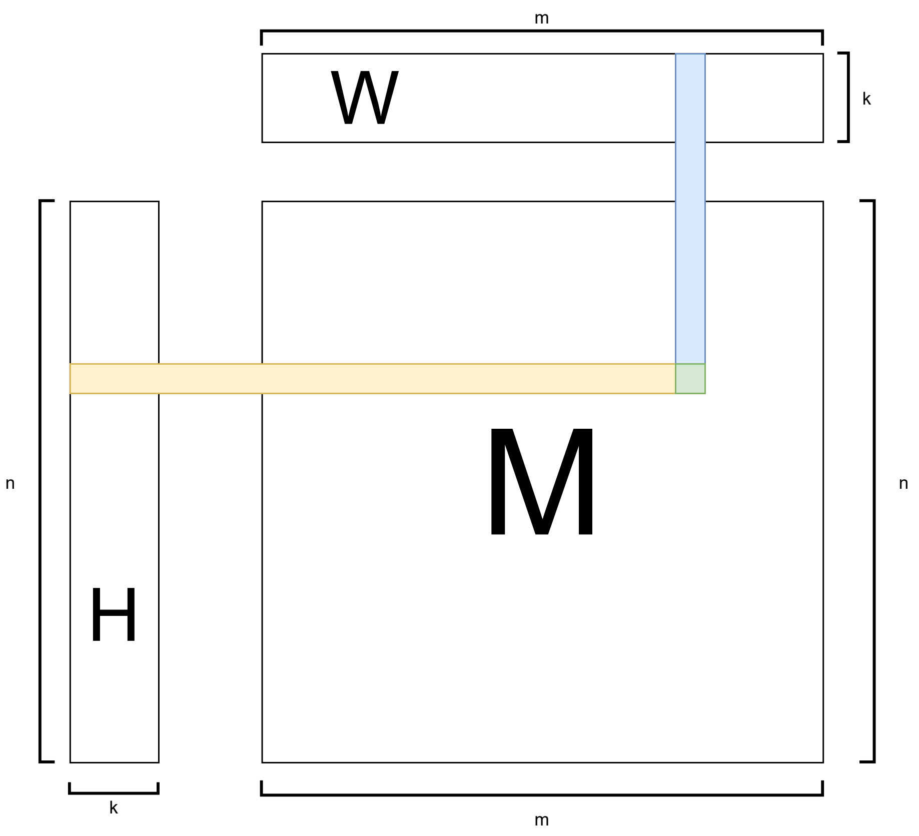

Figure 3.1: A visualization of how M is constructed as a product of W and H

3.3.1 Stochastic Gradient Descent

Gradient descent is a machine learning technique wherein one tries to find the minimum value of a function by computing the output for a given input value, then shifting the input value slightly, and seeing how the output moved. If the output grew smaller, one continues to shift the input value until one finds the minimum output value. To be certain that one has found the true minimum output of the function, however, one must examine every possible input value of the function.
This quickly becomes impractical for most functions, and for these functions we turn to Stochastic gradient descent instead. For SGD, we perform a gradient descent at a random set of starting inputs. This can lead to the output being a local minimum, i.e. an input to the function such that any small change in input causes the output value to rise, but there exists some other input that causes a lesser output. One therefore must choose how many inputs one tests to lower this risk by deciding how tolerant one's purpose is to local minima.
When we refer to a small change in input, this is a value that is decided beforehand, as it is important to the proper behaviour of the technique, while the proper value is dependent on the specific function being minimized. Typically this value is referred to as the step size of the Stochastic Gradient Descent, and we will denote it with α.
This technique is often applied to Matrix Factorization problems, as it allows us to incrementally improve the fit of H and W such that they produce a good estimate of M. There is no formula for matrix factorization, and as such we need to find the two factors via trial and error. With SGD, we start with H and W as matrices of the correct dimensions, with random initial values.

When performing a stochastic gradient descent, it is possible to make the output values of one is producing fit the available data too well, in such a way that it gets further away from a real fit to all expected data. To combat this, we introduce a regularization parameter, which we will refer to as λ. By setting lambda to an appropriate value, we can control how strong the fit of the output will be to the input matrix. This also means that when performing SGD, we are not waiting for the accuracy to reach a known threshold, but instead for it to stop changing, and stabilize around a value where each update is not producing a change, or very little change.
An SGD update is defined in Eq. 3.3. For each epoch that the computation is running, we select a certain amount of random values in M, and update the corresponding values in H and W.

$$\begin{aligned} W_i &\leftarrow W_i - \alpha((M_{i,j} - (W_i H_j)Hj) - \lambda W_i) \\ H_i &\leftarrow H_i - \alpha((M_{i,j} - (H_i W_j)Wj) - \lambda H_i) \end{aligned} \tag{3.3}$$

4 Graphcore Intelligence Processing Unit

The IPU is designed both on a hardware and software level to be well suited to machine learning. To do this, the design of the IPU aims at providing strong performance in highly parallel tasks that represent diverse workloads. The goal here is to be able to compute on a large quantity of data where it may is desirable to perform different operations on different segments of the data. This type of parallel execution is often referred to as Multiple Instruction, Multiple Data (MIMD) Parallelism, for the fact that it performs multiple different instructions on multiple pieces of data in parallel[Jia+19].

4.1 Hardware

A traditional CPU is typically optimized to execute code that is not parallel or has relatively low levels of parallelism. The design of CPUs involves a lot of optimizations per core on a relatively low core count. Modern CPUs also typically include several levels of memory. A common configuration for a Multi-core CPU is for each core to have a small amount of memory referred to as the L1 cache, which is comprised of the fastest memory technology available, located close to the core on chip. After the L1 cache, CPUs have one or more levels of cache that is shared between all cores, before hitting the main memory of the

CPU. Generally each level of cache has more storage, but is slower and further from the cores[Hig90]. Current day CPU caches are typically implemented using Static Random Access Memory (SRAM). This type of memory is utilized for caches for its superior read/write speeds as compared to any other feasible technology for the purpose[Kor+18]. However, SRAM is quite costly compared to other memory techniques, as well as occupying more physical space per byte than other technologies, such as DRAM[KRS00]. These complex cache layouts reduce the latency of memory accesses by selecting data that is likely to be used in the future into faster, closer memory.

The IPU naturally invites comparisons to Graphical Processing Units (GPU). This is because they are both processor architectures that aim at optimizing for workloads with high degrees of parallelism. Their intent differs in the type of parallelism that is offered by them. A GPU is optimized for Single Instruction, Multiple Data (SIMD) parallelism, where a single set of instructions is executed on a large amount of data in parallel. A typical example of this is in, as the name suggests, processing of graphical workloads[Gui13]. GPUs achieve their high degree of parallelism by having a large number of basic processor cores that are divided into groups of threads called warps. Each warp is then scheduled, managed, and executed together. Each thread in a warp can work on different data, but all these threads can only execute the same instruction in parallel. Code within a warp can have conditional branching, such that certain threads will branch to different instructions from other threads in the warp, but in cases where threads diverge in instructions, the warp can still only execute one instruction at a time, such that the GPU will execute the different branches in series, freezing the threads in the warp that are not in the current branch[Gui13]. While modern GPUs have a more complex memory layout than historical GPUs had, they share the key philosophy behind hiding memory access latency by instantiating a large number of threads such that when a warp is waiting for memory, a different warp can be context-changed into while the first warp is blocked.

Similar to a GPU, a IPU offers a large core count. The GC2 (Mk 1) IPU has 1116 cores, where each core has 256 KiB of memory, while the GC200 (Mk 2) IPU has 1472 cores where each core has 642 KiB of memory[Graa]. This highlights the core difference in design philosophy between a IPU and a GPU. In an IPU, each core is more complex than those of a GPU, such that each of them is capable of executing a program independently, such that in theory each of the 1472 cores of a GC200 IPU can be executing a unique program on unique data in parallel without incurring a penalty over all cores executing the same program. Graphcore refers to the combination of a processing core on the IPU and the local memory of that core as a *tile*. When the IPU processes data, all instructions and data that is to be used by a tile must reside on the local memory of that tile. Like GPUs, each IPU tile also benefits from having more threads assigned to it than can run at once, such that it can switch out a thread in the event that it is blocked, thus hiding memory latency. The memory on the tiles is implemented as SRAM, and because of this get similar R/W performance to a CPU cache[Jia+19]

As we have discussed above, the IPU then delivers a very fine grained MIMD parallelism, where each parallel thread can be doing a completely separate thing. However, this is not very useful in and of itself, as with the features we have described, making an IPU as a single device perform a task presents several problems. Each tile has relatively little memory, and manually partitioning memory such that each part of it fits within a certain tile, and then making sure the instructions that pair to that data is also sent to the correct tile. Partially this problem is solved in software with the Poplar SDK, but these solutions are made feasible via the Interconnect Architecture of the IPU system. This interconnect is a data bus that connects each tile of the IPU to every other tile, providing a high data bandwidth with low latency, such that it is feasible to load data onto the IPU without concern for locality, and the use the interconnect to distribute the data to the tiles that need it, before executing programs on the tiles. In addition to this, each IPU contains a module called an IPU link, which effectively merge the IPU interconnects of one or more IPUs, to create a multi-IPU system that transparently appears as a single IPU to a programmer. The interconnect does have latency penalties depending on distance between the tiles, with additional penalties for crossing IPU link boundaries, thought the benefit of such multi-IPU systems presents in a potentially much higher core count, with a corresponding much higher aggregate memory[Jia+19].

An important aspect of the philosophy of the design of the IPU is the Bulk Synchronous Parallel model for generalizing parallelization of workloads. In the paper introducing the BSP model, it is described as a model not of hardware, or software, but a model that can inform the design of both[Val90]. According to BSP, a device or framework that is suited for parallelized workloads should be designed such that it has three core attributes. The first of these is that it should have some amount of components that can perform computation, and some amount of components that have memory. In the IPU architecture these attributes are covered by the tiles, which have both a processor and memory. Second, a BSP capable system should have some method of routing data between sets of processing and/or memory components, which in the IPU is covered by the interconnect. Finally the BSP model requires that the device should have some method of synchronizing the processing components. In the IPU, the synchronization is also dealt with via the interconnect.

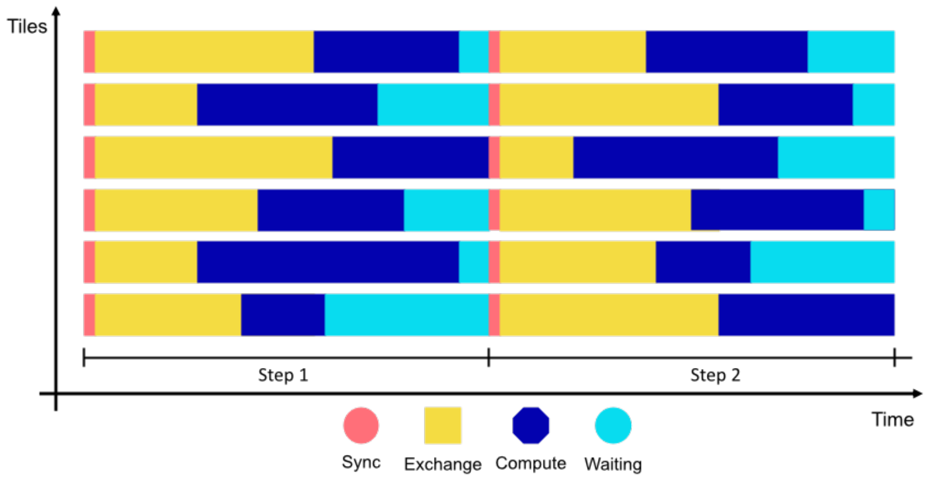

Figure 4.1: Activity of tiles during two steps of graph execution, from [Graa]

4.2 Software

While the BSP model underpinning the IPU is seen in the design of the hardware, it also heavily informs the design of the software running on the IPU as well as the programming model used by users of the IPU. This means that a program running on an IPU is segmented into supersteps, such that each step consists of a communication stage where each tile makes sure that it has the data needed for its instructions on its local memory, followed by an execution step where each tile performs its computations, before waiting for all tiles to finish their computations in a final barrier synchronization step. A major benefit of this model is in automatic memory management. As we will see later, the IPU programmer specifies how segmented a set of data is, that is how many tiles it is distributed across, and the first step in the superstep will ensure that the data resides on the tiles that require it automatically. When programming an IPU, our program defines how the set of tensors on the graph should look like, then either use pre-defined Vertex-code from the SDK, or write our own vertices. When we have the code for the vertices and the tensors ready, we define the connections between sections of the tensors and copies of the vertices, and map these vertices onto tiles of the IPU. Preferably, we want the parts of a tensor that are used by a certain vertex to reside on the same tile as that vertex, since this provides the fastest possible R/W for the tile. In addition to this, we also want to maximize hardware utilization, which in this case means mapping all tensors and vertices evenly over all available tiles.

When the IPU executes the graph, the graph is sectioned into steps. At the start of each step, all the tiles synchronize with each other. Once synced, the

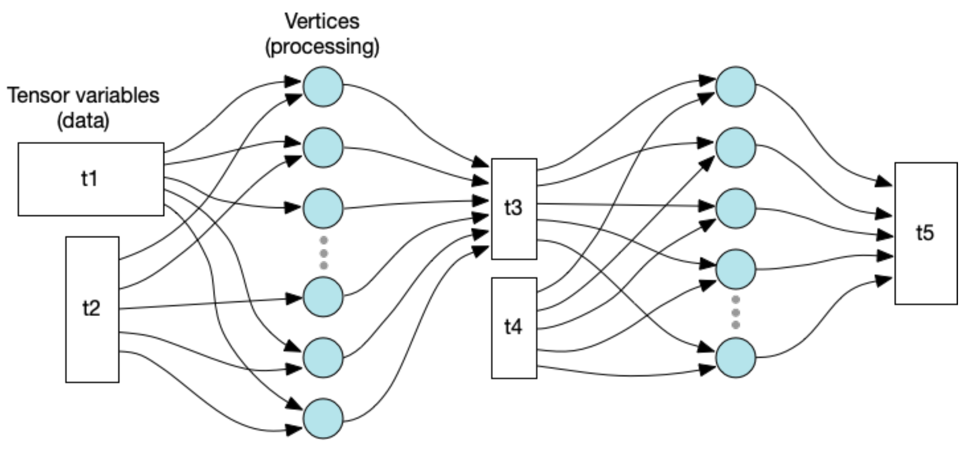

Figure 4.2: A Poplar program as a Graph, from [Graa]

tiles get the data they need for their computation from other tiles as needed, then perform their computation, before finally waiting for the synchronization at the next step, as illustrated in Fig. 4.1.

4.3 Utilizing the IPU

Writing code for the IPU is done using the Graphcore-supplied SDK Poplar. In Poplar, the program to be run on the IPU is referred to as a *graph* When writing code in Poplar, variables are defined as *tensors*, while sets of instructions to be run on one or more tiles are called *vertices*. Thus, when we build a graph for an IPU we first define some set of tensors which will hold input data for the full IPU program. After this, one or more tensors can be define variables to hold data during execution of the graph, and for output data.

Once a set of tensors is defined, we can define a set of vertices to be included in the program. These can be written as codelets for the graph we are building, or gotten from a number of pre-defined vertices included in the SDK. When creating a vertex, we specify the tensors, or a subset of a tensor, which represents the input and output data of that vertex. In this way, we can define many copies of the same instructions with different data, and thus achieve a programming model similar to the SIMD parallelism of a GPU, while still having the flexibility of an IPU. We can for instance have one step in the graph where the execution resembles a SIMD program, and then flow into a second step where many different sets of instructions are performed on different sections of the data.

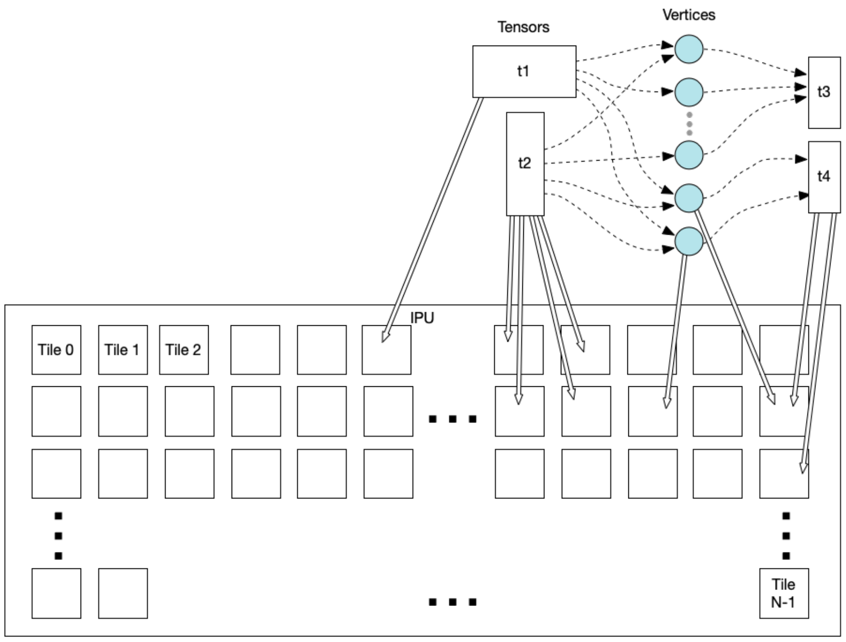

Figure 4.3: Placement of vertices and tensors on tiles of an IPU, from [Graa]

If we can fit our entire model, I.e. the set of instructions and all data, into the aggregate memory of the tiles, we can generate this graph, and execute it fully on the IPU and be done with our computation. If we cannot do this, however, we must define use the input and possibly the output tensors of the graph to update the data that is being evaluated between executions of the graph.

/5 Design

We implement a Stochastic Gradient Descent-based Matrix Factorization program that runs using an IPU. In this program, we implement a Governor that attempts to speed up the program by selecting the best parameters for the algorithm. This program is specifically written for using the MovieLens dataset, as this is a large dataset of real world data, and should represent a realistic workload[HK15].

5.1 Structure

We separate the program into three structures as illustrated in Fig. 5.1.

5.1.1 Main

The "Main" structure builds the IPU program and calculates the accuracy of the computations performed. For measuring accuracy we use Root Mean Square Error (RMSE) given by $\sqrt{\frac{\sum_{i,j \in M,W,H}(M_{i,j} - \langle W_i, H_j \rangle)^2}{N}}$. In addition to this, the main structure generates the random subset of Ω, Ω_{sub}, though the size of it is decided by the Governor.

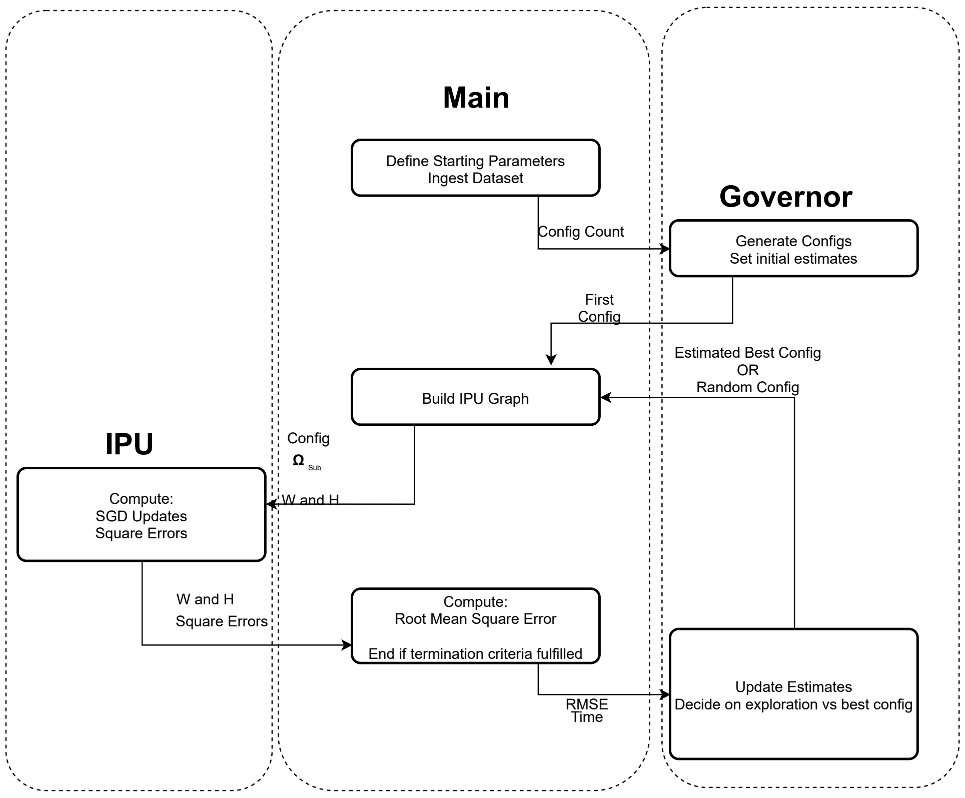

Figure 5.1: Overview of the program flow

5.1.2 Governor

The Governor holds and modifies estimates of configurations, and selects the best one. When the Governor is instantiated at the start of the program, it is given an amount of configurations to generate. We will refer to the generated set of configurations as $Conf$. Each configuration in $Conf$ consists of three parameters. α, the step size, samples per epoch, defining how many values from Ω to send to the IPU per epoch, and finally iterations per epoch, defining how many updates the IPU should perform on those values in a single epoch. When these configurations are generated, each of those parameters are given a random value.

The technique for selection a configuration is a modification of the VDBE method used in JouleGuards[Hof15] for selecting the best system configuration. We refer to the measured time taken to execute an epoch e using configuration $conf$ as $\bar{t}(e)_{conf}$ and the change in RMSE for that epoch $\bar{r}(e)_{conf}$. The estimate for the time taken for e is given as $\hat{t}(e)_{conf}$, and the estimated change in RMSE is given with $\hat{r}(e)_{conf}$. After a new Epoch is run, the Governor updates its estimates using eq. 3.1, and then computes $\epsilon(e)$ using eq. 3.2. Note here that we have renamed the weight constant, as in JouleGuard they denote this with α, which we are using to refer to the learning rate of the matrix Factorization. We use $weight = .85$, as in the JouleGuard paper.

$$\begin{aligned}\hat{r}_{conf}(e) &= (1 - weight) \times \hat{r}_{conf}(e-1) + weight \times \bar{r}_{conf}(e) \\ \hat{t}_{conf}(e) &= (1 - weight) \times \hat{t}_{conf}(e-1) + weight \times \bar{t}_{conf}(e)\end{aligned} \tag{5.1}$$

$$\begin{aligned}x(e) &= e^{\frac{-|\frac{\hat{r}_{conf}(e)}{\hat{t}_{conf}(e)} - \frac{\hat{r}_{conf}(e)}{\hat{t}_{conf}(e)}|}{5}} \\ p(e) &= \frac{1 - x(e)}{1 + x(e)} \\ \epsilon(e) &= \frac{1}{|Conf|} \times p(e) + (1 - \frac{1}{|Conf|}) \times \epsilon(e-1)\end{aligned} \tag{5.2}$$

Once $\epsilon(e)$ is calculated, we generate a random number r. If $r < \epsilon(e)$, we select a random configuration, and if not we select the configuration with the best estimated performance, as measured by $\frac{\hat{r}_{conf}(e)}{\hat{t}_{conf}(e)}$.

5.1.3 IPU Vertices

We have two different IPU vertices. The first performs an SGD update on a value of H or W, while the second measures the square error of a set of values. The SGD updates are distributed across the tiles of the IPU, and then the values of

W and H are updated. After this is done, the square error of the updated values is calculated, and these results are sent to the Main portion of the program. This is also the portion of the program that accepts the parameters controlled by the governor. The α parameters decides the step size of the SGD update, as can be seen in Eq. 3.3. The Iterations Per epoch parameters decides how many times the SGD update is performed on each of the samples from M that have been selected, while the samples per epoch decides how many samples are selected. These three parameters then combine to control how fast the selected samples move towards the final value in an epoch, as well as how many samples, which in turn should give the governor control over how fast the entire output is moving towards the desired value.

/6 Implementation

6.1 Main

The main portion of the program is written in C++, and includes all usage of the Poplar SDK in this implementation. A simplified pseudocode for this section of the program is shown below

```
$\Omega \leftarrow disk$
$W \leftarrow$ random values
$H \leftarrow$ random values
$conf \leftarrow Governor.get\_first\_config()$
**while** iterations $< 300$ **do**
    $\Omega_{sub} \leftarrow conf_{SPE}$ from $\Omega$
    Build tensors and datastreams for H, W and $\Omega_{sub}$
    $Update\_W \leftarrow$ Compute Set of SGD Update Vertices for updating W
    $Update\_H \leftarrow$ Compute Set of SGD Update Vertices for updating H
    $Calc\_SE \leftarrow$ Compute Set of Square Error Vertices
    **function** GRAPH PROGRAM
        Copy $\Omega_{sub}$ to device
        Copy W to device
        Copy H to device
        **function** REPEAT($conf_{IPE}$)
            Run $Update\_W$
            Run $Update\_H$
        **end function**
```

```
            Run $Calc\_SE$
            Copy W to host
            Copy H to host
            Copy Square Errors to host
        end function
        $H, W, SquareErrors \leftarrow$ run Graph Program on IPU
        $RMSE \leftarrow \sqrt{\frac{SquareErrors}{Conf_{SPE}}}$
        $conf \leftarrow Governor.get\_new\_config(RMSE, time)$
    end while
```

This portion starts by reading all data from the dataset into a C++ standard library vector of triples. The ratings in the MovieLens dataset are arranged such that the userId are contiguous, that is the ID range $[1..max >$ is the set of all integers between 0 and max, so this presents no problem when deciding one of the indexes of a rating in M, however, the Movie ids in the dataset do not have this property, and so a mapping from movie id to index in m is needed. We create this mapping by making the index of each movie the index it has in a sorted list of all movie ids.

Once the dataset has been read and preprocessed, we generate two vectors, one of size $rows * k$ for W and another of size $columns * k$ for H, and fill these with random values. Once this is done, we enter the loop that runs until the program is done. In this loop we take a generate Ω_{sub}, the random subset of Ω to run the SGD updates on for this iteration. Here we define the datastreams that will copy data to and from the IPU to the host device, and also the tensors we will use to hold data during execution of the graph. Once this is done, we build the three Compute Sets of Vertices that will be executed in the graph. The first two of these are both sets of SGD Update Vertices, with one having W as the "candidate" tensor, and H the "opposite", and the other Compute Set having the roles of W and H reversed. The third Compute set is made up of Square Error Vertices.

Once all Streams, tensors and Compute sets are defined, we build the program to be executed on the IPU. This program copies all needed data, including the full values of H and W, and then runs the two SGD Update Compute Sets IPE, times, where the value of IPE is gotten from a config received from the governor. It then runs the last Compute Set, before copying the newly updated H and W, as well as all Square Errors back to the host.

Once the graph has been executed on the IPU, we calculate the Root of the mean of the Square Errors that the IPU calculated, and send this value along with the time this iteration took to run to the governor, requesting it give us a configuration.

6.2 Governor

The Governor is also written in C++, and runs on the host CPU like the main portion of the program. It is implemented as a class, containing the set of randomly generated configurations $Conf$ as a member, and having two methods associated with it. The first of these methods is called at the start of the program, to get the first config. This method simply returns the first of the randomly generated configurations. The second method is what updates the estimates for the previously run config, then calculates $\epsilon(e)$ using Eq. 5.2 and uses it to decide which configuration to return. If $\epsilon(e) \geq r$ where r is a random number between 1 and 0, we pick the configuration with the best estimated change in rmse over time, otherwise we pick a random configuration from $Conf$.

6.3 IPU Vertices

6.3.1 SGD Update Vertex

The first Vertex is the SGD Update Vertex, which we describe in pseudocode as such:

```
**for** i in indexes of Cand, Opposite **do**
    $diff \leftarrow \Omega_{(x,y)} - (Cand[i] \times Opposite[i])$
    $regularized \leftarrow \lambda * Cand[i]$
    $Cand[i] \leftarrow Cand[i] + \alpha((diff - Opposite[i]) - regularized)$
**end for**
```

Where $\Omega_{(x,y)}$ denotes a single rating from Ω (and Ω_{sub}), Cand represents the vector gotten from taking either the x-th vector from W or the y-th vector of H, such that the Product of these two vectors should approximate $\Omega_{(x,y)}$. α denotes the learning rate, and λ the regularization parameter, as described in Sec. 3.3.1.

6.3.2 Square Error Vertex

The Second Vertex is the Square Error Vertex, which takes two vectors that come together to approximate a value in Ω_{sub}, and calculates the dot product of the two vectors, and squares the difference between this product and the value from Ω_{sub}.

7 Evaluation

7.1 Specifications

The program included with this thesis was written in c++ using the poplar SDK created by Graphcore, and during testing has been compiled using g++ 7.50 on a machine running Ubuntu 18.04.5 LTS. The tests have been run on a Graphcore POD64 system, using one Graphcore M2000 IPU.

7.2 Experimental Parameters

When performing experiments, we investigate how the parameters controlled by the governor affect the results of the computation. We let the program run for 300 epochs, so that differences in sample size will not cause an early termination of some runs. Fig. 7.1 shows how the RMSE changes over the epochs when all the parameters are set at the beginning. In these runs, $\alpha = 0.012$, Iterations per epoch is 1, and samples per epoch is set to 5000. We see that the RMSE falls rapidly for the first 50 epochs, before slowly levelling out, and seeming to get stable somewhere in the 150-200 epoch range. For tests where the governor controls one or more parameters, the parameters are set to fall within certain bounds.

Param	min	max
α	0,0005	0,05
IPE	1	20
SPE	500	5000

The bounds for α were decided by observing that any value above 0,05 caused rapid divergence, and 0,0005 would result in very slow convergence. For the iterations per epoch, a value below one would result in an epoch where nothing was done barring copying data back and forth to the device, while we set the upper bound to an arbitrary value we judged sufficiently greater than one as to give us usable data. This is the same with the lower bound for samples per epoch, where 500 was judged to be small enough to demonstrate the impact of the variable, while the upper bound was selected due to the nature of the dataset in combination with Poplar. When creating a vertex for a single superstep of the IPU execution, Poplar enforces the property that a value can only be read or written from. This means that if we have two entries in Ω that share one value in their coordinate, we cannot evaluate them in the same epoch. For the MovieLens dataset, we found that a SPE much over 5000 would result in the random selection of Ω_{sub} failing to find enough usable samples frequently.

For the tests where the governor controls none of the parameters, they are set as such $\alpha = 0.012, IPE = 1, SPE = 5000$.

7.3 Results

Note that in all graphs, the Measured RMSE is the RMSE of the Ω_{sub} for that epoch only. We will also show the true RMSE as calculated for every value in Ω at epoch 300. For the time graphs, we show the measured time it took to execute a given epoch, as well as the average time of all epochs. Observing the RMSE at epoch 300 in Fig. 7.1, it appears to have held stable at around 160 since around epoch 150. Calculating the true RMSE of the output W and H of the program, as they were after epoch 300, we see that the RMSE of the full dataset is indeed 160,5. We also see that the time per epoch is relatively constant.

In Fig. 7.3, we see that it appears the RMSE of each Ω_{sub} appears to drop much faster, and we reach a low point before epoch 100, however we see that the graph is significantly more noisy, and it is difficult to judge when a true stable point is reached. However, at the end of epoch 300, the true RMSE of Ω to H and W was 164,3, which is very close to the value without the governor. The time per epoch is unchanged, since the amount of computation per epoch is unchanged.

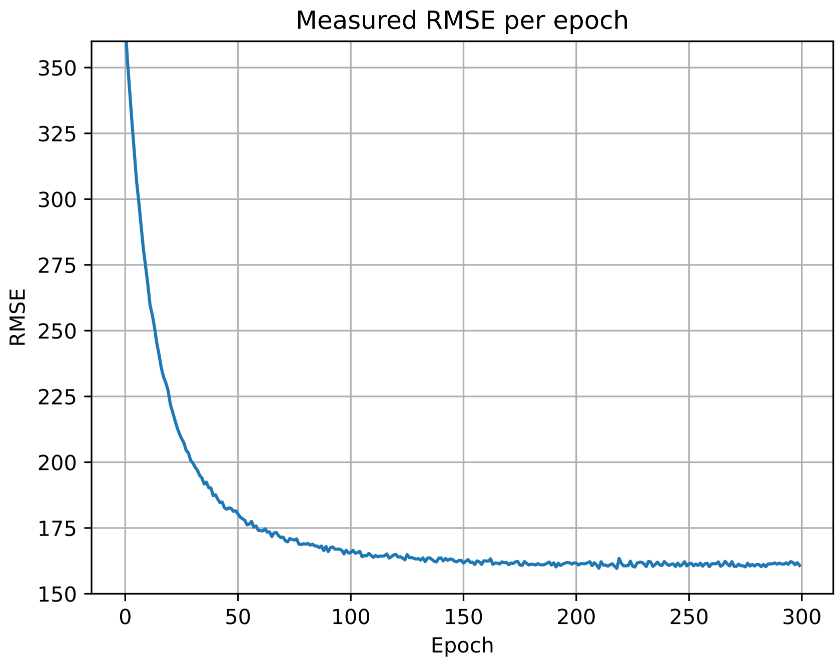

Figure 7.1: Change of RMSE with the governor controlling no parameters

Letting the governor control the amount of iterations per epoch, as seen in Fig. 7.5, there is a significant change in the way our measured RMSE changes, as it appears to hit a value around the expected 160 nearly immediately. This is because every iteration performs a SGD update, and so we expect the value of Ω_{sub} to be closer to our computed values at the end of our epoch. It is important here to note that the RMSE of Ω_{sub} is not necessarily an accurate representation of the RMSE for Ω, as it only shows how close a given row or column in H or W is to a single point in our desired output. We also see that the RMSE appears to increase again, and indeed after epoch 300 the true RMSE of Ω is 178,8. We will discuss possible reasons for this in the Discussion chapter. Note that the time per epoch in Fig. 7.6 is less stable in this case, but not much higher.

In Fig. 7.7, we see that changing the samples per epoch alone does not change the appearance of the graph much, as with alpha it appear to exhibit similar behaviour to the control, albeit with a noisier graph. However, in Fig. 7.8, we see that the time per epoch has grown noisier, but with a lower average time. We also see that the RMSE of Ω is 161,3.

In Fig. 7.9, we see that when we let the governor control all three parameters, the graph very much resembles the one in Fig. 7.5, where only the IPE was governed. This is due to how strong the effect of the IPE change is to the

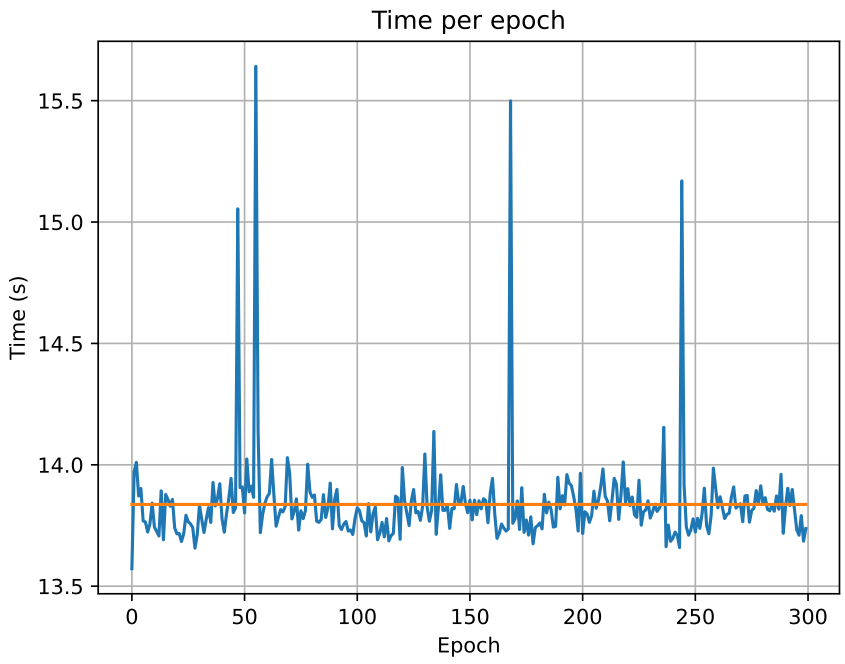

Figure 7.2: Time per epoch with the governor controlling no parameters

measured RMSE, and effects of other changes are drowned out by it. In Figs. 7.11, 7.12, and 7.13, we show how measured RMSE is affected by letting the governor all combinations of two of the three parameters. Again we see in Figs. 7.11 and 7.13 that where the governor controls IPE, we see a much faster descent to a value around 175, however again the values appear to be higher, and indeed the full RMSE after epoch 300 is 179,4 and 185 respectively. For the test where the governor controls α and SPE, shown in Fig. 7.12, we see behaviour quite similar to Fig. 7.3, where only α was governed.

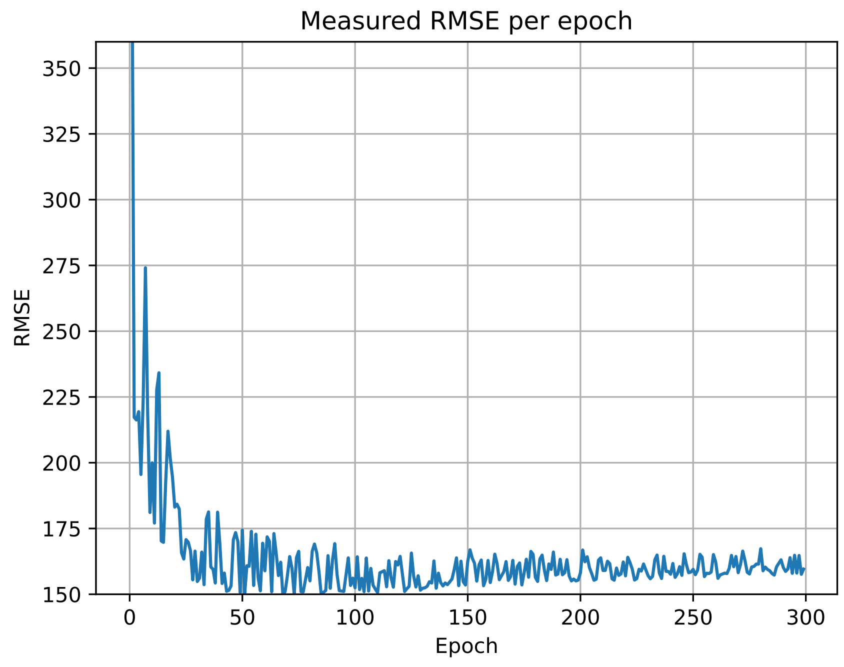

Figure 7.3: Change of RMSE with the governor controlling only α

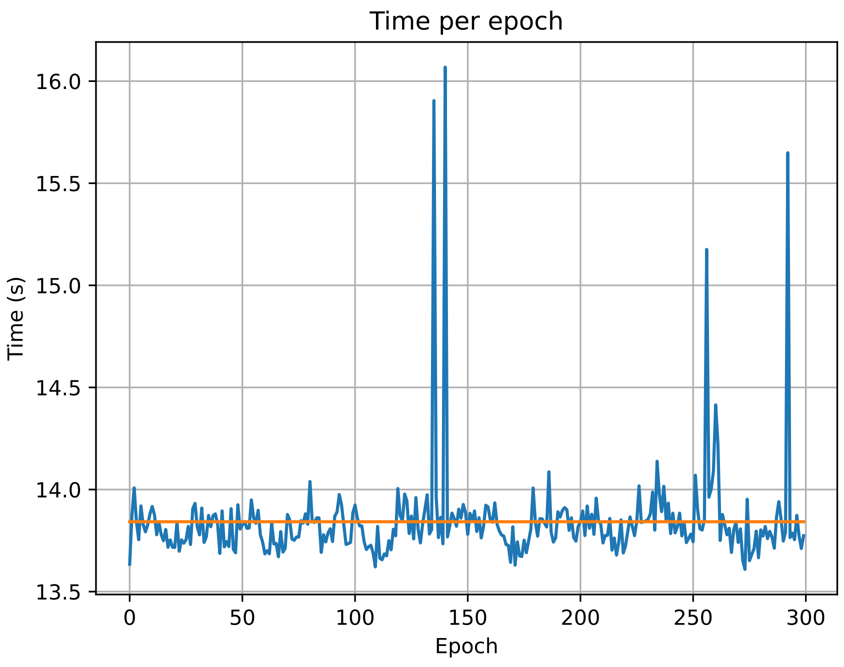

Figure 7.4: Time per epoch with the governor controlling only α

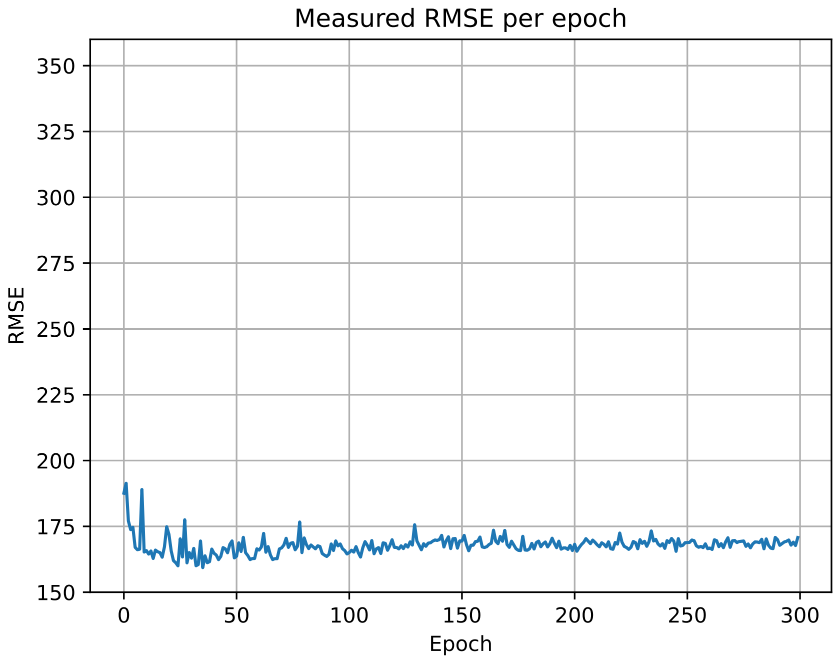

Figure 7.5: Change of RMSE with the governor controlling only iterations per epoch

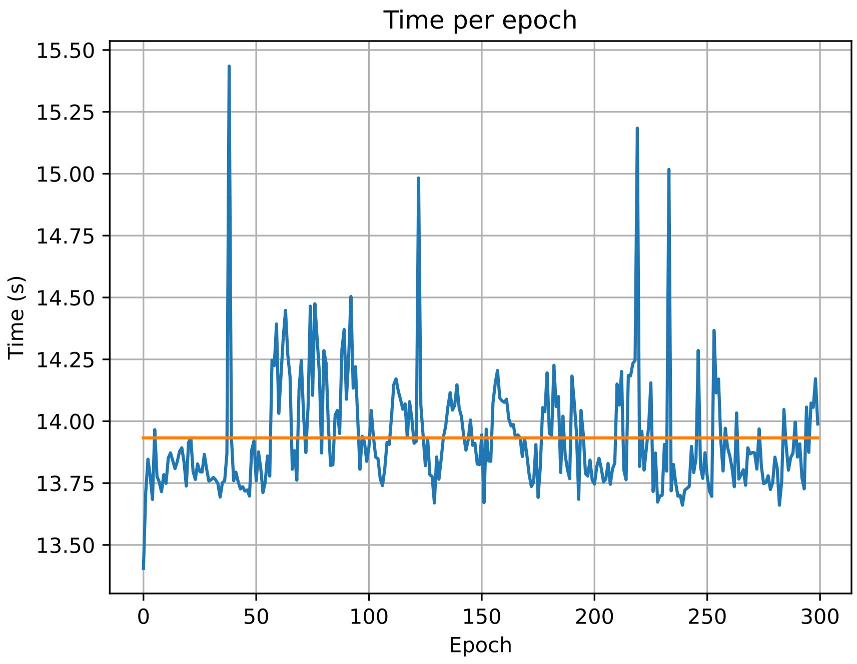

Figure 7.6: Time per epoch with the governor controlling only iterations per epoch

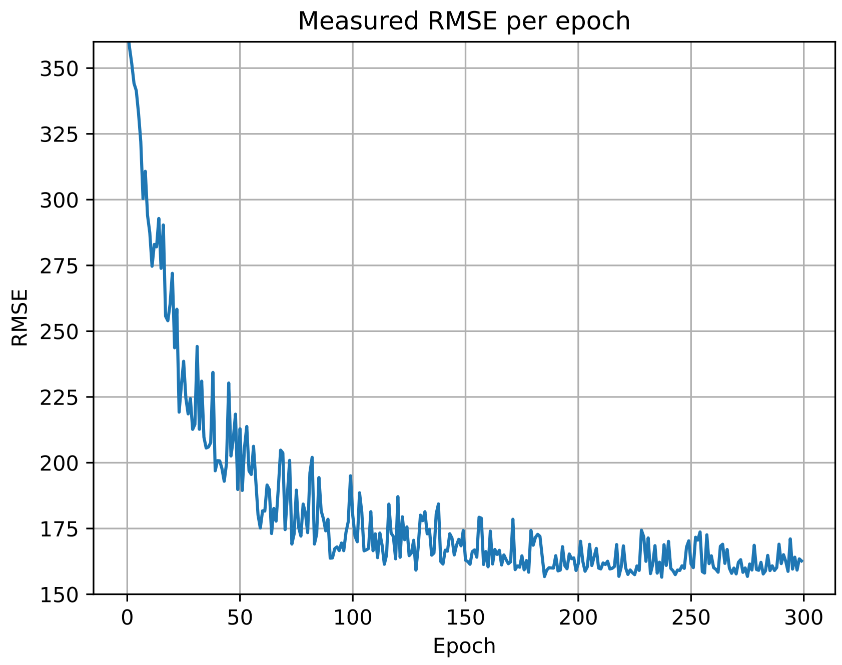

Figure 7.7: Change of RMSE with the governor controlling only samples per epoch

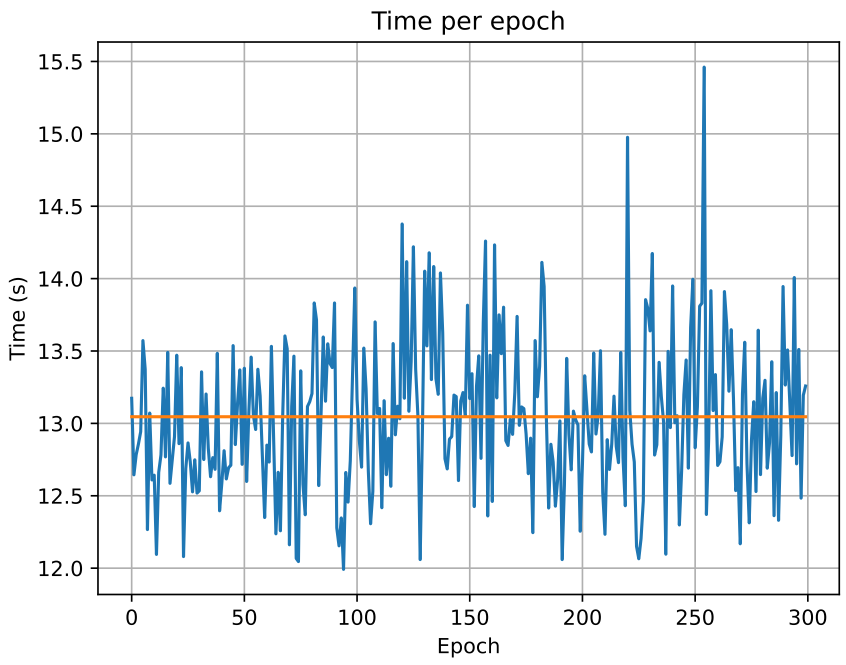

Figure 7.8: Time per epoch with the governor controlling only samples per epoch

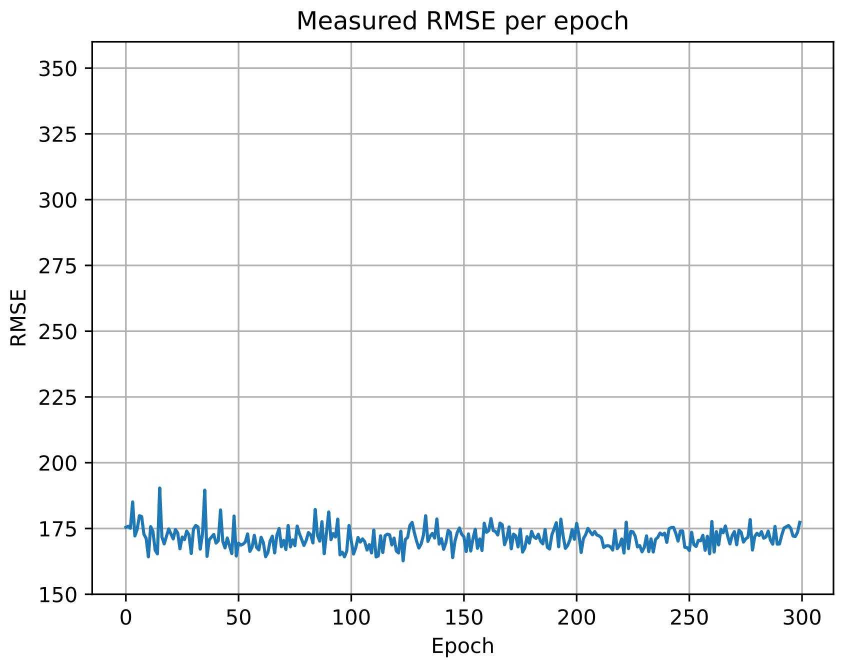

Figure 7.9: Change of RMSE with the governor controlling all three parameters

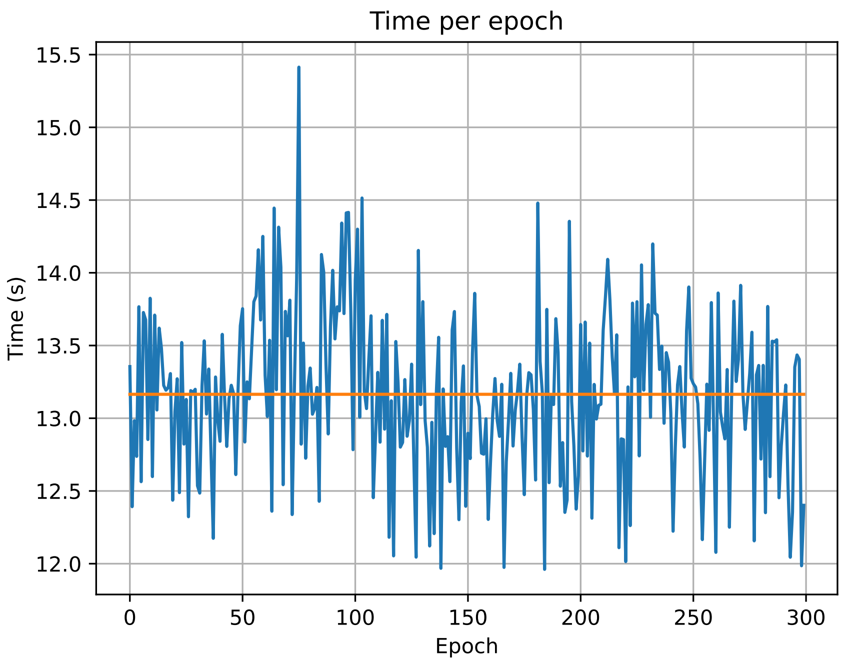

Figure 7.10: Time per epoch with the governor controlling all three parameters

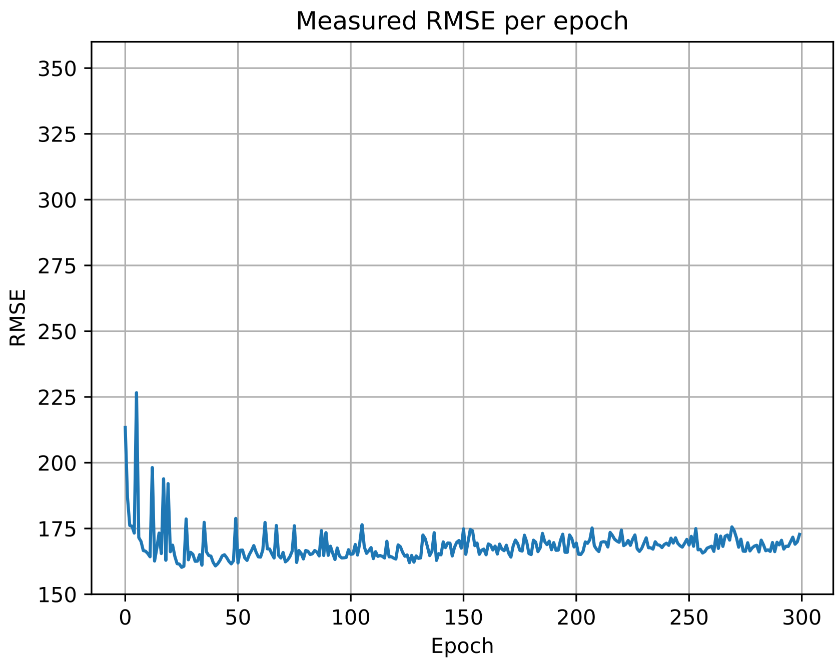

Figure 7.11: Change of RMSE with the governor controlling IPE and SPE

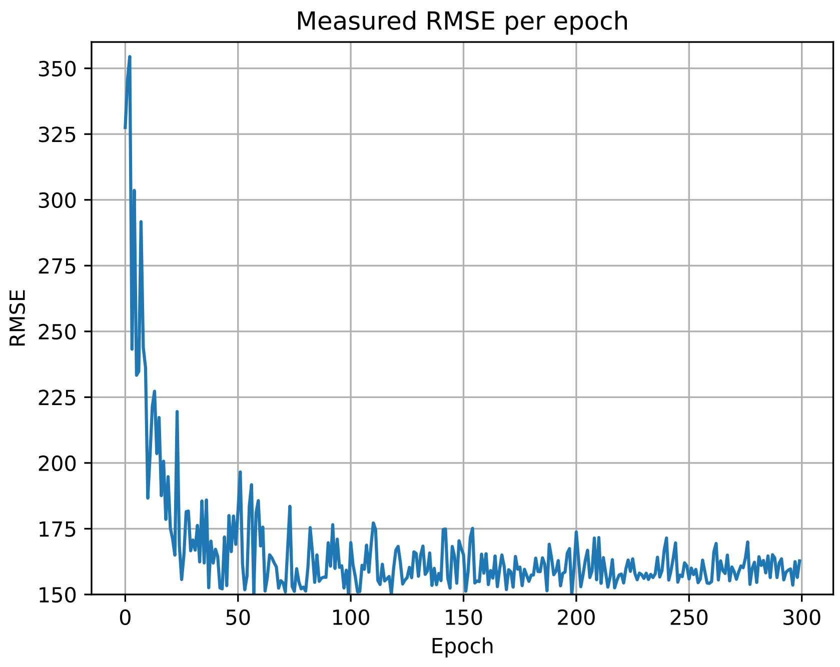

Figure 7.12: Change of RMSE with the governor controlling α and SPE

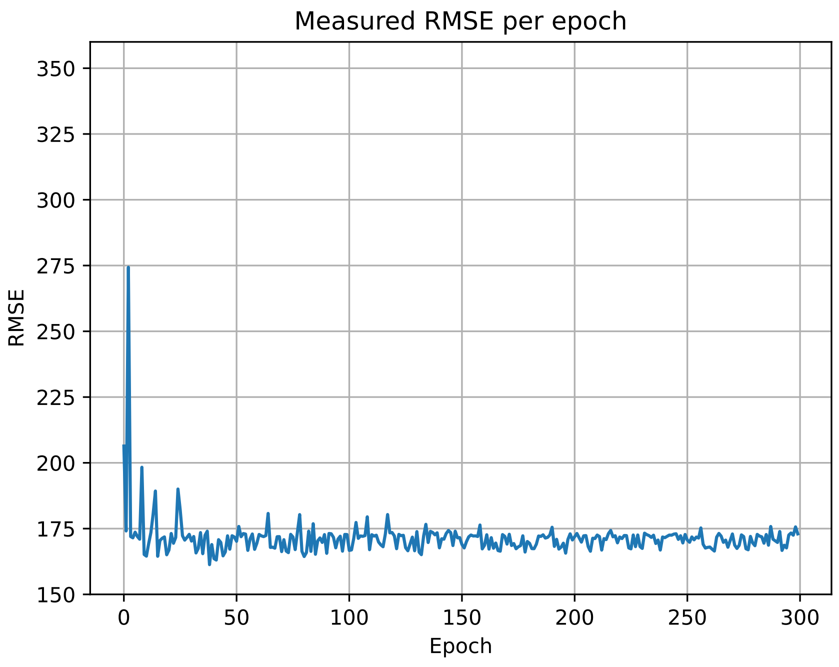

Figure 7.13: Change of RMSE with the governor controlling α and IPE

8 Discussion and Future Work

8.1 Applying JouleGuard

We have elected to only attempt to apply one of the two parts of JouleGuard to the problem of Matrix Factorization. This is because the second part, where the algorithm decides how much to let the degrade the output of the client program, requires a strict ordering of accuracies of the possible configurations of the program. While it does not require knowing the impact a certain configuration will have on the final result of the computation, it requires that the configurations be ordered according to the client programs preference, I.e. this part of JouleGuard must know which configuration is the one that will deliver the highest degree of accuracy[Hof15].

In SGD, each iteration of the algorithm will deliver some change to the accuracy of the whole set. Generally, the less a value has been changed from its initial, random, value the more it will move towards the correct value. Sometimes, especially with a learning rate that is set too high, SGD can start to diverge from the correct value, that is, the accuracy will start decreasing after a time. This happens when the values in H and W pass the values they optimally should be, and start growing too large or small. In addition to this, we want to tune SGD such that it does not reach too high an accuracy with the known values of M, since this can indicate and over-fit to the data, meaning that H and W are tuned too strongly to the data in such a way that it misses the trend real trend of the data in favour of producing the exact values of the known data. This can especially happen in datasets that are uneven in how many values are known for different items in the set, which is the case with the MovieLens dataset, where some movies have far more ratings than others, and some users have far more rating than others. What this implies for the application of JouleGuard is that we also have no expectation of reaching 100% accuracy, even without applying the technique. Indeed, setting a power budget for SGD would only involve stopping the execution when you near the budgeted amount of power, as each iteration is already moving the accuracy of the output towards the desired result, meaning that terminating computation at any point is equivalent to JouleGuards tuning of parameters to generate a less accurate result.

8.2 VDBE Convergence

In none of our tests using a governor does the VDBE method find any stability in which configuration it believes to be the best. We believe this is because we use the change in RMSE as a performance metric for the governor, and this is expected to change over the run of the algorithm, as can be very clearly seen in Fig.7.1, where the governor is not doing anything. This means that even if the governor changed nothing, it's estimates for the change in RMSE will nearly always be wrong, and often by a large amount, such that it will frequently search for new configurations. Despite this, allowing for multiple configurations shows that there is speed to be gained in leveraging some technique for selecting these parameters. However, we suspect that if there is an algorithm that is more appropriately suited for SGD that can still find a configuration, this might be able to get the same level of improved performance without creating so much noise in the accuracy. The noise in the accuracy makes it much harder to algorithmically determine when the algorithm is stabilized, and is another reason we have opted for a set amount of epochs for the tests in this thesis.

8.3 Selection of Parameters

8.3.1 Changing α

Changing the learning rate of SGD during runtime produces faster convergence, as a higher learning rate means that each update brings more change to the values. However, a learning rate that is too large can cause undesirable behaviour, where the value oscillates over and under the correct value, and if it is far too large, can even cause divergence, where the value keeps going away from the correct value after passing through it. By setting it using our governor, we hope to avoid these behaviours, in particular the causing of divergence. We believe this is achieved, as as soon as the change in RMSE becomes negative, the governor will not select that configuration. However, there remains a possibility of such values being chosen for α, and this only being corrected after divergence has begun, and in such cases, while the algorithm will still converge in the end, it will do so slower than if we had not diverged first. We have yet to observe this behaviour in testing, and this might indicate that the maximum bound of the governors allowed values for α is set to an appropriate value.

8.3.2 Changing Iterations per Epoch

Having a distinction between epochs and iterations is not something we have seen in other literature concerning SGD. The reason we create this distinction was originally motivated by the way we chose to program the IPU, where we cannot send new data from the host memory to the device memory while our device code is running. Because of this, if we wished to do more per time we ran the IPUs, it must be on the same data. We quickly saw that this caused desirable behaviour, however, and decided to include it as a parameter for the governor. As we show in the Evaluation chapter, letting the governor control IPE, alone or with any combination of the other parameters, flattens the measured RMSE graph per epoch, while not increasing the run-time per epoch significantly.

This flattening of the measured RMSE curve does make it much more challenging to determine at what epoch one should terminate execution. Normally for RMSE one wants to stop running when the measured RMSE stops changing epoch by epoch, or in practice when it changes less than some acceptable value. When we raise IPE, we essentially hit close to this flat point for each Ω_{sub}, such that comparing the measured RMSE epoch by epoch gives us a much less clear view of the true RMSE of the program. This is exacerbated by the fact that our program only puts one entry per index in H and W in each Ω_{sub}, so that what we are seeing is that the values in H and W become close to that specific index in Ω, without necessarily coming close to the value that best matches all relevant matches in Ω. If we look at Fig. 3.1, we know that each vector in H and W needs to represent a full row or column in M, but the measured RMSE will always only match it to one point in a single epoch. This is not an issue when we measure RMSE across multiple epochs since we select new random values from Ω each epoch, such that we will expect to see an even distribution of the RMSE of a a vector between all relevant point in Ω as the amount of epochs we have run increases. What this implies is that the measured RMSE in a single epoch does not show much on its own, and that we are interested in is the trend of RMSE over multiple epochs. This trend is completely hidden by raising IPE, such that it becomes very hard to decide when to terminate execution. This is another reason we decided to run all tests for a set amount of epochs.

8.3.3 Changing Samples per Epoch

We want to at least have enough vertices such that all tiles of the IPU are executing the compute set, and for optimal hiding of memory access latency we want several threads running per tile, since this means the IPU can swap out a thread when it is blocked by memory access. This means we want the amount of vertices in a single execution step to be some multiple of the amount of tiles on the graph. The way we have implemented our algorithm, each of our three compute sets have SPE amount of vertices, thus we want this value to be high. However, this value also sets the size of Ω_{sub}, which must be copied to the IPU every epoch. This is why we believe there should be some optimal value for this parameter. Tests such as the one shown in Fig. 7.7, where the governor only control SPE did not show more success in achieving a stable best configuration using VDBE, so if there is an optimal value, we did not find it. Our belief about the possibility of there being some optimal value is backed up by Fig. 7.8, where we see that the time per epoch is much more chaotic than in Fig. 7.2, while having a somewhat lower average.

9 Conclusion

This thesis discusses Approximate Computing, with a particular focus on JouleGuard, a system that leverages both machine learning and control theoretic constructs for optimizing running of programs under an energy budget and showcases hardware specialized for machine learning workloads, the Graphcore Intelligence Processing Unit. We implement a version of SGD-based Matrix factorization on the IPU, leveraging some of the techniques from JouleGuard to optimize it. We show that while this method does not behave in a way that suggests it is the optimal solution for this particular problem, we see that it still causes faster convergence.
By demonstrating this, we show that Graphcore IPUs exhibit enough flexibility that there are significant gains to be made by configuring the programs running on them well, and that machine learning algorithms are sensitive to the parameters that we define, in particular Iterations per Epoch parameter we introduce in an attempt to reduce the amount of host-device data transfer.

Bibliography

[Hig90] Lee Higbee. “Quick and easy cache performance analysis.” In: *ACM SIGARCH Computer Architecture News* 18.2 (1990), pp. 33–44.

[Val90] Leslie G Valiant. “A bridging model for parallel computation.” In: *Communications of the ACM* 33.8 (1990), pp. 103–111.

[KRS00] Paul Keltcher, Stephen Richardson, and Stuart Siu. “An equal area comparison of embedded dram and sram memory architectures for a chip multiprocessor.” In: *HP LABS TECHNICAL REPORT HPL-2000-53*. Citeseer. 2000.

[Gui13] Design Guide. “Cuda c programming guide.” In: *NVIDIA, July* (2013).

[HK15] F. Maxwell Harper and Joseph A. Konstan. “The MovieLens Datasets: History and Context.” In: *ACM Trans. Interact. Intell. Syst.* 5.4 (Dec. 2015). ISSN: 2160-6455. DOI: 10.1145/2827872. URL: https://doi.org/10.1145/2827872.

[Hof15] Henry Hoffmann. “JouleGuard: Energy Guarantees for Approximate Applications.” In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP ’15. Monterey, California: Association for Computing Machinery, 2015, 198–214. ISBN: 9781450338349. DOI: 10.1145/2815400.2815403. URL: https://doi.org/10.1145/2815400.2815403.

[GUH16] Carlos A. Gomez-Uribe and Neil Hunt. “The Netflix Recommender System: Algorithms, Business Value, and Innovation.” In: *ACM Trans. Manage. Inf. Syst.* 6.4 (Dec. 2016). ISSN: 2158-656X. DOI: 10.1145/2843948. URL: https://doi.org/10.1145/2843948.

[Kor+18] Kunal Korgaonkar et al. “Density Tradeoffs of Non-Volatile Memory as a Replacement for SRAM Based Last Level Cache.” In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 315–327. DOI: 10.1109/ISCA.2018.00035.

[Jia+19] Zhe Jia et al. “Dissecting the graphcore ipu architecture via microbenchmarking.” In: *arXiv preprint arXiv:1912.03413* (2019).

[Graa] Graphcore. *IPU Programmer’s Guide*. https://docs.graphcore.ai/projects/ipu-overview/en/latest/index.html. Accessed: 2021-05-02.

[Grab] Graphcore. *Poplar and PopLibs User Guide*. `https://docs.graphcore.ai/projects/poplar-user-guide/en/latest/introduction.html`. Accessed: 2021-05-02.

A Running the Program

Compiling the program included with this thesis requires a system with g++, make and the Poplar SDK installed. When these prerequisites are met, run make in the directory to compile the program and the IPU vertices. The program is not set up to run using a virtual IPU model, and thus requires the system it be run on have a IPU. When the program is compiled, it can be invoked with the command ./approximationOptimizer, and it requires a path to a dataset formatted in the exact way the MovieLens dataset is formatted be passed as the first argument. Further optional arguments allow you to decide which directory to write output data about each epoch to, how many configurations the governor should generate, and finally which of the parameters the governor should control.