Eivind Lie Andreassen

Automatic Model Parallelism for Deep Learning Using Execution Time Modelling and Evolutionary Computation

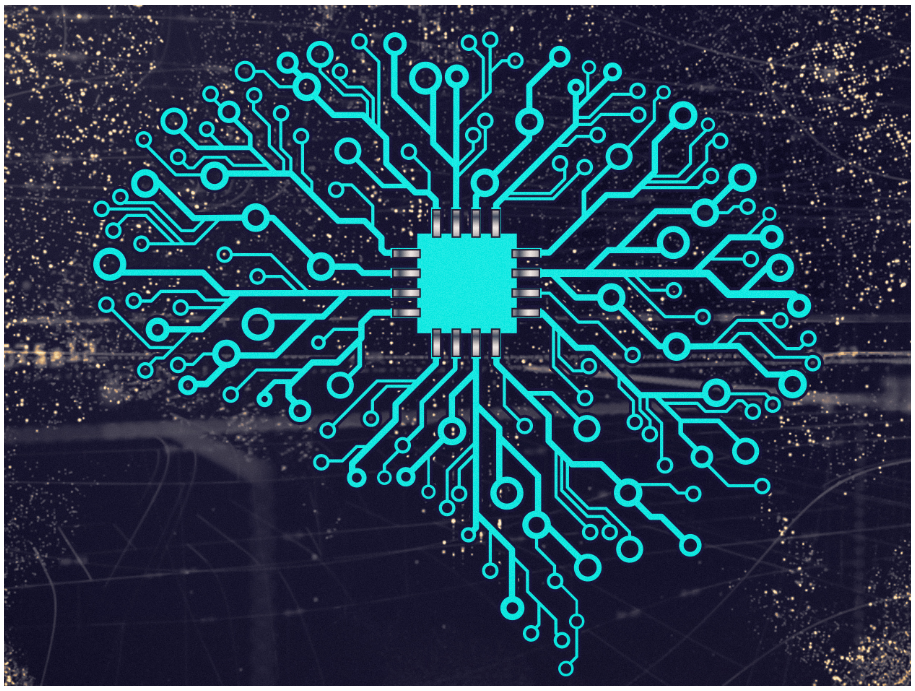
Master's thesis in Computer Science

Supervisors: Keith L. Downing, Arjun Chandra & Lorenzo Cevolani

June 2020

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science



NTNU
Kunnskap for en bedre verden

Eivind Lie Andreassen

Automatic Model Parallelism for Deep Learning Using Execution Time Modelling and Evolutionary Computation

Master project, spring 2020

Data and Artificial Intelligence Group
Department of Computer Science
Faculty of Information Technology and Electrical Engineering

NTNU

Abstract

Using methods from the field of evolutionary computation combined with an execution time simulator for deep neural networks, this project aims to automate the configuration of model parallel training strategies for deep learning. Recent years have seen significant advances in the power and utility of deep learning, but also an increase in the complexity of deployed models, leading to large computational requirements. Parallelism techniques have become essential, both in order to reduce training time and to fit the models in the memory of available computational devices. Applying parallelism to deep learning is non-trivial, but much of the complexity of the task can be alleviated by applying optimization techniques such as evolutionary algorithms. Moreover, through the use of a simulation of the training process, parallel configurations can be found without access to expensive training hardware, while terminating faster than possible if evaluation runs on real hardware were to be carried out.

This report presents an execution simulator for neural networks, and two optimization algorithms for finding configurations for neural networks – a genetic algorithm, and a MAP-Elites algorithm. The focus is on solving the device placement problem, in which the individual operations in a neural network are placed onto a set of devices for model parallel execution. In the experiments, the two algorithms are shown to outperform a baseline consisting of a hill climbing and a simulated annealing algorithm. The algorithms are able to find good solutions across several problem instances, with the optimal solution being found in the simplest instances.

The impact of the execution simulator is also evaluated through experiments. These indicate that the execution simulator gives an approximately correct ordering of solutions with regards to their quality, indicating that an optimization process run against the simulator will yield solutions that are valid for application in the real world.

Sammendrag

Dette prosjektet tar sikte på å automatisk finne modell-parallelle konfigurasjoner for dype nevrale nettverk ved hjelp av en kombinasjon av evolusjonære algoritmer og simulering av kjøretid for dyp læring. De siste årene har kraften og nytteverdien av dyp læring økt betydelig, men kompleksiteten til modellene har samtidig økt, noe som har ledet til et stort behov for regnekraft. Dette har gjort parallelliseringsteknikker helt essensielle – både for å redusere tiden det tar å trene opp slike modeller, og for å få plass til modellene i hurtigminnet til tilgjengelige prosesseringsenheter. Bruken av parallellisering for dyp læring er et ikke-trivielt problem, men mye av denne kompleksiteten kan avlastes ved å benytte moderne optimeringsteknikker, slik som evolusjonære algoritmer. Videre kan bruken av en simulering av treningsprosessen muliggjøre en slik prosess uten behov for tilgang til dyr treningsmaskinvare. Dette vil også la prosessen terminere fortere enn hvis evalueringen skal foregå gjennom testing på den fysiske maskinvaren.

Denne rapporten presenter en kjøretidssimulator for nevrale nettverk, og to optimeringsalgoritmer som kan finne treningskonfigurasjoner for nevrale nettverk – en genetisk algoritme, og en MAP-Elites-algoritme. Fokuset er på å løse enhetsplasseringsproblemet, som innebærer å plassere individuelle operasjoner fra et nevralt nettverk på prosesseringsenheter, slik at nettverket kan bli trent i en modell-parallell konfigurasjon. I eksperimentene gir de to optimeringsalgoritmene bedre resultater enn et sammenlikningsgrunnlag bestående av "Hill Climbing"-algoritmen og "Simulated Annealing"-algoritmen. De evolusjonære algoritmene er i stand til å finne gode løsninger for en rekke probleminstanser, og finner optimale løsninger i de enkleste instansene.

Påvirkningen kjøretidssimulatoren har på løsningene blir også evaluert gjennom eksperimentene. Resultatene indikerer at simulatoren gir en tilnærmet korrekt sortering av løsningene basert på kjøretid. Dette betyr at en optimeringsprosess som bruker simulatoren for evaluering av løsninger vil ende opp på en endelig løsning som er gyldig for bruk i den virkelige verden.

Preface

This master project was carried out at the Norwegian University of Science and Technology in the spring of 2020, as part of the course *TDT4900 – Computer Science, Master's Thesis*. It concludes my five-year education in Computer Science. The project was conducted in collaboration with Graphcore, and was supervised by professor Keith L. Downing of NTNU, and Arjun Chandra and Lorenzo Cevolani of Graphcore.

There is no doubt that deep learning has accomplished numerous impressive feats in the last few years, and it is a highly interesting field of research. I was therefore never in doubt that I wanted my master's thesis to touch this field. One option would be to look into the application of deep learning to a specific problem. However, what appealed to me about the topic that I ended up writing about, was how it is an important foundational issue for deep learning, the solution of which can facilitate improvements within all areas of deep learning research. I also think it is important to always consider what tools actually fit the task at hand, instead of using whatever is most popular at the moment, as I think we often tend to do. The ability to combine the classic discipline of evolutionary computation with the current public favourite – deep learning – was therefore particularly alluring.

Parallel to this project, the world has faced one of the greatest crises of our time. A highly contagious virus has spread throughout the globe, paralyzing society in countries all around the world. In a way, the crisis has made matters such as education and scientific research seem quite small. At the same time, we have seen how technology can be a massive aid and enable us to carry on even through such extreme situations. I am therefore proud to be able to, in a small way, make a contribution to the advancement of technology.

I would like to thank my supervisors for valued input and guidance. A special thanks goes to my supervisors at Graphcore for the initial problem proposal, and for also allowing me to choose my own direction within its boundaries. Thanks also to family and friends for supporting me throughout the project.

Finally, thank *you*, who at this point have made it through the first few pages of my thesis. I hope that you will find the following parts interesting – I very much enjoyed the work behind them.

Eivind Lie Andreassen
Trondheim, June 12, 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent years have seen many impressive results produced by deep learning methods. In games such as Go and chess, deep neural networks have advanced to proficiency levels comparable to or exceeding that of humans [Silver et al., 2016, 2017]. Deep learning models are able to accurately recognize the content of images and the words of human speech [He et al., 2016; Synnaeve et al., 2019]. However, the models used in these applications are massive in both size and complexity, leading to immense computational costs. The training process of neural networks may stretch over several days, and the models often do not fit into the memory of any single computational device. Furthermore, the requirements of deep learning currently grows at a higher rate than advances in hardware technology.

In order to meet the requirements of deep learning, multiple computational devices must be utilized simultaneously. In general, there are two different methods for doing this. Data parallelism performs the same type of computation on all devices, but with each device processing different data. Conversely, model parallelism splits the model across devices, with each device performing parts of the total calculation. Data parallelism can greatly reduce the execution time of the training process, but since the entire model is replicated across all devices, it does not reduce the memory consumption on a single device. Consequently, model parallelism is the only option when the memory of any single device is insufficient to hold the entire model.

The task of finding the right model parallel training configuration for a large neural network is a non-trivial one. Moreover, it is often desirable to combine model and data parallelism in order to exploit the advantages of both, further increasing the complexity of the task. This leads to a need for significant experience in order to solve this task effectively, hurting the accessibility of this type of models. In addition, this task expends valuable time that could otherwise be put towards improving the model itself.

Modern optimization methods can be utilized to automate this task. Previous works have applied both reinforcement learning [Mirhoseini et al., 2017; Addanki et al., 2019] and dynamic programming [Jia et al., 2018a] to this end. Other optimization methods such as evolutionary algorithms should be equally applicable. The automation of this task frees up time for the deep learning experts that can be put towards other tasks. Conceivably, such automated systems may also be able to discover non-trivial solutions that can provide additional speed-ups over expert placements.

Optimization algorithms rely on frequent evaluation of an objective function – in this case, the execution time of a given network configuration. Using end-to-end benchmarking of a neural network for this purpose requires significant amounts of time – even if only running for a few training steps. Improving the speed of a single evaluation can lead to a substantial speed-up of the process as a whole. One method for improving the evaluation time is through the introduction of a model of the network execution time [Jia et al., 2018b; Addanki et al., 2019]. By using such a model for evaluations instead of benchmarks, the time until convergence for the optimization process can be drastically reduced.

1.1 Goal and Research Questions

There are clear advantages to automating the process of distributing operations of a neural network over multiple devices. I call this problem the *device placement problem*, inspired by the terminology established by Mirhoseini et al. [2017]. Previous approaches have largely focused on the application of reinforcement learning to this problem, but evolutionary computation has previously been successfully applied to similar problems. For this project, a genetic algorithm and a MAP-Elites algorithm was implemented, and the evaluation of these algorithms form the first part of the research goal.

An execution simulator was also implemented. The application of such a simulator when solving the device placement problem can yield significant speed-ups. However, if this has a large negative impact on the final solutions produced through the optimization, the use of this simulator may not be justifiable. Evaluating the impact of the execution simulator was therefore a crucial task.

Combining these two factors, the overall goal of the project was as follows:

Goal *Evaluating the performance of two simulation-based evolutionary algorithms for optimizing device placement of deep neural network, and the impact of the simulator on produced solutions.*

The first research question formalizes the evaluation of the optimization methods. There are two main properties of an optimization algorithm that need to be considered: how good the solutions it produces are, and how much computation is needed to arrive at these solutions. Since a simulator was used in this project, securing an efficient process, it was decided that the quality of the solutions was the most important factor. However, in order to ensure a fair comparison, the available computation time must be equal for all methods. The evaluation must also be performed against some baseline. In this project, this baseline consisted of a combination of two simple optimization methods – hill climbing and simulated annealing – and, when available, trivial solutions to the problem.

Research Question 1 *How does the performance of the genetic algorithm and the MAP-Elites algorithm compare to a simple baseline of trivial solutions and classic optimization methods, with respect to the quality of solutions produced within a fixed amount of computational time?*

The second research question formalizes the evaluation of the execution simulator. The execution simulator was only meant to be used in the optimization process itself, and therefore the strict accuracy of the simulator was secondary. The primary measurement for the viability of the execution simulator was its ability to facilitate the production of good solutions to the device placement problem. It was therefore most important to determine its impact on the quality of solutions.

Research Question 2 *How does the use of an execution simulator influence the quality of device placements proposed by an algorithm that uses simulated run time as an objective function during optimization?*

1.2 Research Method

This project used an experimental, empirical approach to research. Prototypical implementations were made of proposed methods, and experiments based on test runs and calculations of suitable metrics formed the basis for evaluation of the methods. Whenever possible, multiple test runs were performed for each experimental configuration, allowing statistical analysis of the results, and decreasing the impact of random fluctuations in performance. However, due to the nature of the methods and problem, some experiments were so computationally expensive that they by necessity had to be limited to a smaller number of runs. In such cases, qualitative evaluation became essential.

Practical experiments provide good data on how the methods perform when applied to real problems. Additionally, evolutionary algorithms do not lend themselves to theoretical analysis. Therefore, empirical evaluation was the only applicable option.

1.3 Contributions

The contributions of this paper are an execution simulator for neural networks, building on previous efforts by Qi et al. [2016] and Addanki et al. [2019], along with the evaluation of two evolutionary algorithms for solving the device placement problem. The simulator supports the estimation of run time for any device placement of a given neural network on a given device configuration, without requiring access to the hardware itself. Furthermore, it does so at a fraction of the time required for benchmarking the real execution time. Together, these components form a novel system for solving the device placement problem.

1.4 Thesis Structure

The rest of this thesis is structured as follows. Chapter 2 goes into the background for the thesis. An overview of concepts that are important for understanding the rest of the report is given in Section 2.1. Section 2.2 contains the protocol outlining the process of the literature review, while Section 2.3 presents related work. Section 2.4 gives the motivation for this project, based on the concepts introduced in the preceding sections.

Chapter 3 presents the system developed in this project, and relevant technologies that have been used. Section 3.1 presents the execution simulator, while Section 3.2 presents the genetic algorithm and the MAP-Elites algorithm that have been implemented for solving the device placement problem. Section 3.3 gives a brief overview of PyTorch, which is the deep learning framework used for benchmarking neural networks on real hardware. Finally, Section 3.4 gives a short summary of the chapter.

Chapter 4 contains the presentation of the experiments that have been carried out in order to evaluate the system, along with the presentation and analysis of their results. Section 4.1 gives an overview of the experimental plan and an outline of each experiment, while Section 4.2 goes into details about the experimental setup that is shared across experiments. Section 4.3 then gives a detailed description of each individual experiment, along with its results and their analysis. Section 4.4 summarizes the experiments chapter.

Chapter 5 concludes the report. Section 5.1 contains the discussion, while Section 5.2 summarizes the contributions of this report. Finally, Section 5.3 gives an outline of possible directions for future research.

Chapter 2

Background Theory and Motivation

This section goes into the background theory necessary for understanding this thesis, and earlier work that has been done within the fields of automatic device placement and distribution strategies for deep learning, execution modelling of deep neural networks, and evolutionary computation for process scheduling problems. Section 2.1 explains some important background concepts necessary for the understanding of the rest of the thesis. Section 2.2 contains the review protocol, explaining how the sources for the related work were discovered. Section 2.3 presents the most relevant related work in the field. Section 2.4 motivates the work done for this thesis, with a basis in the background theory and related work presented in the preceding sections.

A literature review was carried out in the project preceding this thesis [Andreassen, 2019]. This chapter contains an amended version of the corresponding chapter from that report. In particular, a section describing related work in the field of using evolutionary computation for process scheduling has been added.

2.1 Background

2.1.1 Artificial Neural Networks

An artificial neural networks is a type of model inspired by the human brain, consisting of a network of interconnected neurons. In their simplest form, each of these neurons is a linear combination of its inputs and a set of associated weights. By changing the weights of the neuron, the contribution of each input to the output can be controlled, thus modifying the behaviour of the model. In order to introduce some non-linearity to the model, increasing its predictive power, an activation function is typically applied to the aggregated output of a neuron. Due to the process used for training neural networks, this function needs to be differentiable. A popular choice has historically been sigmoid functions such as the logistic function:

$$f(x) = \frac{1}{1+e^x}. \tag{2.1}$$

In recent years, the simpler rectified linear unit (ReLU) has become the dominant activation function:

$$f(x) = max(0, x) \tag{2.2}$$

The linear combination of weights and inputs, and the non-linear activation function together constitute a single neuron. [Mitchell, 1997, pp. 81-95]

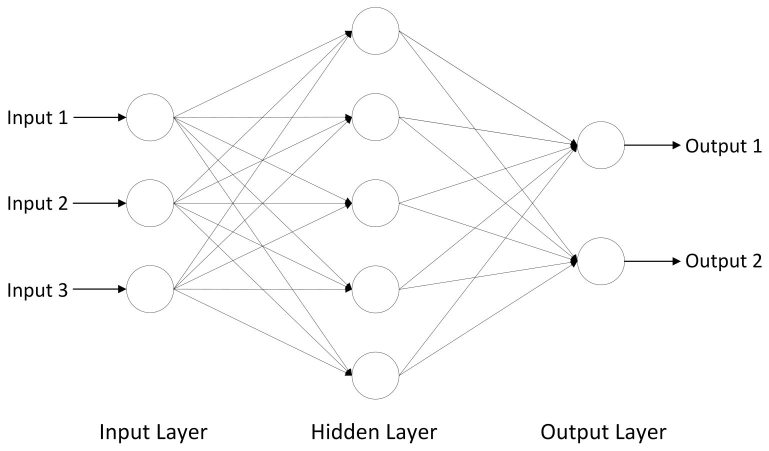


Figure 2.1: Illustration of a multilayer perceptron neural network.

In order to increase the power of the model, allowing it to represent a larger variety of functions, neurons are arranged in networks called multilayer perceptrons (MLP), as shown in Figure 2.1. Multilayer perceptrons are often also called feed-forward networks, as the data flows in one direction through the network, or fully connected networks, since all layers use all the outputs from the previous layer. An individual layer in such a network is often called a densely connected or simply dense layer, since it is connected to all the neurons in the previous layer. [Goodfellow et al., 2016, pp. 164-167]

Neural networks are normally optimized using a process called gradient descent. The intuition behind this method is that we can follow a path in the steepest downward direction towards a minimum in our error space. Mathematically, this is defined as going in the reverse direction of the gradient of the error with regards to the parameters that we want to update, which in this case are the trainable weights of our neural network model. If the gradients for the entire error space could be calculated, a minimum could be found analytically. In practice, this is not possible, and instead the gradients are calculated and the weights updated using what is called the backpropagation algorithm. This entails a forward and a backward pass through the network. In the forward pass, the inputs for the given training sample are fed into the network, and the outputs of each neuron are calculated, all the way to the output layer. In the backward pass, the chain rule for differentiation is repeatedly applied using the calculated outputs (often called activations) of each neuron in order to calculate the gradients. Once the gradients for the relevant inputs are calculated, a small step is taken in the direction opposing the gradient of each trainable parameter, and the process is repeated. [Mitchell, 1997, pp. 95-100]

Originally, the entire available data set would be used to calculate the gradients for each training step. However, this is computationally expensive for large data sets. Moreover, this method has proven to be susceptible to getting stuck in local minima. Instead, a smaller fraction of the total data set is used to calculate the gradients for each training step. This is called stochastic gradient descent, since we no longer follow the true gradient, but an approximation, introducing some stochasticity. This stochasticity enables the algorithm to get out of some local minima, in addition to being much faster to execute, since a much smaller amount of data is used for each training step. The extreme variant of stochastic gradient descent is using only a single training sample for each step. This is rarely used; instead, we rely on minibatch stochastic gradient descent. Minibatches, often only called batches, are subsets of the training data that are used for each step, providing a balance between extreme stochastic gradient descent and the deterministic variant. [Mitchell, 1997, pp. 92-93]

Most operations used in neural networks can be defined as operations on matrices or their generalization: tensors. For instance, the functionality of a single simple neuron can be written as a matrix multiplication combined with an element-wise application of the activation function. All such operations can be generalized to operations on tensors, where minibatches can be included as one of the dimensions of the tensor. This allows the representation of an entire training step on the network as iteratively applying a set of operators on tensors. These operations and intermediate tensors are often collected in computational graphs, where each node is an operation, and each edge represents the flow of tensors between operations. This representation has multiple advantages. Firstly, such tensor operations are easily mapped to modern GPUs, providing access to accelerated execution. Secondly, such a graph allows simple distribution of computation between multiple devices. [Goodfellow et al., 2016, pp. 205-210]

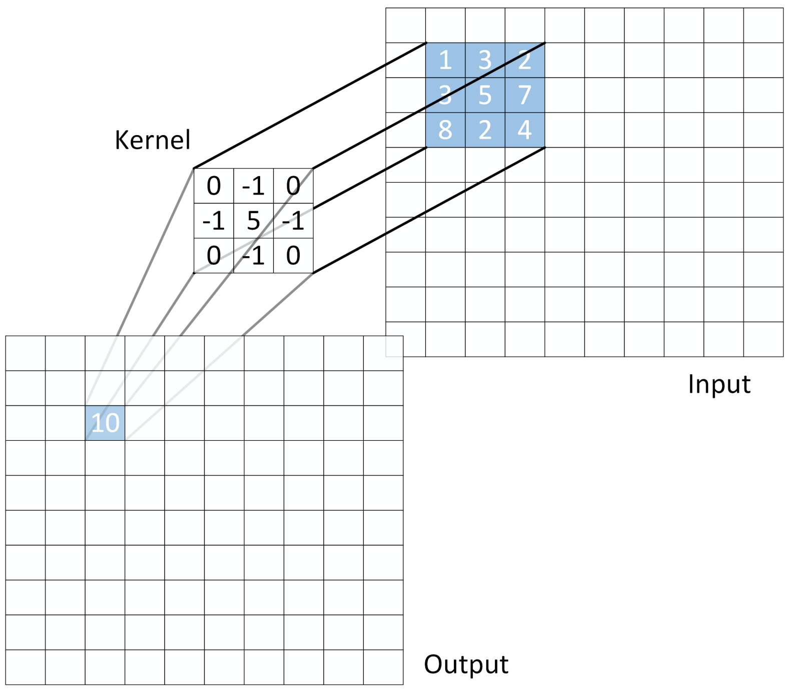


Figure 2.2: Illustration of a 2D convolution, where a kernel is mapped over the input grid, producing an output grid. In this case, the kernel is a sharpen operation, commonly used in image processing. The kernel takes the value of a given pixel and surrounding pixels, multiplies it by the corresponding weight, and takes the sum of the individual results to produce a single value.

2.1.2 Deep Learning

Deep learning is a special class of machine learning methods based on artificial neural networks. Deep learning entails the use of artificial neural networks that are considered "deep" — typically with more than one hidden layer (or equivalently, with more than three layers in total). One application of deep learning is to simply increase the expressive power of classic feed-forward neural networks. However, the direction of deep learning that has recently received the most attention in research and practical applications is the use of more specialized layers. This allows a neural network to automatically discover good features as part of training. Two popular examples are convolutional neural networks (CNNs), which excel at image recognition, and recurrent neural networks (RNNs), which are good at recognizing patterns that develop over time, such as in speech recognition.

Convolutional neural networks introduce convolutional layers that map a kernel over all input values, as illustrated in Figure 2.2. The same kernel is used to produce outputs from all input values. The figure shows a 2D convolution, but in principle, any dimensionality is possible, as long as it corresponds to the dimensions of the input data. 2D convolutions are well known from the field of image processing, with specific kernels available for operations such as sharpening, blurring, and edge detection. When used in a convolutional network, however, the weights of the kernel are trainable parameters. This enables a convolutional network to produce features from the raw input values that will aid it in the classification task, removing the need for expert-generated features. Typically, several kernels are combined in a single convolutional layer to produce multiple outputs. [Goodfellow et al., 2016, pp. 330-334]

An important characteristic of convolutional layers is their ability to offer translational invariance; that is, the positioning of a given element in an image does not affect the model's ability to classify it correctly. This is achieved in part because the same parameters are used for the entire input. However, another mechanism that helps achieve this is pooling layers, which are often applied directly after convolutional layers. Pooling layers map a kernel over the input in a similar way to a convolutional layer, and produce an aggregated result. The aggregate can be either the maximum or minimum value, or the average. Using the maximum value, producing what is called max-pooling, is the most common.

The recurrent neural network is a different type of specialized deep neural network. Such networks are fed with sequential data, and have mechanisms for saving the values of previous outputs in the network. Such mechanisms come in the form of backwards connections in the network, and explicit memory structures in individual neurons that can be controlled by the networks. Recurrent neural networks excel at applications where temporal relations are important, and have been successfully applied to fields such as speech recognition and machine translation. [Goodfellow et al., 2016, pp. 373-420]

Deep learning typically requires massive amounts of data and processing power. The increase in the number of layers massively expands the number of trained parameters. This sets high requirements for the availability of memory and computational power. Moreover, the current trends in machine learning are towards progressively larger models, with a state-of-the-art CNN having upwards of 101 layers and $\sim$829M parameters [Mahajan et al., 2018], and a state of the art RNN having $\sim$1.5B parameters [Radford et al., 2019].

2.1.3 Data Parallelism

The massive computational requirements of deep learning necessitates techniques for distributing the load over multiple devices. Data parallelism is one such technique. When using data parallelism, the entire model is replicated across multiple computational devices. Each device processes part of the total input data, carrying out both forward and backward passes. This allows large parts of the computational load to be parallelized, and provides good load balancing, since all devices carry out the same amount of computation.

During training, the parameters of the network are updated in order to improve the predictions made by the network. In order to maintain the same network on all devices, a synchronization technique is therefore required. This technique can be either centralized or decentralized. In the centralized case, a single parameter server is responsible for all updates on trainable parameters, and individual workers send gradients and fetch updated parameters from this server. In the decentralized case, all workers must exchange gradients so that these can be aggregated and applied in a parameter update by each worker, ensuring that all workers carry out an equal update of the parameters.

In both cases, significant amounts of communication is required. This can constitute a serious overhead, and reduce the utilization of individual devices as they are required to wait for communication to complete. Due to this issue, data parallelism is most efficient for sparsely connected architectures or architectures with a significant computational load, leading to the speed-up in computations outweighing the overhead introduced by communication. An example of a sparsely connected architecture is the two-tower version of AlexNet [Krizhevsky et al., 2012], where the initial inputs are fed into two branches with a series of convolutional operations trained on separate GPUs, with the outputs of the branches only combined in the last couple of layers. This allows the network to train on two devices while minimizing communication.

The size of modern deep neural networks does not only constitute a computational obstacle — the size of parameters and activations may also be so large that the network does not fit into the memory of a single device. Since data parallelism replicates the entire network across all devices, it does not solve this problem.

2.1.4 Model Parallelism

Model parallelism is another technique for distributing a neural network over multiple devices. When model parallelism is applied, each device executes a separate part of the model from all the other devices. Examples include distributions of the layers of a network, the neurons of a single densely connected layer, or different kernels in a convolutional layer, across multiple devices. Parallelizing over more domain-specific dimensions such as the width or height of an image in a convolutional layer is also possible.

As with data parallelism, communication overheads can significantly increase the execution time of the network when using model parallelism. However, unlike data parallelism, the manner in which the network is parallelized can deeply impact this overhead when using model parallelism. This is due to the difference in number of connections and size of transferred data between different parts of the network. Therefore, utilizing model parallelism requires knowledge about the network architecture in order to determine suitable dimensions for parallelism. Applying model parallelism is therefore often more complicated than using data parallelism.

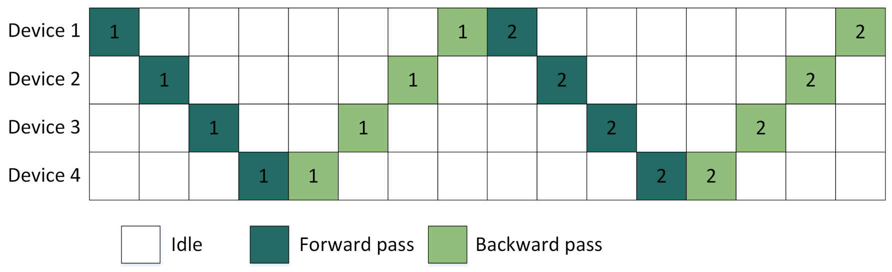
Since the network itself is distributed between devices when using model parallelism, the memory consumption on each individual device is reduced. Model parallelism can thus solve the problem of models growing too large to fit onto a single device. Typically, this has been the reason for applying model parallelism, with data parallelism being preferred when the goal is a computational speed-up.

Pipeline Parallelism

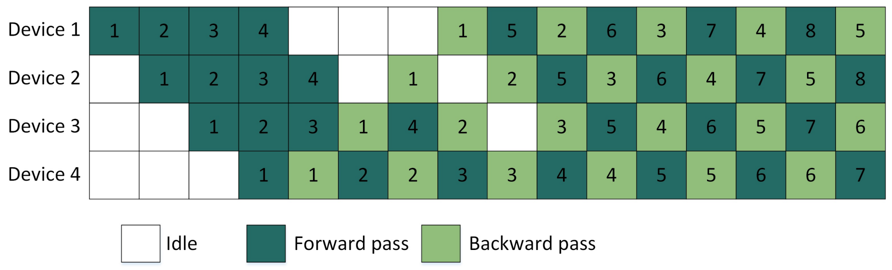
Pipeline parallelism is a special case of model parallelism that has gained some attention in recent years. When using pipeline parallelism, the model is divided into multiple stages consisting of consecutive layers, with each stage mapped to a separate GPU. This creates a pipeline, similar to the execution of different tasks in a modern CPU. In the trivial case, with a single batch of data being handled by the network at any given time, as shown in Figure 2.3a, this is slower than using a single GPU. At any given time, only a single GPU will be executing, with all other GPUs waiting for the data required for them to execute their part of the network. Moreover, the distribution of the network between devices adds communication to the process.

However, if multiple batches are allowed to be executed at once, different devices can process different batches at the same time, allowing for true parallelism. This is shown in Figure 2.3b, which visualizes the execution of pipeline parallelism in the PipeDream system [Harlap et al., 2018]. In this case, the network is distributed across four devices, and a maximum number of four batches are allowed in the system at any given time. As can be seen, after an initialization phase, a degree of parallelism of four is achieved, providing full utilization of devices. Furthermore, computation and communication can be overlapped, reducing the impact of communication times.

Pipeline parallelism is a fairly complicated technique. Careful scheduling of batches is required in order to achieve good utilization of resources. Moreover, since multiple batches are being executed at the same time, trainable parameters must be versioned in order to ensure that forward and backward passes are carried out using the same weights. This also means that many of the calculations in the network will be carried out using stale weights, impacting the output of the optimization process. Consequently, when using pipeline parallelism, the final result of the training process is not necessarily the same as if no parallelism was applied. Finally, the saving of versioned weights considerably increases the memory requirements of the training process. This means that pipeline parallelism is



(a) A naïve pipeline implementation for a deep learning task. A single batch is processed, completing both forward and backward passes, before a new batch is started. Notice how there is no real parallelism here, and there is a huge under-utilization of resources.



(b) A more advanced pipeline strategy taken from the PipeDream system [Harlap et al., 2018]. Notice how after the system has reached steady state, all devices are being utilized at the same time, with a degree of parallelism equal to the number of devices.

Figure 2.3: Examples of execution timelines of two pipeline configurations of deep learning on four distinct devices. In both implementations, stages consisting of consecutive operations in the network are placed on individual devices. However, the scheduling of computations differ between the approaches, with the naïve approach only allowing a single batch of data in the pipeline at any time, while the PipeDream approach allows as many batches as there are devices in the pipeline at any given time.

unsuitable when the problem is models being too large for device memory, but it is an option for increasing the throughput of the training process.

2.1.5 Evolutionary Computation

Evolutionary computation – also called evolutionary algorithms – is a category of optimization algorithms inspired by natural evolution. It can be considered a framework into which several algorithms fit – notably also some classic algorithms that are not directly inspired by evolution, such as hill climbing and simulated annealing. Evolutionary algorithms iteratively improve a solution or set of solutions by for each step creating variations of the solutions and evaluating their performance. The main components of an evolutionary algorithm are:

- representation
- evaluation function
- population
- parent selection mechanism
- variation operators
- survivor selection mechanisms.

The representation specifies how candidate solutions should be defined in the algorithm. The evaluation function provides the means through which the quality of any given solution is evaluated. The population contains the candidate solution(s) at any given step in the algorithm run. The variation operators specify how new solutions are produced from the existing solutions. These may take a single or multiple parent solution(s) as input. The former is often called mutation, while the latter is usually called crossover. Finally, parent and survivor selection mechanisms determine which solutions form the input to the variation operators, and which outputs from the variation operators should be accepted into the population, respectively. [Eiben et al., 2015, pp. 25-34]

Hill Climbing

Hill climbing is a simple local search optimization algorithm. Algorithm 1 shows pseudo-code for hill climbing, with the problem formulated as a maximization problem. It starts from a single initial solution, and continually considers neighbouring solutions. Any time a better solution is found, this solution is accepted as the new solution, and the neighbours of this solution are then similarly evaluated.

Random hill climbing, in which a random neighbour is selected at each step, fits into the evolutionary computation framework with a population size of one, random mutation as the only variational operator, deterministic selection of the only solution in the population as parent, and deterministic selection of the best of the new candidate and existing solution as survivor selection. There are also other variations of the hill climbing algorithm, such as steepest-ascent hill climbing, in which all neighbours are evaluated at each step, with the best neighbour being selected. [Russell and Norvig, 2016, pp. 122-125]

Algorithm 1: Random Hill Climbing

```
Result: locally optimal solution $s$
1  $s \leftarrow$ generate random solution
2  $f \leftarrow$ evaluate $s$ using evaluation function
3  for $N$ iterations do
4      $s_1 \leftarrow$ pick a random neighbour of $s$
5      $f_1 \leftarrow$ evaluate $s_1$ using evaluation function
6      if $f_1 > f$ then
7          $s \leftarrow s_1$
8          $f \leftarrow f_1$
9      end
10 end
```

In a convex space, the hill-climbing algorithm is guaranteed to find the optimal solution. However, in solution spaces with local optima, the hill-climbing algorithm is prone to getting stuck in these. Nevertheless, it is a simple algorithm to implement, and is often used as a baseline for optimization problems.

The behaviour of the hill climbing algorithm is largely deterministic, with only the initial solution and the order in which neighbours are considered being random, and it does not have any tweakable hyperparameters.

Simulated Annealing

Simulated annealing can be seen as an improved hill climbing algorithm. As discussed above, the hill climbing algorithm is prone to getting stuck in local optima, since it never accepts a solution that is worse than the current solution. Simulated annealing will accept any evaluated neighbour solution that is better than the current one, just as in hill climbing. However, it will also have a small chance of selecting the neighbour solution even if it has a worse score than the current solution. In the evolutionary computation framework, this change in the survivor selection mechanism is the only difference between hill climbing and simulated annealing. Algorithm 2 shows pseudo-code for the simulated annealing algorithm, with the problem formulated as a minimization problem.

Algorithm 2: Simulated Annealing

```
Data: temperature schedule $T$
Result: solution $s$
1  $s \leftarrow$ generate random solution
2  $f \leftarrow$ evaluate $s$ using evaluation function
3  for $N$ iterations do
4      $s_1 \leftarrow$ pick a random neighbour of $s$
5      $f_1 \leftarrow$ evaluate $s_1$ using evaluation function
6      $\Delta E \leftarrow f_1 - f$
7      $r \leftarrow$ select a random number between 0 and 1
8      $T_1 \leftarrow$ use $T$ to calculate current temperature
9      if $f_1 < f$ OR $r < \frac{1}{1+\exp(\frac{\Delta E}{T_1})}$ then
10         $s \leftarrow s_1$
11         $f \leftarrow f_1$
12     end
13 end
```

The probability of selecting a worse solution is given by a selection function, which takes into account how much worse the new solution is, together with a temperature parameter T_1. In Algorithm 2, the following selection function is used:

$$P = \frac{1}{1 + \exp(\frac{\Delta E}{T_1})}, \tag{2.3}$$

where P is the probability of selecting the new solution, $\Delta E = \textit{new\_score-old\_score}$ is the difference between the score of the old and new solution, and T_1 is the temperature. A cooling schedule T is typically applied to T_1, lowering its value as the optimization process progresses. Consequently, the algorithm will move more randomly in the initial stages of the run, while in later stages it will move more determinedly towards the nearest optimum. This process is inspired by the natural process of annealing, in which metal or glass is heated to high temperatures, and then gradually allowed to cool. [Russell and Norvig, 2016, p. 125]

The main parameter of simulated annealing is the temperature and cooling schedule, determining the amount of stochasticity in the search.

Genetic Algorithms

The genetic algorithm is a typical evolutionary algorithm. As opposed to the hill climbing and simulated annealing algorithms, the genetic algorithm evolves a population of several candidate solutions. Usually, both parent and survivor selection mechanisms are applied, and for the variational operators, both crossover and mutation are used. The population consists of encoded solutions, named the genotypes, that can be decoded into the actual solutions, named the phenotypes.

Algorithm 3: Genetic Algorithm

```
   Data: population size $q$, mutation rate $m$, crossover rate $c$, elite size $e$,
         generations $N$
   Result: approximately optimal solution $s$
 1 $P \leftarrow$ initialize a population of size $q$
 2 for $N$ generations do
 3     evaluate and rank $P$
 4     $E \leftarrow$ select the $e$ best solutions from $P$
 5     $S \leftarrow$ select $q \cdot c$ individuals from $P$ for reproduction
 6     $S \leftarrow S \cup E$
 7     $C \leftarrow$ apply crossover to pairs of individuals from $S$ to produce offspring
 8     apply mutation to the individuals in $C$ using the mutation rate $m$
 9     $P \leftarrow$ select $q - e$ survivors from $P \cup C$
10     $P \leftarrow P \cup E$
11 end
12 $s \leftarrow$ the best individual in $P$
```

Algorithm 3 shows an implementation of a genetic algorithm with elitism, which means that the best solutions of each generation are always carried over to the next. The first step is the generation of the initial population. This can be done randomly, through the application of domain-specific heuristics, or with trivial solutions. Evolution is then performed through iterative application of the genetic operations.

For each generation, the current population is evaluated and ranked according to the evaluation function. The set of parents for crossover are then selected based on the parent selection mechanism. An elite consisting of the e best solutions is also determined, and added to the set of parents. The parents are arranged into pairs.

Next, crossover is applied to each pair of parents, producing a pair of offspring. A variety of crossover operations exist, and it can be tailored to implement domain-specific heuristics. However, two normal variants are single-point crossover, in which both parents are split at a randomly selected point and the tails exchanged, or uniform, in which each gene is randomly assigned to one or the other of the offspring. There also exists a variation of single-point crossover named n-point crossover, in which multiple crossover points are selected. The crossover operation builds on the idea that certain features of candidate solutions may be beneficial, and that these can be combined through the combination of the parents. As such, it should be constructed so that it facilitates the transfer of good properties between candidates.

After the crossover is finished, mutation is applied to the offspring. As with crossover, there are a number of different mutation types, and they may be domain specific. However, a typical implementation is the random selection of a new value for each gene with probability m. Mutation is usually applied in order to increase the diversity of explored solutions, counteracting premature convergence.

Finally, after the variational operations are applied, the survivor selection mechanism determines which of the candidates are carried over into the next generation. With elitism, the elite is guaranteed to be included, and the selection mechanism therefore selects $q - e$ of the candidates.

The selection mechanisms are usually either rank-based, where sampling is performed based on a distribution over the candidates in order of their fitness, or tournament-based, in which the best candidate from a group of randomly sampled candidates are selected. It is also possible to apply fitness-proportionate selection, where the distribution is directly based on the fitness score of the candidates.

Genetic algorithms can be implemented in a variety of different versions, with the addition or removal of certain features. One notable example is elitism, as applied in the implementation explained above. Some implementations also use either just crossover or just mutation. However, the general structure of the algorithms are similar.

Important parameters in a genetic algorithm are the population size, crossover and mutation rates, and if elitism is applied, the elite size. In addition, the genotype representation is important, as it determines how the crossover and mutation operations will impact the solution, and how much of the real solution space is reachable by the algorithm.

2.2 Literature Review Protocol

With the goal and research questions in mind, a set of inclusion criteria that would identify relevant papers was formulated. The main concerns of the thesis can be split into two categories: automating device placement for deep learning, and modelling the performance of deep neural network. However, the field of automated device placement is still quite young and small, and to the best of my knowledge, little work has been done on applying evolutionary computation to this problem. Therefore, work done on applying evolutionary computation to process scheduling, and specifically multiprocessor scheduling, which is a highly related field for which evolutionary computation has been successfully applied, was also included.

The inclusion criteria were as follows:

Inclusion Criteria

IC1 The main concern of the paper is one of: 1. the automation of training configuration for deep learning, 2. the use of evolutionary computation for multiprocessor scheduling, or 3. performance modelling of deep neural networks.

IC2 The paper is from a primary study.

IC3 The paper presents a novel method, or a novel variation of a previously published method.

In addition to evaluating the relevance of the papers using the inclusion criteria, the quality of the papers was evaluated against the following quality criteria:

Quality Criteria

QC1 There is a clear statement of the aim of the research.

QC2 The study is put into the context of other research.

QC3 System and algorithm design decisions are justified.

QC4 Where applicable, the test data set is reproducible.

QC5 The study algorithm is reproducible.

QC6 The experimental procedure is thoroughly explained and reproducible.

QC7 It is clearly stated in the study what other algorithms or methods the algorithm(s) or method(s) of the study have been compared with.

QC8 The performance metrics used in the study are explained and justified.

QC9 The test results are thoroughly analysed.

QC10 The test evidence supports the findings that are presented in the paper.

The criteria were individually and qualitatively applied to each paper, with papers deemed of insufficient quality being excluded from the background literature.

This master's project is based on an initial proposal provided by the supervisors from Graphcore, along with which a set of relevant papers were provided. From these, the papers *Device Placement Optimization with Reinforcement Learning* [Mirhoseini et al., 2017] and *Beyond Data and Model Parallelization for Deep Neural Networks* [Jia et al., 2018b] were included. Especially the paper by Mirhoseini et al. proved to be an important formative paper for the field of automatic device placement. Therefore, papers citing this paper were considered for inclusion.

The search for papers citing Mirhoseini et al. [2017] was carried out using Google Scholar. From this set of papers, the papers *A Hierarchical Model for Device Placement* [Mirhoseini et al., 2018], *Spotlight: Optimizing Device Placement for Training Deep Neural Networks* [Gao et al., 2018], *Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning* [Addanki et al., 2019], *GDP: Generalized Device Placement for Dataflow Graphs* [Zhou et al., 2019], and *Simulating Performance of ML Systems with Offline Profiling* [Huang et al., 2020] were included.

Many of the papers mentioned until now use reinforcement learning. However, the application of evolutionary computation to the device placement problem was considered equally relevant, and became a chosen focus of this project. Searches were therefore conducted in Google Scholar for "device placement evolutionary computation" and "device placement evolutionary algorithms", yielding no relevant results. This indicates that the application of evolutionary computation to this problem is largely unexplored.

The search was then expanded to include applications of evolutionary computation to similar problems, as reflected in IC1. Since this is a field in which a lot of work has been done, the search was constrained to the application of genetic algorithms to the problem, as the genetic algorithm had been chosen as the main candidate for this project. A search was conducted in Google Scholar for "genetic algorithm multiprocessor scheduling". From the results of this search, the papers *A Genetic Algorithm for Multiprocessor Scheduling* [Hou et al., 1994], *An Incremental Genetic Algorithm Approach to Multiprocessor Scheduling* [Wu et al., 2004], and *Scheduling Multiprocessor Tasks with Genetic Algorithms* [Corrêa et al., 1999] were included.

Parallel to the master project, I participated in a course on evolutionary computation, wherein the MAP-elites algorithm was discussed. This algorithm was deemed a relevant candidate for this project, due to its ability to produce multiple diverse solutions of good quality. The algorithm was introduced in the paper *Illuminating search spaces by mapping elites* [Mouret and Clune, 2015], which has been included as related work.

2.3 Related Work

In this section, the main works in fields relating to this project are presented. The works are coarsely grouped according to the main topics of the papers, with Section 2.3.1 presenting works that explore automated solutions to the device placement problem, Section 2.3.2 looking at papers that explore performance modelling of deep neural networks, Section 2.3.3 presenting papers that explore the transfer of device placement policies to networks that were excluded from the training set, and Section 2.3.4 presenting work done on the application of evolutionary computation to the related multiprocessor scheduling problem. However, there is significant overlap between the groups.

2.3.1 Device Placement Optimization

Mirhoseini et al. [2017] introduced the use of reinforcement learning for automated optimization of device placement for a computational graph. They used a sequence-to-sequence recurrent neural network to encode placement of a set of operation groups on a set of devices. The execution time of the produced placement, as evaluated by training the placement for a few steps on the actual hardware, was used as a reward signal, and the network was trained using the REINFORCE policy gradient method [Williams, 1992]. Since this method becomes prohibitively expensive when the number of operations is large, operations were manually placed in co-location groups based on a set of heuristics. The authors reported speed-ups of up to 23.5% over expert-designed placements on a set of well-known models. However, the time required to create the placement was significant. Moreover, the use of manually-created co-location groupings limited the effective search space. I will refer to this method as *ColocRL.*

Building on ColocRL, Mirhoseini et al. [2018] introduced an extra network to the model, creating a hierarchical approach. The new network was a feed forward network that learnt effective groupings of operations, removing the need for manually created co-location groups. Embeddings for each group were then created and passed into a sequence-to-sequence RNN similar to the one used in ColocRL. The networks were jointly trained using REINFORCE. Experiments showed that the hierarchical model performed at least as well as expert-designed placement on all but one network architecture. Moreover, the hierarchical approach outperformed ColocRL on the same network trained on a set of slower processors than the ones used by ColocRL. However, the method still required running the network on the actual hardware for each evaluated placement, binding up valuable computation time. I will refer to the method introduced in [Mirhoseini et al., 2018] as *HierarchicalRL.*

Gao et al. [2018] also built on the work done with ColocRL, focusing on improving it by applying proximal policy optimization, which is a newer reinforcement learning algorithm than the REINFORCE algorithm used in the earlier paper. Another contribution was a mathematical formulation of the device placement problems as a Markov decision problem, which in turn allowed the authors to offer a proof of guaranteed improvements in the device placement problem. Building on the mathematical foundations, they presented the *Spotlight* algorithm, which iteratively maximized the performance lower bound of the device placement. The algorithm used co-location groupings created with the same heuristics as ColocRL. The experiments showed an improved execution time of the placements produced by Spotlight when compared with both expert placements, placements produced by a graph partitioning algorithm combined with a cost model for execution time, and placements produced by ColocRL.

2.3.2 Performance Modelling

A problem with an end-to-end approach for finding efficient training configurations is the substantial amount of time required in order to evaluate each proposed configuration. Jia et al. [2018a] introduced a modelling approach in order to alleviate this problem. The algorithm took a computational graph and a device graph as input, and produced an assignment of operations to devices. Evaluation was carried out by benchmarking individual operations on the different devices, and simulating data transfer between devices by using the bandwidth of the interconnects. A dynamic programming approach was used to find an optimal solution given the simulated run-time costs. Another contribution of the paper was the introduction of a more comprehensive search space, allowing parallelization of individual layers of the neural network across all dimensions, including *sample*, *width and height*, and *channel*. Due to difficulties related to implementing such parallelization in existing deep learning frameworks, the authors opted to implement their system in a low-level parallelization framework called Legion [Bauer et al., 2012]. Experiments showed improvements in training throughput over pure data parallelism, a trivial model parallelism technique, and Krizhevsky's *One Weird Trick* [Krizhevsky, 2014]. I will refer to this approach as *OptCNN*.

Building on OptCNN, Jia et al. [2018b] further expanded and formalized the search space used in finding optimal device placements. They dubbed the search space *SOAP*, which included the search space used by OptCNN in the form of the *Sample*, *Attribute*, and *Parameter* dimensions, as well as allowing cross-layer parallelism in the *Operator* dimension. The authors showed that this search space includes previous approaches as special cases. In order to allow searching over such a large space, the authors employed an execution simulator built on the same principles as the one used in OptCNN. A Markov-chain Monte Carlo method, namely the Metropolis-Hastings algorithm [Hastings, 1970] was used to explore this search space. Experiments showed improvements in throughput over pure data parallelism and expert-designed approaches over a variety of architectures. The approach was also shown to outperform ColocRL on Inception v3 [Szegedy et al., 2016] and GNMT [Wu et al., 2016], and OptCNN on the Inception v3, GNMT, RNNLM [Zaremba et al., 2014], and RNNTC [Kim, 2014] networks. I will refer to this method as *FlexFlow*.

Qi et al. [2016] presented *Paleo*, a performance model tailored for deep neural networks. Paleo relied on estimations of the number of floating point operations required by each operation in the network, dividing this by the peak floating point operations per second carried out by the respective processing unit to get the execution time of the operation. It also applied a model of communication time based on link bandwidths, with models of several implementations of the MPI allreduce operation, which is frequently used for synchronization of network parameters in data parallel settings. This allowed Paleo to make predictions of the execution time of a variety of network architectures on any number of workers of a given type connected with a given network. The main focus of Paleo was modelling how networks scale with data parallelism, but estimation of a limited class of model parallel configurations was also possible – namely, parallelization of convolutional layers across the channel dimension. The experiments showed remarkably accurate predictions when compared with actual run times of the given networks and configuration in the TensorFlow framework.

Huang et al. [2020] proposed using a benchmarking approach similar to the one used for OptCNN and FlexFlow, where individual operations are benchmarked while communication is simulated, combined with archiving of the results. Furthermore, they suggested that such an archive of benchmarking results can be crowd-sourced, with users being able to contribute results from their given hardware configuration and software versions. Experiments showed impressive accuracy, with estimated training time of VGG-19 [Simonyan and Zisserman, 2014], Resnet50, and ResNet152 [He et al., 2016] being within 2% of real execution time. However, building a comprehensive archive would require significant effort, and the chance of specific configurations not being present in the archive means that access to the training hardware would still be required.

2.3.3 Transfer Learning of Device Placement

Addanki et al. [2019] proposed a method for learning generalizable device placement strategies, named *Placeto*. By producing embeddings of the computation graphs, Placeto could generalize to previously unseen graphs. A neural network was trained to produce device placements based on given graph embeddings. Experiment results showed that placements produced by Placeto on previously unseen computation graphs were almost as good as the ones produced by Placeto when trained specifically to optimize the placement for that particular graph, while requiring significantly less time to produce the placement. It should be noted, however, that even in the generalized case, Placeto trained a model for that particular hardware configuration, and any hardware changes would require training a new model.

Addanki et al. also utilized an execution simulator in Placeto in order to reduce optimization time, similar to the ones used in OptCNN and FlexFlow. The simulator initially identified all operations of the network, profiling them by measuring their run time on all available devices. As such, access to the hardware that the network would be trained on was required. The simulation was then performed using an event based system, with events for all tasks related to executing the network handled in the order of execution. The execution time of individual operations was estimated using the times from the profiling stage, while communication times were estimated by dividing the transferred data by the link bandwidth, assuming full utilization of the channel.

Zhou et al. [2019] also looked into generalizable device placement, in a system named *GDP*. The system combined a graph neural network, producing embeddings for a variety of computational graphs, with a placement network, with both networks being trained end-to-end. As opposed to HierarchicalRL and Placeto, which both combined a network for grouping operations with an LSTM network, GDP utilized an attentive Transformer-based network [Vaswani et al., 2017; Dai et al., 2019]. This type of network is better at capturing long-term dependencies, removing the need for the grouping of operations. Moreover, it enables prediction of the full placement of a graph in a single time step, making for faster training. Experiments tested GDP trained both for a single graph and batch training of multiple graphs, with both approaches showing similar or better results than HierarchicalRL for all tested graphs. Results also showed that for some graphs, the model that was trained on a batch of graphs performed better than the one trained only on the specific graph. Experiments on generalized predictions for graphs not in the training set showed only slightly worse performance than HierarchicalRL, with little difference between predictions made with and without fine-tuning to the relevant graph.

2.3.4 Evolutionary Computation for Process Scheduling

To the best of my knowledge, evolutionary computation has not been applied to the device placement problem. However, there is relevant work done on the application of evolutionary algorithms to similar problems. In particular, the multiprocessor scheduling problem, in which a set of partially interdependent tasks are scheduled onto a set of heterogeneous processors, is very similar. The individual operations of a neural network can be considered tasks, and the data flow through the network constitutes dependencies between the tasks. In fact, the multiprocessor scheduling problem is a more complex problem than the device placement problem, as the order of tasks must be considered. For neural networks, this is typically handled by a deep learning framework.

Hou et al. [1994] proposed a genetic algorithm approach to minimizing the makespan for a multiprocessor scheduling problem. They used an encoding of the solution as a list of lists, with each sublist containing the processes to be executed on a given processor, in the order that they would be executed. Initialization, crossover and mutation procedures were designed so as to preserve order constraints within the processes executing on each processor. This was done by ordering the list by *height*, defined as the number of predecessors in the task graph. This restriction made parts of the solution space inaccessible. For the crossover operation, a random height was chosen, and the tails of equal processor lists with height higher than this number were exchanged between the chromosomes. The mutation operator would select a task at random, find another task in the chromosome with the same height, and exchange the two tasks. Ordering constraints between processors were handled in the calculation of the fitness, which was defined as the inverse of the makespan when these constraints were satisfied. The fitness function did not consider communication costs. In the experiments, the algorithm was unable to find the optimal solution to the test problems, but it was within 10% of the optimal solution across a variety of task graphs.

Corrêa et al. [1999] built on the work done by Hou et al., removing the restrictions which made parts of the search space unreachable. In order to achieve this, they removed the restriction that tasks must be scheduled in order of increasing height. For initialization, tasks were picked at random from the set of tasks to which all predecessors had been scheduled, and assigned to a random process, until all tasks were assigned. For crossover, tasks were divided into two sets V_1 and V_2. For V_1, all predecessors of any task were guaranteed to be contained in the set. The tasks in V_2 were then rescheduled according to a list heuristic based on their ordering in the other parent. For mutation, the entire chromosome was regenerated based on similar heuristics. A second version of the algorithm was also implemented, integrating knowledge-based heuristics in the crossover and mutation operations. Experiments showed improvement over the algorithm developed by Hou et al. for both implemented versions, with the knowledge-enhanced version performing best overall. This showed that both the increased search space and the integration of knowledge-based heuristics improved the quality of solutions. However, this came at an increase in computational cost.

Wu et al. [2004] introduced a different approach to applying genetic algorithms to the multiprocessor scheduling system. Instead of selecting a representation that was guaranteed to be valid, they considered validity when calculating the fitness score. Moreover, they introduced a dynamic fitness score, which would increase the difficulty of the problem as the search progressed. Individuals were represented as a list of tuples, with each tuple assigning a task to a processor, and the order of tuples defining the ordering of tasks. Simple 1-point crossover was used, while mutation consisted of randomly selecting a new value for either the task number or processor number of a tuple. The fitness score was a weighted sum of the validity and makespan of the solution. Validity was defined as a combination of the fraction of tasks present in the solution, and the fraction of valid sequences of a given length. The gradual increase in problem difficulty was achieved by considering incrementally longer sequences, until finally the whole individual represented a valid solution. Experiments compared the genetic algorithm with heuristic algorithms for solving the multiprocessor scheduling problem, and showed that the genetic algorithm generally performed equally good or better. Moreover, the quality of solutions produced by the genetic algorithm on more complex problem instances was higher, indicating more flexibility. However, the computational cost of the genetic algorithm was considerably higher than the heuristic methods.

It can often be beneficial to produce more than one good solution to a problem. For the device placement problem, this is relevant when using a simulator for optimization, and then transferring the solution to the training hardware. Having multiple solutions allows the selection of the solution that works best in the real application.

Mouret and Clune [2015] proposed a technique to this end, named MAP-Elites. The algorithm preserves an archive of solutions, each of which are the best solution of their respective niche, with niches defined by pre-selected phenotypic properties of the solutions. For each step, a single solution is selected from the archive, mutated, evaluated, and placed into its corresponding niche if it outperforms the niche's current elite solution. The result of this process is a set of quality solutions that are guaranteed to be diverse according to pre-selected phenotypic properties. Experiments showed that the MAP-Elites algorithm yielded better coverage of the solution space than a traditional evolutionary algorithm, multi-objective search against novelty and local competition scores [Lehman and Stanley, 2011], and random sampling. Moreover, it also outperformed these algorithms with respect to the quality of the solution.

2.4 Motivation

With the size of modern deep learning models, the problem of partitioning neural networks across multiple devices is a prevalent one. Moreover, the complexity of this task constitutes a palpable impediment for the development and application of such models. The automation of this process would therefore be advantageous.

As presented in Section 2.3, there have been numerous attempts at handling this problem. Multiple approaches rely on end-to-end reinforcement learning, requiring an extensive number of expensive network evaluations in order to find a good placement. This leads to a time-consuming process that negatively impacts the combined time of finding a placement and training the network. Some works have explored the use of benchmarking-based simulators, greatly reducing the amount of direct evaluations required. However, such approaches still require access to the hardware itself – access that may be a valuable resource, and which is often a shared commodity. Building databases of benchmarking results can alleviate this, but there will still occasionally be operations or hardware configurations that have not been benchmarked. As such, the dependency on the hardware is still present. Recent works have also explored the training of generalizable neural networks that perform device placement. However, these networks are tuned to a specific hardware configuration, and therefore require a new model to be fitted to each computer system that training will be run on.

Through the application of an execution modelling approach, the optimization time can be greatly reduced over end-to-end approaches, and the optimization can be carried out without access to specialized hardware. Thus it becomes possible to optimize the placement of the network on a commodity computer before committing the training process to expensive, specialized hardware. Consequently, more computation time on the specialized hardware can be put towards training deep models themselves.

Reinforcement learning has been the prevalent method for device placement optimization. Jia et al. also explored dynamic programming [Jia et al., 2018a] and local search in the form of the Metropolis-Hastings algorithm [Jia et al., 2018b]. However, methods from the field of evolutionary computation have to the best of my knowledge not been applied to this problem. Such methods should be highly capable of performing this type of optimization – supported by their success in related fields such as multiprocessor scheduling, and may possibly even provide improved results over previously tested methods.

Chapter 3

Method and Technology

This chapter gives a detailed explanation of the system implemented for this project, along with a brief overview of central technologies that have been utilized. Section 3.1 describes the execution simulator for neural networks, with a detailed, step-wise explanation in Section 3.1.1. Section 3.2 presents the two optimization algorithms, with Section 3.2.1 describing their shared encoding of candidate solutions, Section 3.2.2 detailing the implementation of the genetic algorithm, and Section 3.2.3 providing the same description for the MAP-Elites algorithm. Section 3.3 gives a brief overview of the PyTorch framework for deep learning, which is used for benchmarking purposes. Finally, Section 3.4 gives a summary of this chapter.

Parts of this section has been adapted from a previous report in the master project [Andreassen, 2019]. The section about the execution simulator is largely the same as in the previous report. The sections about the solution encoding and the genetic algorithm are based on the corresponding sections in the previous report, but have been rewritten.

Both the execution simulator and the optimization algorithms are available as open source [Andreassen, 2020a].

3.1 Execution Simulator

The execution simulator is a key part of the optimization system, enabling estimation of the runtime for a given computation graph in a timescale that is orders of magnitude faster than doing a benchmark run. It runs independent of both any specific deep learning framework and the simulated hardware itself.

The execution simulator is based on previous work done by Addanki et al. [2019] on *Placeto*, and Qi et al. [2016] on *Paleo*, with a majority of the simulator consisting of a combination of the two. Specifically, the general approach for simulating the scheduling of operations and communication is inspired by Placeto, while code from Paleo, which is open source, is used to estimate individual run times of operations and communication times of single data transfers.

The simulator supports convolutional and deconvolutional layers, pooling layers, fully connected layers, and dropout layers. Other layer types are modelled as requiring no computational time. As this set of layers constitutes the majority of computation in a convolutional neural network, with other operations requiring negligible computational time, this makes the simulator suitable for simulating CNNs as well as fully connected networks. However, it does not support recurrent neural networks.

In order to enable the execution time simulation, a number of assumptions are made:

A1 The run time of an individual operation is equal to the number of floating point operations required by the operation divided by the number of floating point operations carried out per second by the relevant processor, yielding perfect utilization of the processor.

A2 The transfer time of an object over a communication channel is equal to the size of the transmitted object divided by the bandwidth of the communication channel, yielding perfect utilization of the communication channel.

A3 Both processors and communication devices carry out operations according to the first in – first out principle.

A4 There is no time delay in neither processors nor communication channels. An operation is processed by the processor as soon as it is ready, and an object is transferred by a communication channel as soon as it is available and the channel is free.

It should be noted that the system implements mechanisms for relaxing some of these assumptions, such as the ability to introduce a penalty to execution and communication times for relaxation of A1 and A2, respectively.

3.1.1 Detailed Description of the Execution Simulator

Figure 3.1 shows a high-level overview of the most important components of the execution simulator, while Algorithm 4 describes the simulation process. The entire system is event based, with three types of events:

wakeup Triggered when a computational device or communication channel that was previously idle is tasked with either executing an operation or transferring a tensor, respectively.

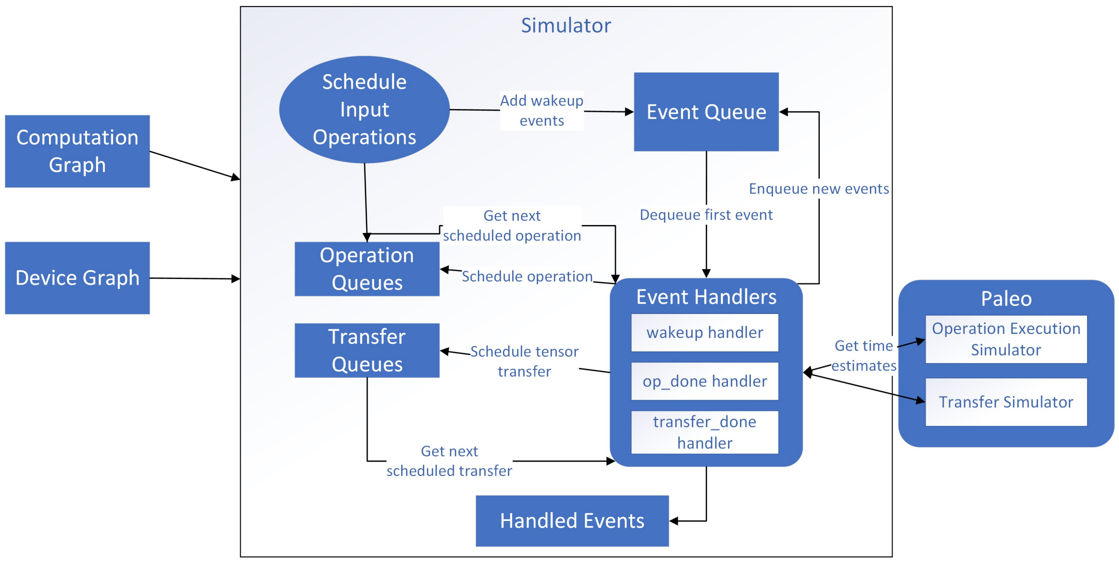


Figure 3.1: High-level overview of the execution simulator.

Algorithm 4: Execution Simulator

```
   **Data:** Computation Graph $C$, Device Graph $D$
   **Result:** Execution Time $T$, Event Trace $E$
 1 $Q \leftarrow$ initialize priority queue
 2 $O \leftarrow$ initialize FIFO queues for each device in $D$
 3 $C \leftarrow$ initialize FIFO queues for each communication channel in $D$
 4 for each input operation $i$ in $C$ do
 5     $d \leftarrow$ device of $i$
 6     if $O_d$ is empty then
 7         add new wakeup event to $Q$
 8     end
 9     insert $i$ into $O_d$
10 end
11 while $Q$ is not empty do
12     $e \leftarrow$ dequeue first event from $Q$
13     handle $e$ using corresponding event handler, described in Algorithms 5,
         6 and 7
14     add $e$ to $E$
15 end
16 $T \leftarrow$ end time of last event in $E$
```

op_done Triggered when an operation has finished executing on a device.

transfer_done Triggered when a tensor has finished transferring over a communication channel.

Each event is saved with metadata about the event type, timestamp that the event started and was finished, which batch the event was part of, which operation that was executed/which tensor that was transferred, which device that it was executed on/transferred over, and which devices it was transferred to and from, if applicable.

Central to the execution of the simulator are a number of queues. For each device, an operation queue is maintained, while a similar transfer queue is kept for each communication channel. Both of these queue types are FIFO queues, implementing the assumptions made in A3. There is also a single event queue, which is a priority queue placing events with the lowest end timestamps first, ensuring that they are handled in the order of execution.

The inputs to the simulator are a computation graph, defining the network that is to be simulated, and a device graph, defining the hardware that we want to simulate execution of the network on. Each operation in the computation graph is assigned to a specific device in the device graph. During initialization, the simulator determines which operations in the computation graph are input operations, defined by not having any parent operations. These are scheduled on their respective devices by enqueuing them in the device's operation queue. The initialization procedure also adds *wakeup* events for all relevant devices to the event queue.

After initialization, the simulator continuously removes the first event from the event queue, and passes it on to the relevant event handler. Since the event queue is prioritized by the end time of the events, this ensures that the events are handled in the order that they are executed. Algorithms 5, 6 and 7 describe the different event handlers. The *wakeup* handler simply dequeues the first waiting operation or transfer for the relevant device, uses Paleo to estimate the time required by the operation or transfer, and creates the corresponding *op_done* or *transfer_done* event. The *op_done* and *transfer_done* event handlers both go through all children of the executed operation or transferred tensor, checking whether they are ready to be executed or whether the output of the operation must be transferred to another device. Operations and transfers are correspondingly scheduled on the operation and transfer queues. The *op_done* handler then checks if there are any waiting operations in the device operation queue, and either immediately executes the first operation, adding the corresponding *op_done* event, or marks the device as free. The *transfer_done* handler does the corresponding actions with the transfer queue.

The simulator continues to handle events until the event queue is empty, at which point the simulation has reached its conclusion. During the simulation, each handled event is appended to a list, resulting in a complete event trace when the simulation finishes. The simulated execution time of the computation graph is simply the end time of the last event in the list, but the event trace enables the calculation of peak memory usage as well as visualization and analysis of the execution of the computation graph. It should be noted that the memory estimation provided by the event trace only takes into consideration the weights of all layers and the saving of activations and gradients. When performing the real training process, there are typically other operations that need to keep track of state, such as optimizers using momentum. The simulator allows limiting the usable fraction of the device memory in order to make room for such data, which is useful when running optimization for a device architecture with limited memory.

Multiple batches can be executed by the simulator at once, which allows simulation of a simple pipeline parallelism. The simulator will function in the exact same way, with operations and events created for all batches. Events and operations are tagged with the batch number that they belong to. The scheduling mechanism is very simple: the system will schedule the execution of a new batch when all the devices of the input operations are idle.

Algorithm 5: *wakeup* Event Handler

```
Data: event $e$, operation queues $O$, communication queues $C$, event
      queue $Q$
1  if subtype of e is transfer then
2      $d \leftarrow$ get transfer device from $e$
3      $t \leftarrow$ dequeue first transfer object from $C_d$
4      $T \leftarrow$ estimate transfer time of $t$ using Paleo
5      add new transfer_done event to $Q$ with start time = end time of $e$, end
         time = start time + $T$
6  else if subtype of e is operation then
7      $d \leftarrow$ get device from $e$
8      $o \leftarrow$ dequeue first operation from $O_d$
9      $T \leftarrow$ estimate execution time of $o$ using Paleo
10     add new op_done event to $Q$ with start time = end time of $e$, end time
         = start time + $T$
11 end
```

Algorithm 6: *op_done* Event Handler

```
Data: event $e$, operation queues $O$, communication queues $C$, event
      queue $Q$
1  $o \leftarrow$ get operation from $e$
2  $d \leftarrow$ get device of $o$
3  for each child $c$ of $o$ do
4      $d_c \leftarrow$ get device of $c$
5      if $d_c \neq d$ then
6          if transfer of the relevant tensor to $d_c$ is not already scheduled then
7              $cc \leftarrow$ get communication channel between $d$ and $d_c$
8              enqueue transfer to $C_{cc}$
9              add wakeup event if $cc$ is free
10         end
11     else if all inputs of $c$ are available on $d$ then
12         enqueue $c$ to $O_d$
13     end
14 end
15 if $O_d$ is non-empty then
16     $o \leftarrow$ dequeue first operation from $O_d$
17     $T \leftarrow$ estimate execution time of $o$ using Paleo
18     add new op_done event to $Q$ with start time = end time of $e$, end time
         = start time + $T$
19 else
20     mark $d$ as free
21 end
```

Algorithm 7: *transfer_done* Event Handler

```
Data: event $e$, operation queues $O$, communication queues $C$, event
      queue $Q$
1  $t \leftarrow$ get transferred tensor from $e$
2  $d \leftarrow$ get target device from $e$
3  $cc \leftarrow$ get communication channel from $e$
4  for each child $c$ of $t$ do
5      if all inputs of $c$ are available on $d$ then
6          enqueue $c$ to $O_d$
7          add wakeup event if $d$ is free
8      end
9  end
10 if $C_{cc}$ is non-empty then
11     $t \leftarrow$ dequeue first tensor scheduled for transfer from $C_{cc}$
12     $T \leftarrow$ estimate transfer time of $t$ using Paleo
13     add new transfer_done event to $Q$ with start time = end time of $e$, end
         time = start time + $T$
14 else
15     mark $cc$ as free
16 end
```

3.2 Optimization Algorithms

In the project, two main optimization algorithms were implemented – both being evolutionary algorithms. The first of the two is the genetic algorithm, which is a classic algorithm typical of evolutionary computation. The second is the MAP-Elites algorithm, which is a more recent advancement that allows the evolution of a set of diverse solutions that each are the elite of their own niche.

3.2.1 Solution Encoding

The encoding of solutions is essential in all evolutionary computation methods. The encoding must possess sufficient expressive power, allowing the entire search space to be represented – or at least guaranteeing that the optimal solutions can be represented. Preferably, the encoding has a one-to-one relation with the actual solutions, giving full coverage while eliminating redundancy and invalid encodings. Moreover, the encoding must properly support the operations used by the algorithms, such as mutation and crossover.

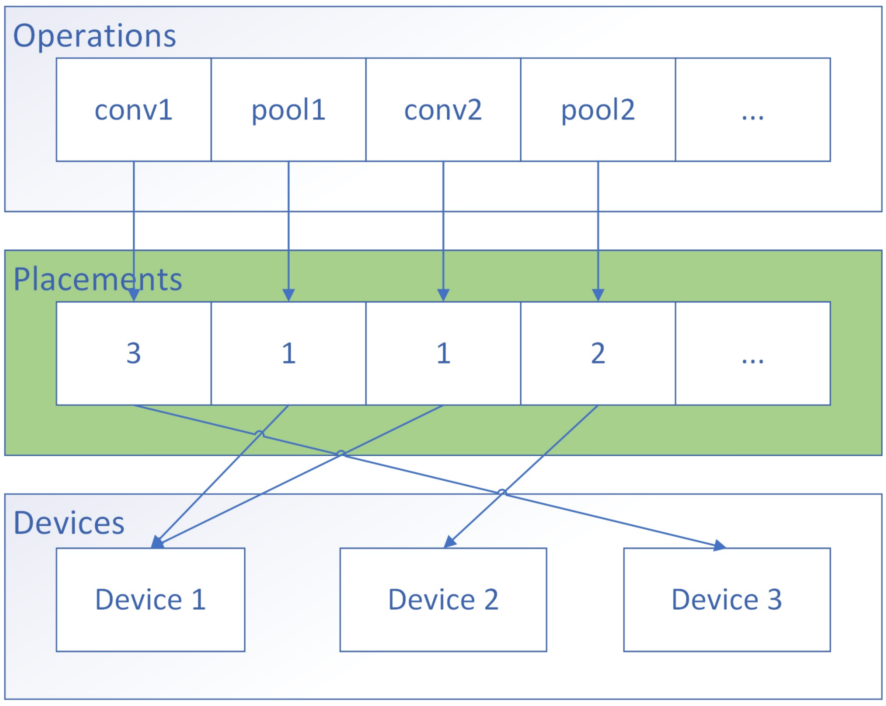


Figure 3.2: Illustration of the solution encoding used by all optimization algorithms.

All algorithms implemented in this project use the same encoding for solutions, illustrated in Figure 3.2. A solution to the device placement problem is the assignment of each operation to a single device. To produce the encoding, the operations of the neural network are sorted in topological order. The encoded solution is then a list corresponding to this order – the highlighted portion in the middle of the figure, with each element of the list being a single number assigning the corresponding operation to a processing device with that ID.

This encoding has a one-to-one relation with the solution space, guaranteeing full representability without redundancy. Moreover, since operations are sorted in topological order, dependency relations are partly represented in the encoding, with neighbouring operations tending to be placed consecutively. In fact, for linear networks, all neighbours in the network will be neighbours in the encoding. This allows crossover and mutation operations to capture solution properties related to these relations, which is highly relevant when considering communication between operations. Finally, the encoding is both simple to implement in itself, and allows for the use of standardized variational operators.

One drawback of the encoding is that it makes for a significant search space. However, this is a result of the solution space itself being considerable. As mentioned above, it is typically desirable to have full representability and a one-to-one relation. Heuristics that aid the traversal of the search space can be implemented in the variational operations.

3.2.2 Genetic Algorithm

The genetic algorithm used in the project employs a quite standard implementation with elitism, following the outline described in Section 2.1.5. It uses the solution encoding described above, and initializes the population with randomly generated individuals, optionally also including trivial solutions – i.e. solutions that only use a single device.

The fitness function for the genetic algorithm is the inverse of the execution time of the solution, with a penalty equal to the memory overflow measured in MB added to the time if the memory of any device is exceeded. This ensures that the algorithm is encouraged to firstly find solutions that fit into memory, and secondly find the most efficient solutions among these.

For the parent selection mechanism, both tournament selection and rank selection is implemented, with the rank selection supporting linear and exponential distributions, with the latter providing the most selection pressure. Survivor selection is either done implicitly by letting all offspring of crossover and mutation operations survive when no elitism is applied, or by taking the union of the elite and a random sample of the offspring if elitism is applied.

For crossover operations, both uniform crossover and n-point crossover are supported. The main mutation operator consists of randomly selecting a new device for individual genes, with the probability for each gene of this happening being determined by the mutation rate. The algorithm also employs a second mutation type customized for the problem at hand, in which a randomly selected zone of consecutive genes are all assigned to the same device. I use the name *zone mutation* for this mutation type. This adds some bias towards solutions with consecutive operations placed on the same device, which is heuristically good. However, care should be taken not to use a too high mutation rate for the zone mutation, as this will harm diversity.

Self-adaptation [Bäck, 1992] is applied to the mutation rates. This is done by placing the mutation rate into the chromosome, and evolving it along with the solution itself. For mutation, the self-adapted mutation rate is mutated before it is being used in the mutation of the solution itself. As such, the mutation rate of a fit chromosome can be interpreted as the mutation rate that created the fit solution. Mutation of the mutation rate is done by adding a value sampled from a gaussian distribution, leading to a small random change in the mutation value. Crossover is applied by calculating a weighted average. A random weight α between 0 and 1 is chosen. The new mutation rate of the first child is then $m_{c1} = \alpha \cdot m_1 + (1-\alpha) \cdot m_2$, where m_1 and m_2 are the mutation rates of parent 1 and parent 2, respectively. Conversely, the mutation rate of the second child is $m_{c2} = \alpha \cdot m_2 + (1-\alpha) \cdot m_1$. Due to the self-adaptation, the algorithm is less sensitive to the mutation rate parameters, which only determine initial mutation rates.

The algorithm applies elitism both for parent and survivor selection. Consequently, the elite, as determined by the elite size, is guaranteed both to be included in crossover operations, and in the final population of the generation. The elitism ensures that previously found good solutions are carried over into consecutive generations. Consequently, the best fitness in the population will never decrease.

3.2.3 MAP-Elites

In the original paper, the MAP-Elites algorithm is described more as a loose set of strategies than as a strict algorithm. The main idea is the use of an archive of the elite solutions in a set of niches instead of a general population, as used in the genetic algorithm.

For this project, the archive was divided into niches according to three dimensions of the phenotypic solutions: the number of devices used by the solution, the number of tensor transfers required by the solution, and the statistical mode of the devices used – i.e. the device on which the highest number of operations are executed. Intuitively, in homogeneous systems, the device mode is unimportant, as all processors provide the same performance. However, when transferring solutions to a shared system, in which some processors may experience higher load from external processes, this can be a crucial property. Moreover, it is beneficial to also support heterogeneous computer systems.

MAP-Elites processes a single solution at a time, chosen from the archive. In the original paper, this selection is performed completely at random, encouraging exploration. However, for this project, it was determined that some selection pressure was desirable – especially considering that there are large groups of solutions that are invalid due to them not fitting into the memory of the available devices. Therefore, a tournament selection mechanism was introduced. The tournament size should typically be kept low, so as to still encourage exploration, albeit to a lesser degree. It should be observed that with a tournament size of one, this selection scheme is equal to random selection.

The MAP-Elites implementation employs a number of different mutation operators. The random mutation operator, as described above for the genetic algorithm, is used as the main mutation operator, and the main source of diversity. However, three customized mutation operators that add some bias to the solutions are also used. One of these is the zone mutation described above for the genetic algorithm. The other two are the replace mutation, in which all genes for a given device is replaced with another device, and the copy mutation, in which the device of a gene is replaced with the device of the preceding gene. All of these operators add bias towards solutions using fewer devices, and solutions where consecutive operations are placed on the same device. Care should be taken not to use too large mutation rates for these mutations. However, due to the preservation of diversity in niches, this is a less pressing problem for the MAP-Elites algorithm than for the genetic algorithm.

As opposed to the original MAP-Elites algorithm, this implementation also uses crossover. The same crossover operations as for the genetic algorithm are supported. If crossover is to be applied at a given iteration, two parents are selected from the archive, and a single offspring is produced. Mutation is then applied to this individual.

After all variational operations have been applied, the candidate is considered for insertion into the archive. This is done by determining the niche of the candidate, and comparing its fitness with the fitness of the previous elite in this niche. If the new candidate is better, it replaces the current elite. Similarly, if the niche is currently empty, the candidate is inserted into the niche. However, if the new candidate is worse than the current elite of the niche, it is discarded. It is therefore important that the archive is large enough to give a decent chance of new candidates being preserved.

The implementation supports parallel execution, which can considerably speed up the algorithm. This is done by calculating several steps that would usually be performed sequentially, in parallel. Consequently, the algorithm may deviate slightly from the standard execution when running in parallel. This happens if two steps executed in parallel operate on the same niche. One step may then use a parent that is essentially outdated. However, the odds of this happening frequently is low, due to the size of the archive and the stochasticity in the selection mechanism. The advantage of parallel execution far outweighs the slightly slower convergence this may result in.

3.3 PyTorch

Whenever the execution of the neural networks on real hardware is required for benchmarking purposes, the system uses network implementations in PyTorch [Paszke et al., 2019]. PyTorch is a framework for deep learning that provides implementations of primitives such as tensors and tensor operations, as well as the normal neural network layer types – e.g. convolutions, dropout, and fully connected layers. PyTorch allows the user to specify the architecture of a neural network at a high level, along with the placement of parts of the network onto specific computational devices. PyTorch then handles fine-grained scheduling of the required operations, including transfers between devices, automatically.

3.4 Summary

The system implemented in this project consists of two parts: a set of optimization algorithms that can solve the device placement problem, and an execution simulator that can significantly speed up evaluations during this process. Two main optimization algorithms have been implemented: a genetic algorithm, and a MAP-Elites algorithm. The main difference between the two is that the MAP-Elites algorithm employs explicit niching of solutions, guaranteeing the discovery of a diverse set of solutions.

The execution simulator mimics the execution of a neural network on a given device architecture, but uses idealized models based on FLOPS and bandwidth characteristics of processors and communication channels in order to estimate the time required for individual operations, and the training process as a whole. This results in a significant speed-up, and removes the dependency on specific training hardware.

The optimization algorithms can also use benchmarking of the networks as an evaluation function. In this project, these benchmarks were carried out using PyTorch, which is a deep learning framework.

Chapter 4

Experiments and Results

A number of experiments were carried out in order to evaluate the optimization methods and the execution simulator, with a focus on answering the research questions. This chapter presents the experiments and their results, along with providing an analysis of the results. In Section 4.1, the experimental plan is outlined, with each experiment presented briefly. Section 4.2 details the parts of the experimental setup that are shared among experiments. Section 4.3 presents each experiment in detail, along with presenting the results and their analysis.

The source code for the experiments is available online [Andreassen, 2020b].

4.1 Experimental Plan

The experiments presented in this chapter are:

E1 Benchmarking the practical bandwidth of the PCIe3 bus.

E2 Measuring variance in batch times during a neural network training process.

E3 Comparing optimization methods when run against the execution simulator.

E4 Transferring solutions from simulation to real hardware during the optimization process.

E5 Comparing simulated and real batch times for different training configurations.

Experiments E1 and E2 aimed at testing some important assumptions made for the execution simulator and benchmark processes, and also giving the data needed to relax these assumptions, if necessary. In E1, the practical bandwidth of a PCIe3 bus was benchmarked at different data packet sizes, and compared to the theoretical bandwidth, to see how the communication simulation held up. In

E2, individual batch times for a neural network training process were measured, in order to see if there were any clear outliers. For example, it is conceivable that the first batch may require more time due to initialization. In such a case, some batch times may have to be dropped when doing benchmarking.

Experiment E3 aimed at evaluating the applicability of the implemented optimization algorithms to the device placement problem. The two main algorithms – the genetic algorithm and MAP-Elites, were evaluated against two simple baseline algorithms – hill climbing and simulated annealing. For the instances where this was possible, the results were also compared to the trivial solution of using a single GPU. All runs were performed against the simulator in this experiment.

Experiment E4 looked at the behaviour of the optimization process when run against the simulator as compared to running on benchmarked execution times. This was done through the transfer of candidate solutions between these two environments during training. By looking at the stability of the training process before and after this transfer, an indication of the impact of the simulator on the training process could be extracted. Due to the large number of function evaluations in the optimization process, and the computational cost of performing benchmarks, it was impractical to perform a large number of runs for this experiment. The results were therefore exclusively analysed in a qualitative fashion.

Finally, E5 endeavoured to produce a quantitative evaluation of the applicability of the simulator in the optimization process. To this end, the batch execution times estimated by the simulator, and those measured through benchmarks, were compared for a number of different placements of a set of neural networks. Through this, it could be determined whether the ordering of solutions produced by using simulated execution times was approximately correct.

4.2 Experimental Setup

The experiments presented in the reports were run on two different servers, named Malvik and Luke01. Table 4.1 and Table 4.2 present the main hardware of these servers. Both servers have two CPUs, but these are treated as a single processor in both the execution simulator and PyTorch. Therefore, they are listed as a single device, with both the number of threads and peak FLOPS reported as the total between the two CPUs. On both servers, each GPU is connected to the CPU with a dedicated PCIe3x16-bus, with a theoretical bandwidth of 128Gbit/s.

Malvik was the main server used for experiments both in the execution simulator and for real benchmarks, and was the server used in the experiments unless otherwise specified. For some experiments utilizing only the execution simulator, device graphs derived from the Malvik setup were used, with varying numbers of GPUs. For some experiments, the device memory available to the system was programmatically limited.

Table 4.1: Hardware of the Malvik server

ID	Device	Processor Clock	Peak GFLOPS	RAM
0	2x Intel Xeon Gold 6132	56 x 2.6 GHz	1800	755 GB
1	NVIDIA V100	1.3 GHz	14000	32 GB
2	NVIDIA V100	1.3 GHz	14000	32 GB

Table 4.2: Hardware of the Luke01 server

ID	Device	Processor Clock	Peak GFLOPS	RAM
0	2x Intel Xeon E5-2650 v4	48 x 2.2 GHz	800	504 GB
1	NVIDIA P100	1.2 GHz	9300	16 GB
2	NVIDIA P100	1.2 GHz	9300	16 GB

Three different neural network architectures were used in the experiments: AlexNet [Krizhevsky et al., 2012], ResNet-50 [He et al., 2016], and Inception V3 [Szegedy et al., 2016]. All networks are convolutional networks designed for classifying the ImageNet [Deng et al., 2009] dataset. AlexNet is a fairly shallow network with 8 layers, while ResNet-50 and Inception are deep networks with 50 and 48 layers, respectively.

For benchmarking purposes, implementations of the models were taken from the *torchvision* library [PyTorch, 2020], and modified so as to allow the operations of the network to be placed on specific devices on demand. Several steps of gradient descent on batches of randomly generated images were performed, and the measured time for all batches averaged to find the benchmarked time required for executing a single batch.

Two mechanisms were employed for monitoring memory consumption during benchmarking. If the physical memory of any device was exhausted, an OOM error was raised by PyTorch and then caught by the system. A constant penalty would then be applied if the benchmark was used for optimization. However, for some experiments, there was a need to introduce an artificial memory limit that was lower than the physical memory. This was done by monitoring the peak memory on all devices, and comparing it to the artificial limit after the training had finished.

The parameters of the two main optimization algorithms – the genetic algorithm and the MAP-Elites algorithm – were tuned by hand. The same parameters were used for all experiments, unless otherwise specified, and the parameters were tuned to perform well over all problem instances. In most cases, it may be possible to achieve better results by tuning the parameters specifically for the problem at hand. However, since an important part of the utility of a device placement optimizer is that it removes the need for expert knowledge, it is desirable to avoid such tuning. Replacing the need for expert knowledge about device placement with a need for expert knowledge about evolutionary computation would in this case be counter-productive. Therefore, the parameters were kept constant, so that the experiments evaluated the performance of a generalized system.

The main parameters of the genetic algorithm are given in Table 4.3, and the main parameters of the MAP-Elites algorithm are given in Table 4.4. For the genetic algorithm, the set-up was quite generic, with the exception of the zone mutation. Single-point crossover was used, and the algorithm used rank-based parent selection with a linear distribution, providing some selection pressure. Elitism was applied, with an elite size of 5. The MAP-Elites algorithm had some additional mutation operators, and accordingly a few more mutation rates. The archive size was set to be automatically calculated for the device mode and devices used dimensions, while the dimension size for the number of tensor transfers was set to 40, which reduced the size of the archive for the larger networks. Tournament selection was used, as this is quite similar to the random sampling in the original paper, but with some added selection pressure. A tournament size of 10 was used.

Table 4.3: Parameters for the genetic algorithm

Parameter	Value
Population size	50
Mutation rate	0.5
Zone mutation rate	0.2
Crossover rate	0.2
Crossover type	Single-point
Parent selection mechanism	Rank
Parent selection distribution	Linear
Elite size	5

Table 4.4: Parameters for the MAP-Elites algorithm. For the archive dimensions, -1 means that the dimension was automatically set to be the same as the number of distinct possible values for this property.

Parameter	Value
Archive dimensions	(-1, -1, 40)
Mutation rate	0.4
Copy mutation rate	0.4
Replace mutation rate	0.01
Zone mutation rate	0.05
Crossover rate	0.4
Selection type	Tournament
Tournament size	10

4.3 Experiments

4.3.1 E1 — Influence of Tensor Size on Communication Bandwidth

The simulator assumes perfect utilization of the communication channels. However, in practice, the computer is unlikely to be able to utilize the full bandwidth. Moreover, due to overheads in the communication process, the observed bandwidth may vary as a function of the size of the transferred package. An experiment was conducted in order to measure this effect.

Randomly initialized tensors of sizes in the range 10^i bytes, $i = 3, 4, ..., 9$ were transferred from the CPU to GPU 0 on Malvik and Luke, with the end-to-end transfer time being recorded. Each transfer was repeated ten times, and the resulting transfer times were used to estimate the bandwidth. Results are displayed in Figure 4.1, showing the mean bandwidth along with a 95% confidence interval.

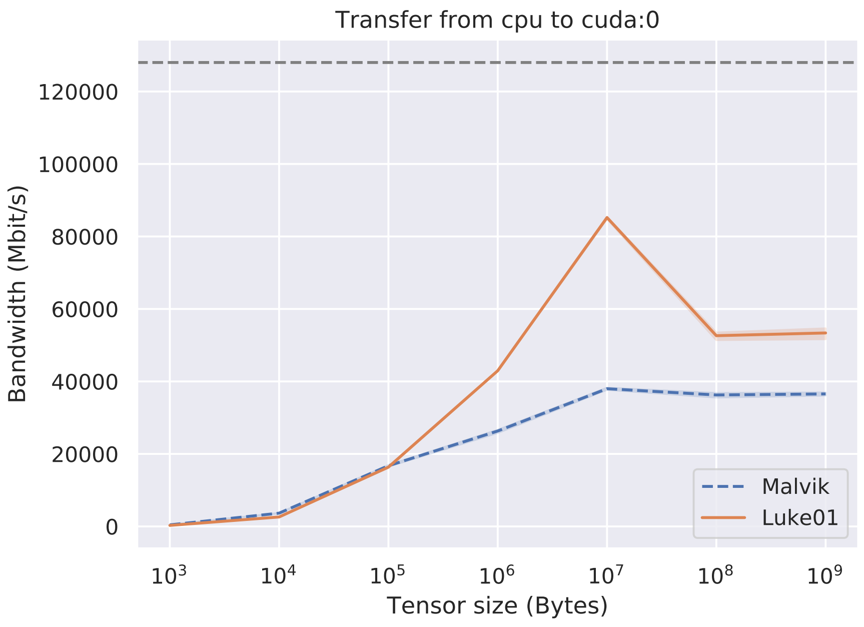


Figure 4.1: Transfer times of tensors of varying size between CPU and GPU on Malvik and Luke01 servers. Theoretical bandwidth is shown as horizontal dashed line.

As can be seen, the observed bandwidth varies significantly as a function of the tensor size. Moreover, even at the peak, the bandwidth is far lower than the theoretical bandwidth of 128 Gbit/s, marked with a dashed line. On average, the bandwidth utilization is 17% on Malvik, and 28% on Luke. The different utilization between the servers may be due to load, as these are shared machines.

Notably, the bandwidth on Luke drops off between tensor sizes of 10^7 and 10^8 bytes. No definite cause for this has been established, but it may be due to buffers filling up, necessitating the use of slower memory, in turn yielding lower utilization of the communication channel. On Malvik, which is equipped with more powerful hardware, the bandwidth flattens for similar tensor sizes.

Based on these results, a constant penalty on available bandwidth was introduced in the simulator, limiting the bandwidth used in estimation of communication costs to 25% of the theoretical bandwidth. For further accuracy improvement, the bandwidth may be made dependent on tensor size. However, this would require comprehensive benchmarking in order to create the bandwidth model for each device graph that the system should simulate.

4.3.2 E2 — Stability of batch training times

The various deep learning systems behave in different ways when it comes to how batches are processed, what types of initialization are necessary and so on. This may manifest as variations in the time required for processing individual batches, with for example the first batch requiring more time than consecutive batches.

A simple experiment was conducted to map out such effects. 75 different training configurations for AlexNet were run for 50 batches on Malvik, with the time required for each individual batch recorded.

Figure 4.2 shows the results. The run time of each configuration has been normalized by subtracting and dividing by the mean, yielding residuals as fraction of the mean run time. These results have in turn been averaged to produce the plot.

The plot shows a clear trend of a periodic drop in run time. It was determined that this likely resulted from the last batch in the data set being smaller, so a new run was performed with this batch being dropped.

Figure 4.3 shows the results of the second run. The periodic drop in batch time is now eliminated, and the magnitude of the variation is smaller. There is a tendency of the first batch of the training process requiring significantly shorter time than the following batches. Apart from this, there is only what appears to be random noise, with small magnitude – all less than 1.5% of the mean time. This shows that the first batch should be dropped when doing benchmarking. Averaging the rest of the batch times should yield a representative execution time.

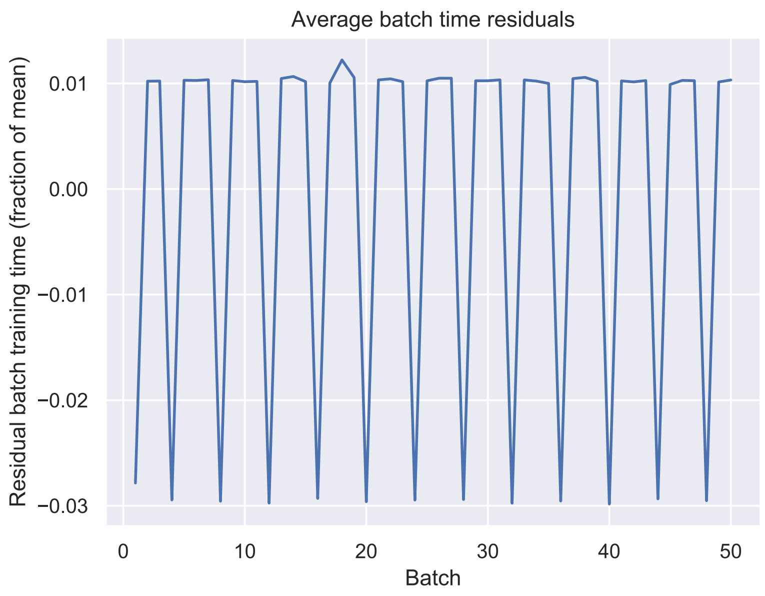


Figure 4.2: Average batch time residuals for first experiment run

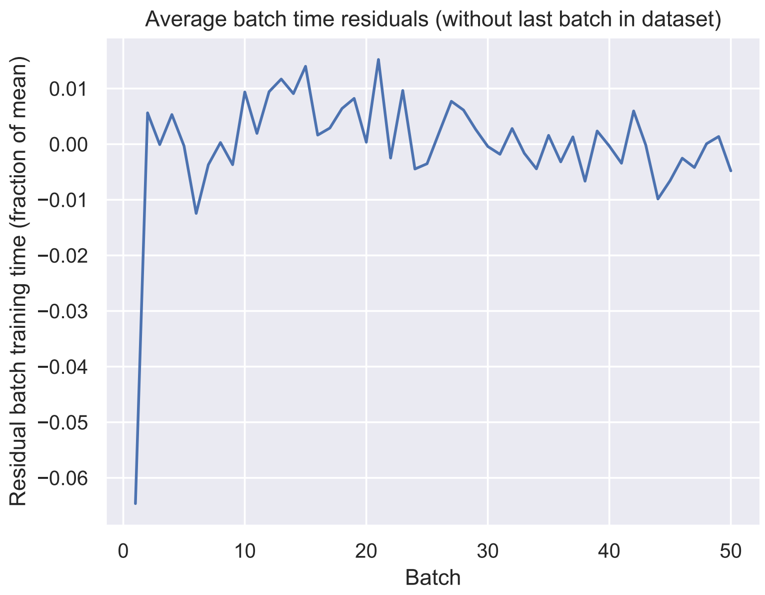


Figure 4.3: Average batch time residuals for second experiment run, with the last batch in the data set dropped.

4.3.3 E3 — Comparison of Optimization Algorithms

The following set of experiments evaluated the performance of the two main optimization algorithms developed in this project – a genetic algorithm and a MAP-Elites algorithm, against two simpler algorithms – random hill climbing and simulated annealing. The goal was to verify that the more advanced algorithms outperform a simple baseline, and exploring how the complexity of the problem affects this relationship.

The random hill climbing algorithm has no parameters, and was implemented as described in Section 2.1.5. For the simulated annealing algorithm, an implementation from the Python library *scipy* was used – namely *scipy.optimize.dual_annealing* with the *no_local_search* parameter set to True, yielding classic simulated annealing. Other parameters were kept at defaults.

In order to give a fair comparison of the quality of produced solution with the same amount of computations, the number of function evaluations was set to 20000 for all algorithms. For the hill climbing, simulated annealing, and MAP-Elites algorithms, this translated to 20000 steps, while for the genetic algorithm with a population size of 50, this translated to 400 generations. This was sufficient time for the score of all the algorithms to have largely converged for all the problem instances.

The optimization algorithms were run against simulated execution times only. Assuming the simulator provides a reasonable model of the real hardware, this is a similar problem to optimizing on the hardware itself. Each configuration was repeated 50 times, except in the pipelined case, where each configuration was only repeated 10 times. This was due to simulation of the 10 batches run in the pipeline case requiring 10 times the computation. In all experiments, device placements were produced for three different networks: AlexNet, ResNet-50, and Inception V3.

In the first version of the experiment, named the *normal* version, the networks were optimized for running on a simulated version of the real Malvik server, as described in Section 4.2. This has two GPUs, each with enough memory to fit each of the three networks. In this case, the optimal configuration is known to be using a single GPU to train the network, as this avoids communicational overheads. As such, the experiment evaluated whether the algorithms were able to find the solution to the trivial version of the device placement problem.

Figure 4.4 shows the results of the normal version. The bars show mean batch execution time across all repeats, while the lines on each bar display a 95% confidence interval. It is clear that for all networks, the hill climbing algorithm performed much worse than the other algorithms. This is to be expected, and indicates that this is a problem space with local optima. For AlexNet, the other optimizers were all able to find the optimal solution. However, for the more complex

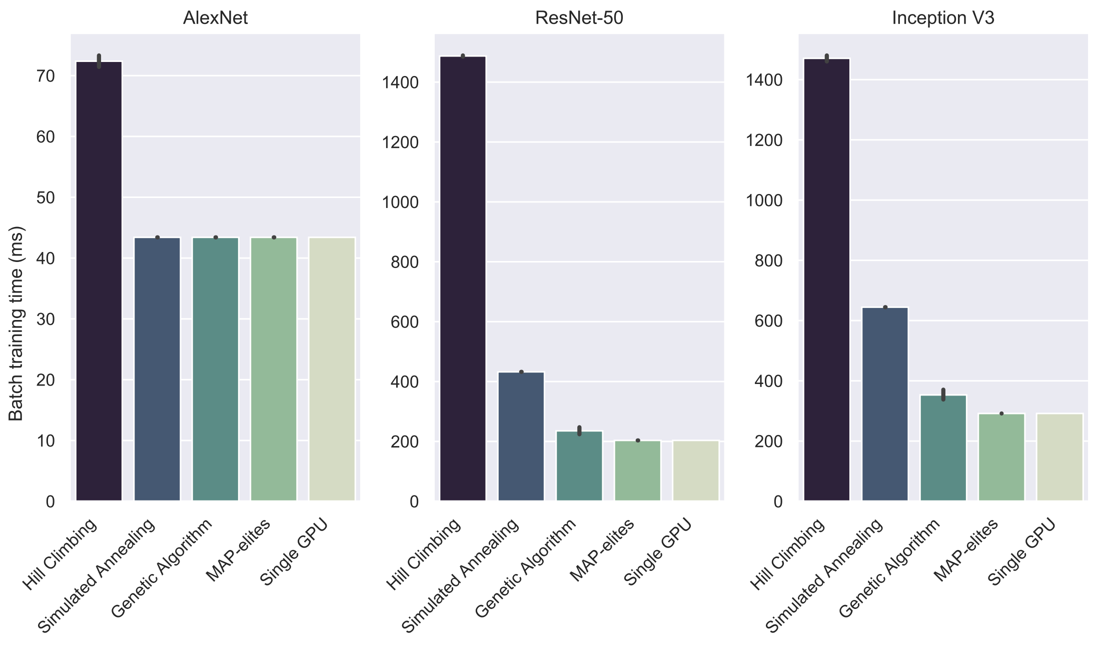


Figure 4.4: Comparison of optimization algorithms without any restrictions.

problems of ResNet-50 and Inception, the simulated annealing algorithm was no longer able to get close to this solution. The genetic algorithm was also not able to find the exact optimal solution, but it came significantly closer. The MAP-Elites algorithm was able to find the best solution for all the networks, which indicates that its focus on diversity was beneficial in this case. By careful tweaking of the genetic algorithm's parameters fitted to each network, it is likely that this small gap could be closed.

In the next version of the experiment, named the *limited* version, the networks were optimized to be run on a device architecture derived from Malvik, but with four GPUs. Moreover, the memory of each GPU was limited so that the network must be placed on all GPUs, or alternatively, the CPU must be utilized. The memory limits of each GPU were set to 200MB for AlexNet, 2.5GB for ResNet-50, and 2.5GB for Inception. Since all GPUs have the same properties, the problem here can be formulated as finding a configuration that fits into the memory of each GPU, while minimizing the communication cost. This is a much more complex problem than the normal version, with no trivial solution.

Figure 4.5 shows the results of the limited version of the experiment. For AlexNet, there was now more separation between the methods than in the normal version. Hill climbing was still significantly worse than the other methods, but the simulated annealing algorithm was no longer able to perform on the same level as

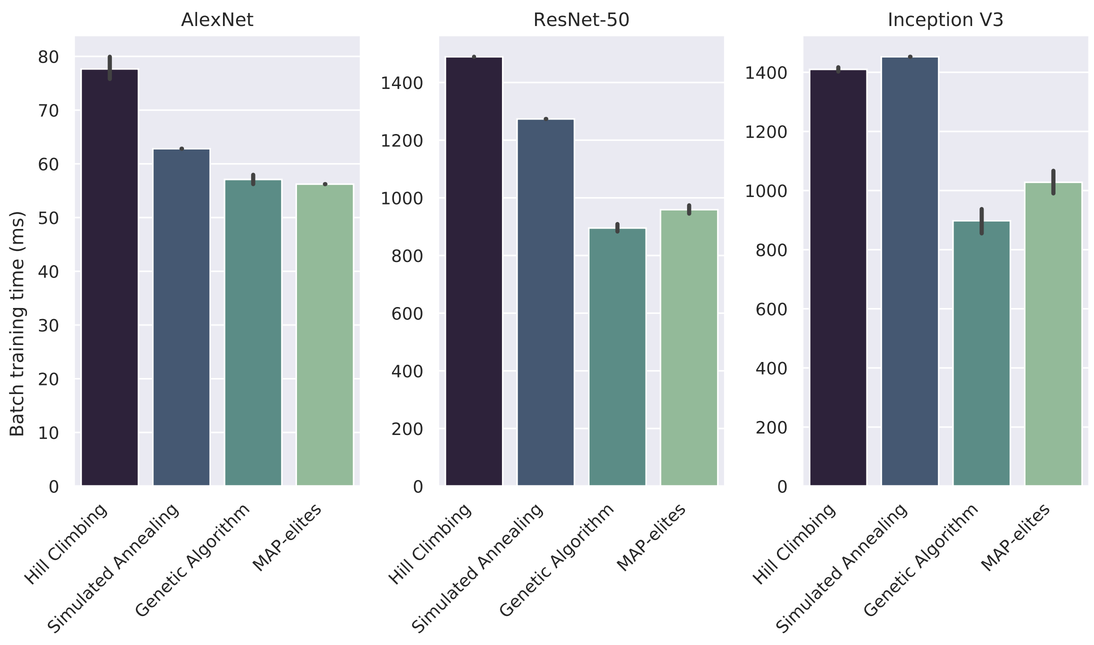


Figure 4.5: Comparison of optimization algorithms with memory limited so that all GPUs must be utilized.

the genetic algorithm and MAP-Elites algorithm. The genetic algorithm and the MAP-Elites algorithm had similar performance.

For ResNet-50 and Inception, the trend is even clearer, with the genetic algorithm performing much better than the simulated annealing algorithm. Interestingly, even the hill climbing algorithm outperformed the simulated annealing algorithm on the Inception network, indicating that it was able to find local minima that performed reasonably well.

The MAP-Elites algorithm performed slightly worse than the genetic algorithm on the ResNet-50 and Inception networks. This is not unexpected – the MAP-Elites algorithm is designed to find a diversity of good solutions, and will not necessarily outperform a regular genetic algorithm when only comparing the single best solution found. An important reason for this is that the MAP-Elites algorithm uses more computational resources for discovering novel solutions – what is typically called exploration, while the genetic algorithm will often converge on a single or small number of good solutions – it has a higher focus on exploitation. Consequently, the MAP-Elites algorithm can be expected to benefit from being allowed to run longer, while the solution of the genetic algorithm is unlikely to change much.

In the final version of the experiment, named the *pipelined* version, multiple batches were allowed to enter the system at once. The experiment was run on an equivalent of the Malvik server with four GPUs, and four batches were allowed in the system at once. This allowed a theoretical parallelism degree of four, but the true achievable parallelism can be expected to be lower than this. There was no penalty for memory usage, as long as the memory of the devices was not exceeded. The experiment therefore tested if the optimizers were able to exploit pipeline parallelism in order to achieve higher throughput than when running the training process on a single GPU. In order to allow batch training times to stabilize, each instance was run for ten batches.

Note that the quality of pipeline-parallel configurations produced by the system was not benchmarked on real hardware, as the task of running a neural network in a pipeline-parallel manner is a non-trivial task in itself, and out of scope for this project. Nevertheless, the task of finding pipeline parallel configurations on the simulator is a complex one, and thus a fitting test case for the optimizers.

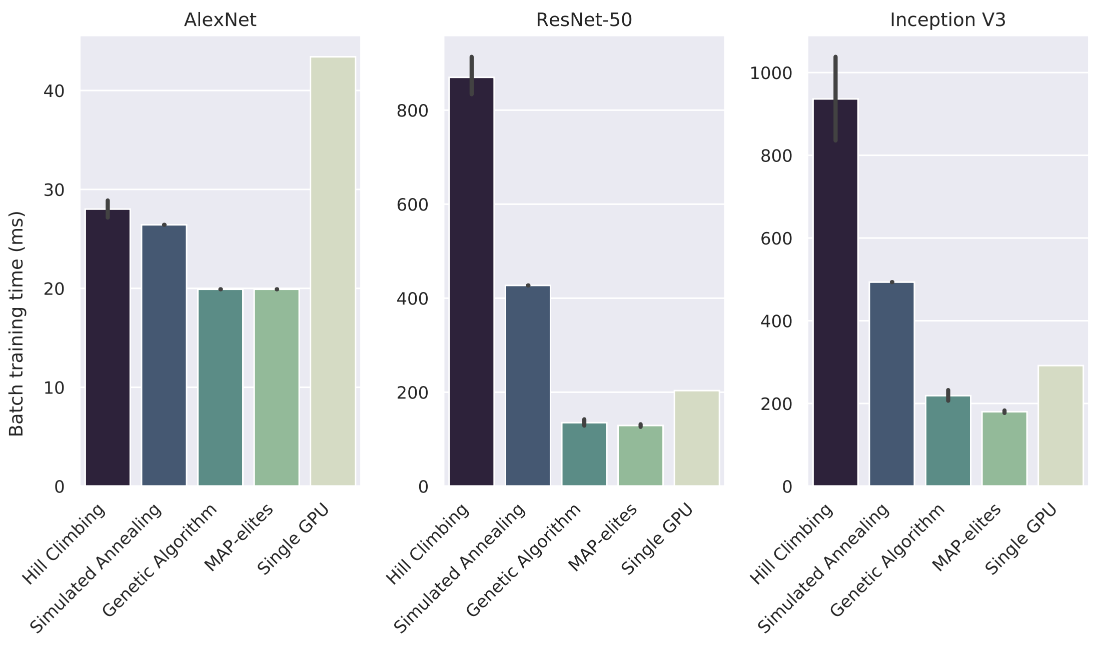


Figure 4.6: Comparison of optimization algorithms with four batches allowed in the system at once. Times are averaged over 10 batches.

Figure 4.6 shows the results of the pipelined version. The diagram shows averaged batch times, and the time required for running a batch on a single GPU is also shown for comparison. For AlexNet, all optimizers were able to find placements that outperform the use of a single GPU. The genetic algorithm and MAP-Elites algorithm performed equally, while the hill climbing and simulated annealing algorithm performed worse.

When running on the ResNet-50 and Inception networks, neither the hill climbing nor the simulated annealing algorithm were able to outperform the single GPU placement. This indicates that they struggled to navigate the considerably more complex problem space. It is interesting to observe that for these two networks, there is significant variance in the quality of solutions produced by the hill climbing algorithm. This indicates that there were many local minima of varying quality, and consequently, the algorithm was sensitive to the initial candidate solution. It should be noted, however, that the sample size was much smaller for this version than for the preceding two versions, which will naturally increase the confidence interval.

Looking at the results for the genetic algorithm and MAP-Elites algorithm on ResNet-50 and Inception, it is clear that both outperformed the placement of all operations to a single GPU. MAP-Elites performed slightly better in both problems, especially for Inception, which indicates that the increased diversity aided it in finding a better solution. However, for ResNet-50, the difference is non-significant ($t(18) = 1.56, p = 0.1371$).

An interesting observation is that although the solutions of the genetic algorithm and the MAP-Elites algorithm for ResNet-50 and Inception outperformed the single GPU placement, the reduction in time is less than 50%. This can indicate that the size of data packages transferred between the layers in these two networks introduced a communicational cost that largely outweighed the speedup acquired through pipeline parallelism. In fact, inspection shows that very few of the solutions utilized all four GPUs, with multiple only using two GPUs.

Figure 4.7 shows an event trace of one of the best solutions produced by MAP-Elites for the Inception network. Each batch is represented by two shades of colour – one for the forward pass, and one for the backward pass. Blocks represent an operation running on a given processor, and red lines represent tensor transfers between processors. It is evident that in this case, three devices were utilized, but device 1 carried out very little computation. On average, the degree of parallelism was less than two, which is consistent with the observations above. Notably, processing of different batches were interleaved on a single processor – this is clearly visible on device 3. This could indicate that there is room for improvement in the pipeline scheduling system – this is quite simplistic in the execution simulator, and Harlap et al. [2018] note that this is a crucial aspect of pipeline systems. A better scheduling mechanism might therefore increase the potential for speedups through pipeline parallelism. This is out of the scope of the project.

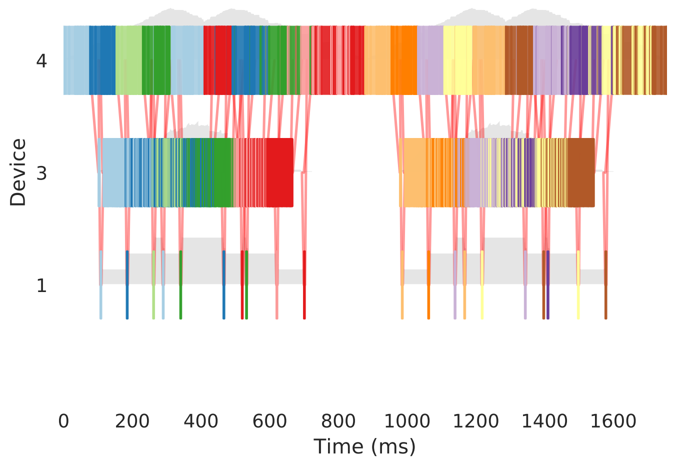


Figure 4.7: Event trace of one of the best pipelined placements for the Inception network. Colours represent operations from individual batches, with each colour being present in two shades: one for the forward pass, and one for the backward pass.

4.3.4 E4 — Transfer of Solutions During Optimization

The following set of experiments looked at the transfer of the population from the simulator to the real hardware during optimization, performing parts of the training on simulated results, and parts on benchmarked results. This was done in order to evaluate the stability of the optimization process during such a transfer, and the difference between the optimization process when using simulated and benchmarked times.

In order to reduce the computational load for the benchmarking part of the optimization processes, the population size was reduced from 50 to 30. This did not appear to have a large impact on the performance of the algorithm. When moving the population from the simulator to the real hardware, the transfer was done by preserving the 30 best solutions in the current population.

In the first instance, a GA optimization process of ResNet-50 with limited available memory was run for 500 generations against the simulator, allowing it time to converge. The solution was then transferred to the real hardware, and run for another 100 generations optimizing against benchmarked execution times. The result is shown in Figure 4.8. As can be observed, by the time the transfer was made, the algorithm had converged. Moreover, the fitness score stayed stable even after transferring to the real hardware, indicating that this was a stable point with

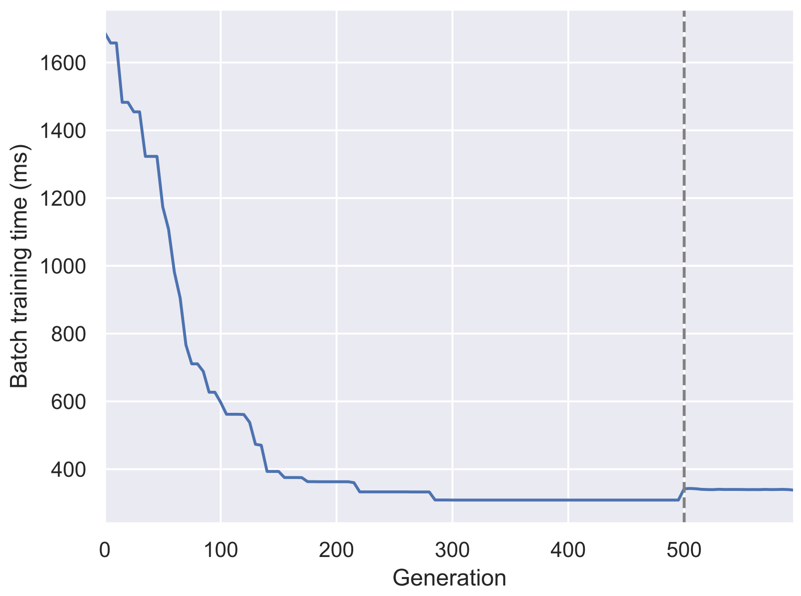


Figure 4.8: Optimization of ResNet-50 with limited available memory, with the population transferred from the simulator to the real hardware at generation 500.

benchmarked results as well. Note that the small jump in batch execution time at generation 500 is due to the difference in simulated and benchmarked results for the same solution.

In the next experiment instance, a genetic algorithm was used to optimize ResNet-50 for Malvik with limited available memory. The genetic algorithm was run for 50 generations on the simulator, allowing it to filter out the worst solutions that would take excessive time to benchmark. It was then run for 300 generations against benchmarked evaluations. Another run of the genetic algorithm was performed on the exact same problem, but running exclusively on simulated times.

Figure 4.9 shows the result of this run. The plot is cut at generation 210, as at this point during the benchmarked run, the server experienced a sudden increase in load, leading to a spike in training time. The entire history is available in Appendix A. For the simulated run, the benchmarked time of the final solution is displayed from generation 190 to generation 210, so the final times of the two runs are directly comparable.

The fitness history of the two runs follow a similar path, indicating that the optimization process is similar when running on the simulator and the real hardware. There is some difference, but the genetic algorithm is a stochastic method, so there will always be some variance between different runs. Observe that the

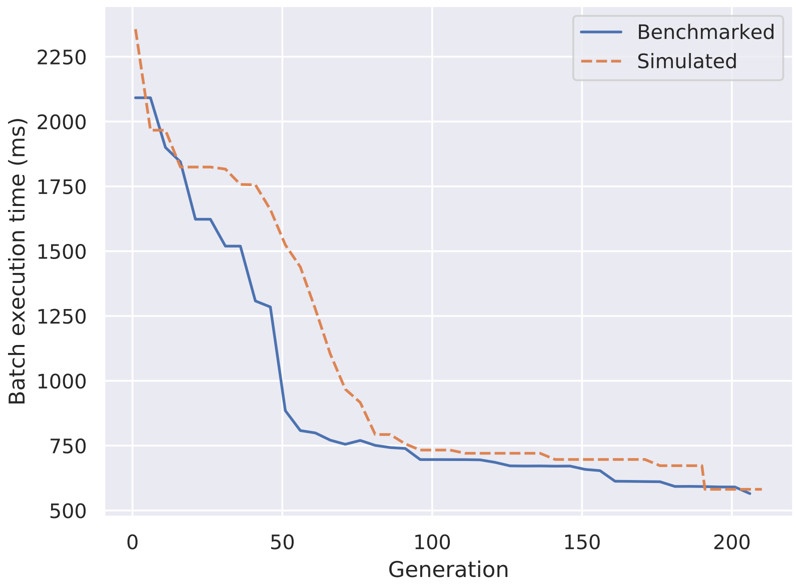


Figure 4.9: Comparison of a GA running on real hardware from generation 50 to a GA only using the simulator when optimizing ResNet-50 for a limited-memory version of Malvik.

quality of the solution for the two algorithms around generation 200 is practically equal. This shows that for this problem, the use of the simulator does not impact the quality of the produced solution.

Figure 4.10 shows another instance with the same setup, in which the benchmarked experiment was allowed to run uninterrupted until completion. This shows the same trend, with the two optimization processes showing similar profiles, and arriving at solutions of approximately equal quality. In this case, it is also clear that both runs have converged, indicating that little further improvement would take place if the algorithm was run for a longer time. (The jump in the plot at generation 330 for the simulated run is simply due to the execution time of the same configuration being slightly different in the simulator compared to a benchmark.)

This run also allows an estimation of the difference in time requirements when doing optimization against real hardware, as opposed to when using simulated evaluations. The full optimization run using benchmarking shown in Figure 4.10 took 19.5 hours. In comparison, the corresponding simulation-based run took 55 seconds. This is a difference of roughly three orders of magnitude. Moreover, this was while running the first 50 generations for the benchmarking run against the simulator, thus removing the most time-consuming part of the optimization process. Note, however, that the simulated run ran on a powerful server, and

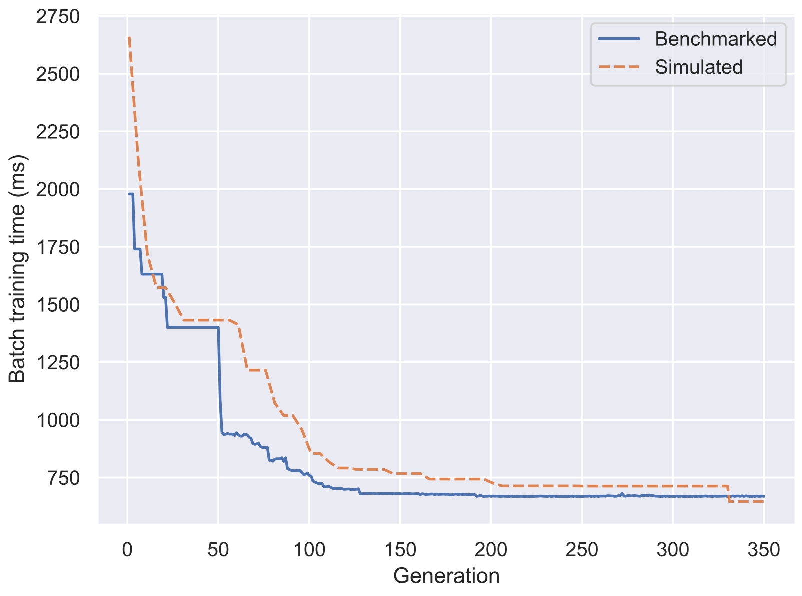


Figure 4.10: Comparison of a GA running on real hardware from generation 50 to a GA only using the simulator when optimizing ResNet-50 for a limited-memory version of Malvik.

employed massive parallelism. Still, the advantage of the simulator-based approach with regards to optimization time is clear.

Oftentimes, neural networks are trained on powerful servers that are shared between several users. The server may therefore already be under significant load when the training task is submitted, directly impacting the optimal placement for the given task. In the next experiment, the optimization process was by design applied for a server on which another large deep learning task was already running. An optimization run was performed using both the genetic algorithm and the MAP-Elites algorithm, and the optimizers were tasked with finding a suitable configuration for ResNet-50. Both algorithms were allowed to run for a total of 20000 function evaluations against the simulator, before the population was transferred to the real hardware. For MAP-Elites, the 50 best solutions were then benchmarked, and the best selected, while the genetic algorithms was allowed to run 10 generations of fine-tuning.

The result is shown in Figure 4.11, with Figure 4.12 showing a magnified view of only the results after the transfer to the real server, allowing easier comparison. The result of the genetic algorithm is initially off the charts after the transfer, indicating that the system runs out of memory when attempting to run this configuration. Through the fine-tuning, the genetic algorithm was able to find a

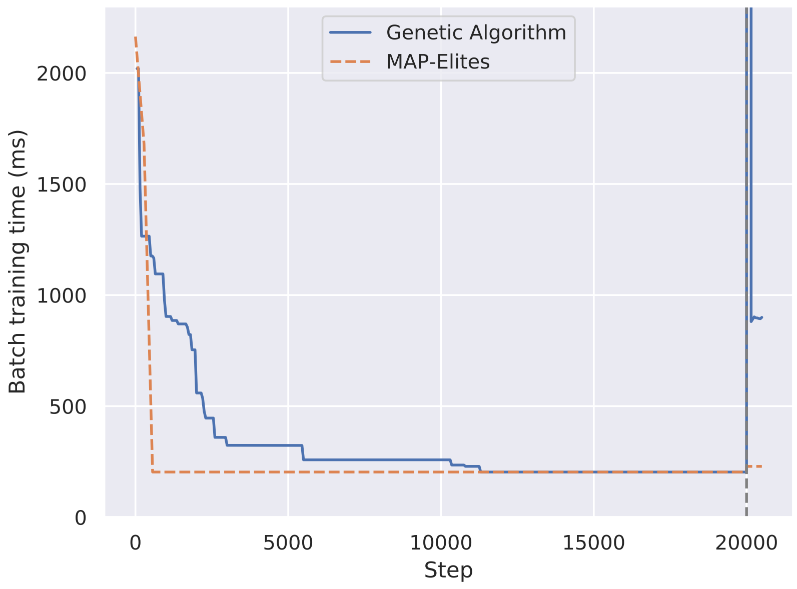


Figure 4.11: Comparison of a genetic algorithm and a MAP-Elites run when transferred to a server that is already under heavy load at step 20000.

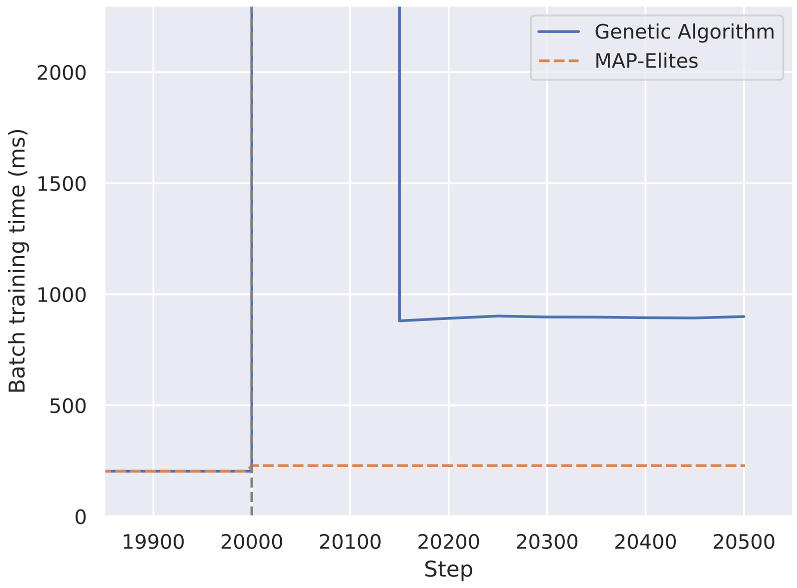


Figure 4.12: Comparison of a genetic algorithm and a MAP-Elites run when transferred to a server that is already under heavy load at step 20000.

configuration that fits into the available memory, but it performed quite poorly. The MAP-Elites algorithm, on the other hand, was able to find a good solution from its repertoire through the benchmarking operation. It appears to have been able to exploit one of the GPUs seeing lower load than the other, yielding a massive increase in performance over the solution produced by the genetic algorithm. Furthermore, it did this in 50 function evaluations of benchmarking, while the genetic algorithm used 150 to arrive at a much poorer solution. This demonstrates the utility of producing a diverse set of quality solutions that can all be considered when doing the solution transfer.

4.3.5 E5 — Comparison of Simulation and Benchmarks

The following set of experiments aimed at evaluating the accuracy and applicability of simulated batch times to the optimization process by comparing simulated and benchmarked execution times for the same network placements. A varied set of placements for a given network were produced by running a genetic algorithm, and saving the best placement of the population at a given interval. The saved placements were then benchmarked on real hardware, in order to allow direct comparison. This process was repeated 20 times for each of the three networks – AlexNet, Inception, and ResNet-50. For AlexNet, the genetic algorithm was run for 50 generations, with a checkpoint each generation, resulting in 50 measurements each run, for a total of 1000. For ResNet-50 and Inception, the genetic algorithm was run for 400 generations, with a checkpoint each fifth generation, yielding 80 measurements per run, and 1600 in total for each network.

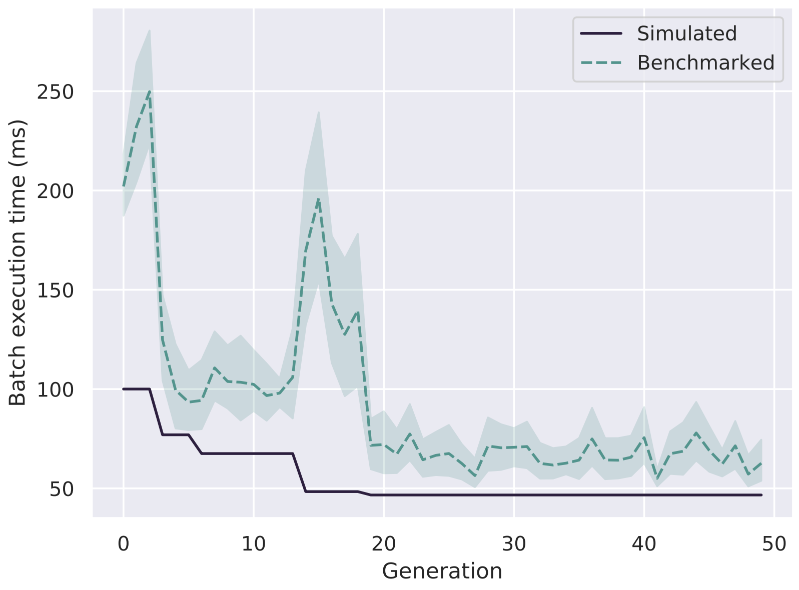


Figure 4.13: Comparison of simulated and benchmarked batch times of placements from a genetic algorithm run optimizing device placement for AlexNet

Figure 4.13 shows an example of such a run for AlexNet. Batch times are plotted against the generation from which the given placement is taken. The solid blue line shows the simulated times which the genetic algorithm used in the optimization process, while the dashed green line shows the mean benchmarked time with a 95% confidence interval. Since the simulated times are from an optimization process, these will naturally decrease as the generation increases. The benchmarked results show some of the same trend, but with considerable noise – notably, there is a large spike around generation 15-20. Noisy results are prevalent in most of the AlexNet runs.

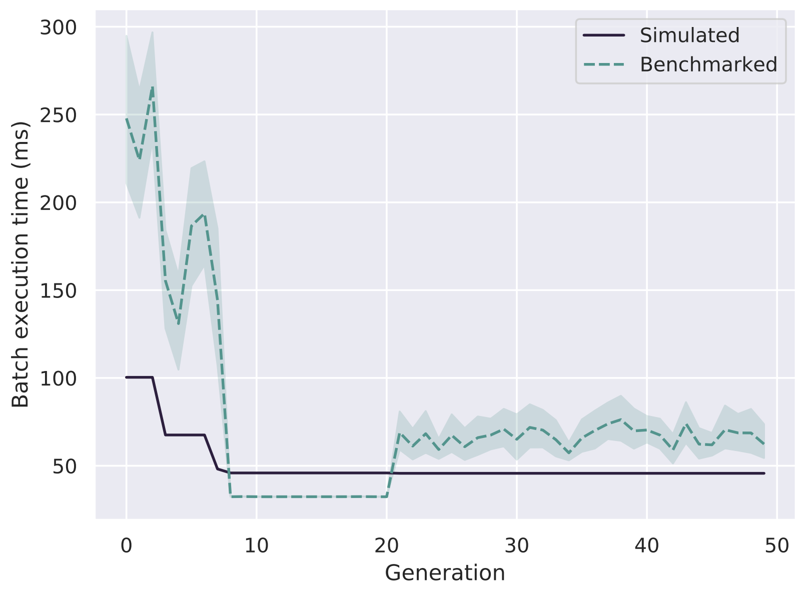


Figure 4.14: Comparison of simulated and benchmarked batch times of placements from a genetic algorithm run optimizing device placement for AlexNet

AlexNet is by far the simplest network used in the experiments, and runs considerably faster than both ResNet-50 and Inception. This makes it more susceptible to interference from other processes on the computer, as the lower computational cost of the network means that the cost of other processes is larger, relatively speaking. It is likely that this can account for much of the noise observed in the AlexNet runs. This is corroborated by the fact that the confidence intervals in the plots of the AlexNet benchmarks are quite large – especially for anomalous regions. This indicates significant fluctuations between the benchmarks of a single placement, which indicates that there may very well be even greater fluctuations between benchmarks of different placements. Moreover, as shown in Figure 4.14 from around generation 20, such anomalies sometimes appear in regions in which the algorithm has already converged, meaning that there is little change in the solution itself. This indicates that the noise is external.

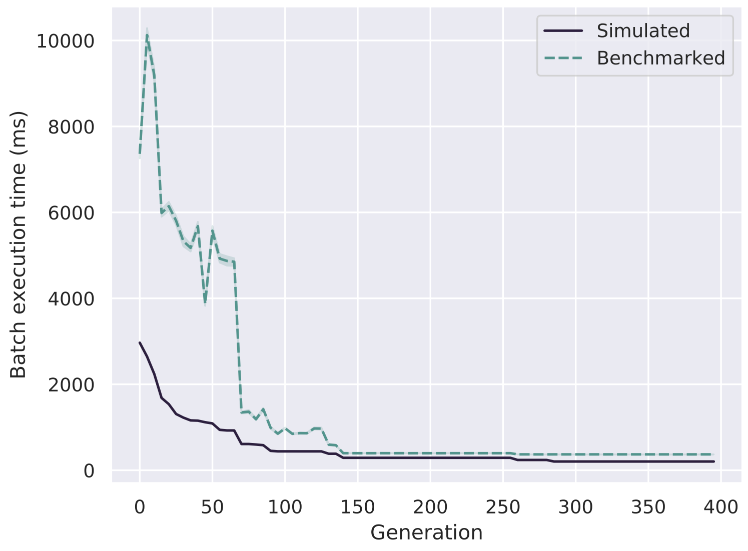


Figure 4.15: Comparison of simulated and benchmarked batch times of placements from a genetic algorithm run optimizing device placement for ResNet-50

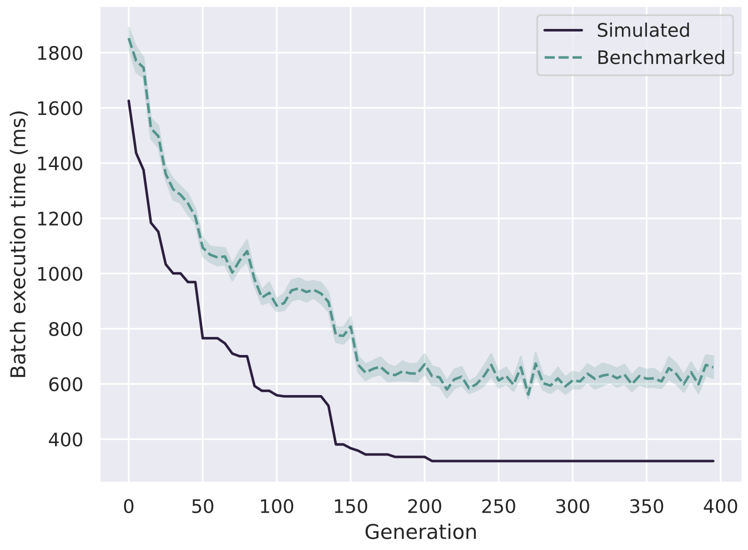


Figure 4.16: Comparison of simulated and benchmarked batch times of placements from a genetic algorithm run optimizing device placement for Inception

The two larger networks gave clearer results. Figure 4.15 shows a comparison plot for ResNet-50. Here, there is a clear trend that solutions from later generations are better when running on the real hardware than solutions from earlier generations. There is still some noise, which may result from inaccuracies in the simulator or interference from other processes. However, towards the later generations, the benchmarked execution times for the later generations were both stable and low, indicating that solutions that were good on the simulator were also good on the real hardware.

Figure 4.16 shows similar results for Inception V3. In this case, the numerical results of the simulator were less accurate, but the trends in the plot are the same – the solutions in later generations were better than the solutions in earlier generations. The numerical results could be improved by tweaking communication and computation penalty factors in the simulator. However, for an optimization process, the exact simulated times are unimportant – as long as the ordering of solutions is correct, the optimizer will arrive at the correct solution.

More comparison plots of the kind shown above are available in Appendix A.

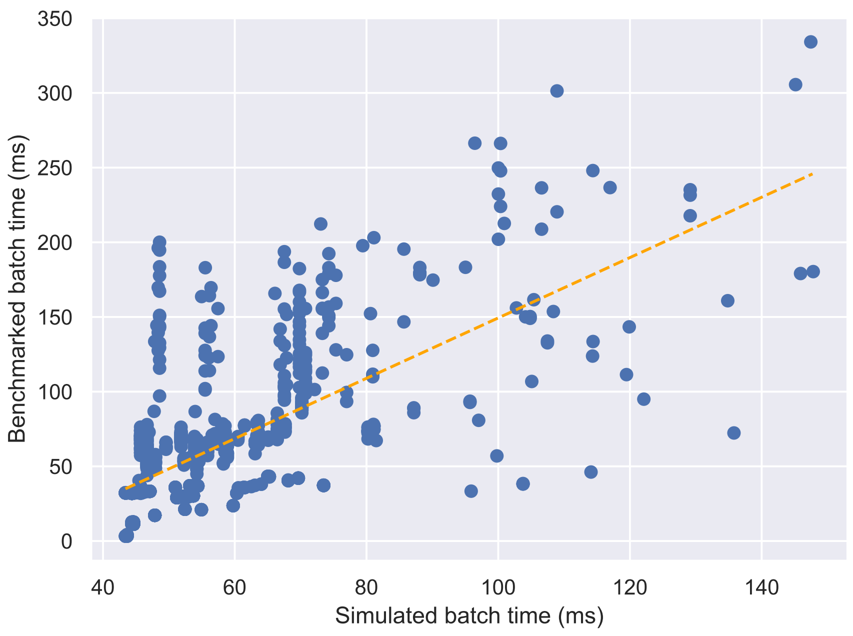


Figure 4.17: Scatter plot comparing simulated and benchmarked execution times for AlexNet.

In order to make a clearer comparison between simulated and benchmarked execution times, all observations for each network were gathered, and scatter plots comparing the two measurement types were produced. These plots are displayed in Figure 4.17, Figure 4.18, and Figure 4.19 for AlexNet, ResNet-50, and Inception, respectively. In each plot, a line fitted with linear regression is also shown.

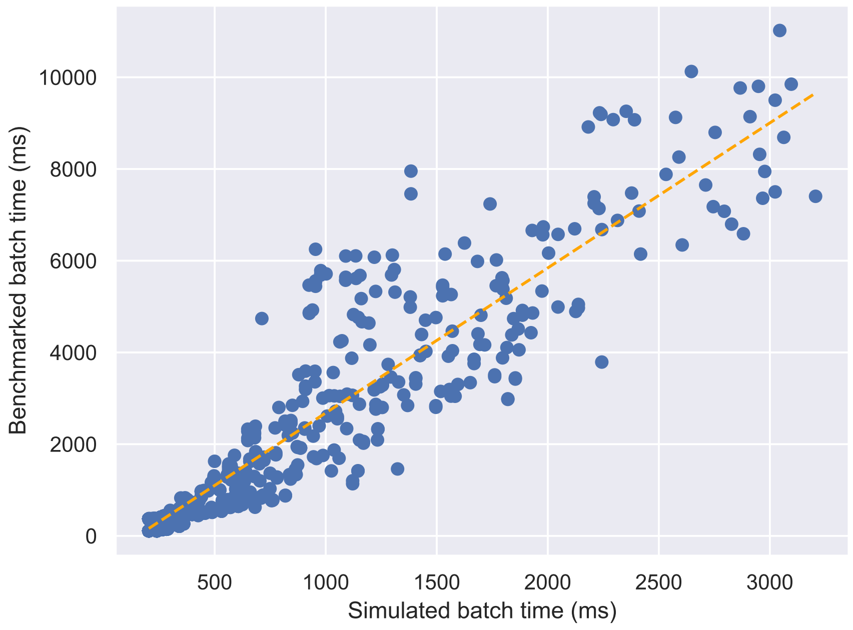


Figure 4.18: Scatter plot comparing simulated and benchmarked execution times for ResNet-50.

The Pearson correlation coefficients are $R_A = 0.672$ for AlexNet, $R_R = 0.939$ for ResNet-50, and $R_I = 0.804$ for Inception.

The AlexNet results are quite noisy, which is consistent with the previous observations. However, for ResNet-50 and Inception, there is a clear positive correlation. The exact numerical values do not match, but as noted above, this is irrelevant for the optimization process itself, as long as the ordering of solutions is preserved. The correlation is especially strong for ResNet, but it is significant also for Inception. The results from ResNet-50 and Inception therefore give a strong indication that the results produced through optimization against the simulator are valid.

While noisy, the results from AlexNet also show a positive correlation. It is likely that the simulator is equally valid for AlexNet, and that much of the noise is a result of the chaotic environment that is a modern computer. This also highlights a different observation: optimizing against the execution simulator is an easier problem than optimizing against the real hardware, since external noise is eliminated altogether.

It cannot be entirely excluded that parts of the anomalous results for AlexNet might be caused by inaccuracies in the simulator, however. Such anomalous regions can also be observed for the other two networks, although they are both less

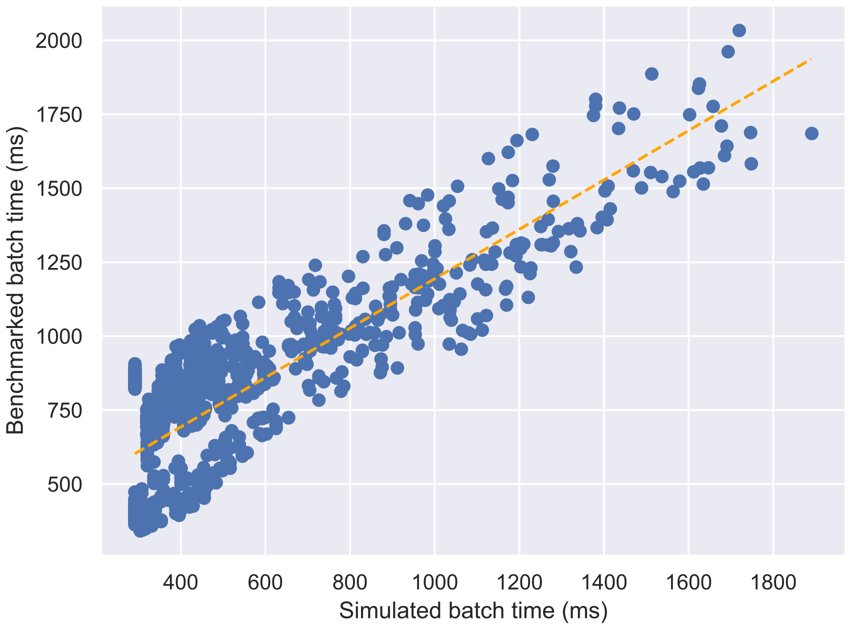


Figure 4.19: Scatter plot comparing simulated and benchmarked execution times for Inception V3.

prevalent and less severe. These anomalies typically appeared in early stages of the optimization processes. Solutions in these stages are characterized by there being a larger spread of operations across devices, leading to more communication. It is therefore most likely that any inaccuracy is located in the part of the system responsible for simulation of communication. E1 showed that the real communication bandwidth is quite far from the idealized model, with packet sizes having a strong influence on the practical bandwidth. When communication occurs more often, queueing of transfers in communication channels will also have a larger influence on communication times. The system does model queueing of transfer operations, but modelling the behaviour of a computer system exactly is non-trivial. In fact, modelling the impact of transfers external to the training process in these queues is practically impossible.

Nevertheless, the trend is that the solutions on which the optimizer arrives are good solutions when measured by benchmarks as well. Since the anomalous regions typically appear in early stages of the optimization, they only affect solutions that are considered of low quality by both simulated and benchmarked measures. The anomalies are therefore unlikely to impact final solutions much. They may, however, prove to be a more severe problem in even more complex problem settings – that is, problems with computation spread over a larger number of devices, thereby requiring more communication.

4.4 Summary

In this section, five experiments have been presented. E1 explored the behaviour of bandwidth on a PCIe3 bus when transferring data packets of different sizes, showing both that practical performance is significantly lower than theoretical, and that it depends a lot on packet sizes. E2 looked at the stability of batch training times in PyTorch, and showed that representative results can be achieved by dropping the time of the first batch in the process. E3 compared the results of the genetic algorithm and the MAP-Elites algorithm with that of a hill climbing and a simulated annealing algorithm, showing that the genetic algorithm and the MAP-Elites algorithm outperformed the control algorithms overall, with the two alternatingly performing best.

E4 looked at optimization processes that were in part performed on simulated and in part on benchmarked execution times. It showed that the process was stable when transferring from simulation to benchmarks, and that optimization performed against simulation and benchmarks showed similar profiles. A comparison of the genetic algorithm and the MAP-Elites algorithm when transferring solutions to a busy server indicated the utility of the diverse solution set produced by MAP-Elites, which aided the discovery of a good solution in a sub-optimal environment.

Finally, E5 directly compared simulated with benchmarked execution times of the same network configurations. Apart from noisy results when running on the AlexNet network, the results showed a clear positive correlation, with good solutions on real hardware being good on the simulator, and vice versa.

Chapter 5

Conclusion

In this chapter, the results of the experiments and their implications are discussed at a high level, with respect to the research questions. Section 5.1 contains the main discussion, while Section 5.2 gives a summary of the main contributions of this project, and Section 5.3 concludes the report by giving an outline of possible directions for future work.

5.1 Discussion

The goal of this project was twofold: to evaluate the performance of the two evolutionary algorithms when applied to the device placement problem, and to determine the impact of the simulator when used as a part of this optimization process. To this end, the experiments presented in Chapter 4 were carried out.

Experiment E3 demonstrated that the two evolutionary algorithms were able to perform well on the device placement problem, finding the optimal solution to the trivial version of the problem, and significantly outperforming the baseline on more complex problems. The experiments were performed at a fixed number of function evaluations, which approximately translates to the same amount of computation being allowed for each method. As such, they tested the ability of the algorithms to reach a good solution in a set amount of computational time. The limit on function evaluations was set to a number that would allow most of the algorithms to converge, which it can be argued favours the more complex algorithms. However, due to the efficiency of function evaluations when using the execution simulator, the cost of extra evaluations is quite low. Therefore, the quality of solutions found upon convergence becomes the most important goal. Moreover, the evolutionary algorithms are easier to parallelize, which translates to the same amount of function evaluations requiring less wall clock time.

The results of E3 did not show any clear difference between the genetic algorithm and the MAP-Elites algorithm, with these two performing equally in some cases, and alternatingly performing best in others. This is as expected, as the two solve problems in slightly different ways, with neither being clearly better than the other when it comes to producing the best single solution. MAP-Elites focuses heavily on diversity and exploration, which may translate to higher quality solutions in some cases. However, the higher selection pressure of the genetic algorithm may in other cases aid convergence in such a way that the final solution is better than the one discovered by MAP-Elites.

Experiment E4 gave indications that the optimization process will progress in similar ways when run against the simulator and against real benchmarks. The gradients of the fitness plots produced in these runs are very similar. Additionally, the quality of solutions produced by the optimization runs were very similar when both were measured by benchmarks. This indicates that the optimization process will progress in similar ways, and that solutions produced by optimizing against the simulator are valid for transfer to real hardware. However, the limited number of runs means that no argument can be made as to the statistical significance of these results.

Another interesting observation from E4 is the comparison between the genetic algorithm and the MAP-Elites algorithm when attempting to find a good placement for a server that was under heavy load. In this run, the MAP-Elites algorithm was able to find a solution that outperformed the one produced by the genetic algorithm using just a few benchmark operations. This property can help bridge the gap between finding good placements on the simulator, and finding placements that perform well on the actual training hardware. It should be observed that also in this case, the limited number of runs is a drawback.

The ability of the MAP-Elites algorithm to find a diverse set of solutions can also help bridge another gap that is a prevalent problem of deep learning. Most modern machine learning methods, including the reinforcement learning approaches that have previously been applied to the device placement problem, are so-called black-box methods. This means that they do not provide any justification for the solution they propose. This can impede the adoption of the method in the industry, as experts may have little confidence in the results. While the MAP-Elites algorithm does not provide any justification of its solutions per se, the production of a set of candidate solutions enables an expert to take control over the selection of a final solution. In this way, the MAP-Elites algorithm can function as a decision support system rather than a fully automated system. Note that this is also achievable with a regular genetic algorithm, if excessive convergence is avoided. However, the MAP-Elites algorithm is guaranteed to provide diverse results.

Experiment E5 offers further support to the validity of solutions produced by the execution simulator. The results show that the ordering of solutions was generally the same when using simulated and benchmarked execution times. The estimated execution times were not fully accurate, but this is not necessary for the optimization process to be valid. Moreover, the task of providing entirely accurate results is impractical, as a computer system is highly complex, with a number of external factors potentially affecting benchmark results. However, as long as the ordering of tasks is correct, an optimization process will arrive at a valid solution.

Nevertheless, there were anomalies that should not be ignored. The measurements show significant noise. Much of this can likely be attributed to external factors, but some anomalies are severe enough that they indicate systematic errors. These anomalies typically concern placements requiring a lot of tensor transfers, implying that an error may exist in the way communication is simulated. In that case, a likely cause is the way scheduling on communication is handled by the simulator – especially regarding how busses are arranged in the computer architecture itself. Achieving a fully correct simulation of these systems may require large increases to the complexity of the simulator.

It is important to observe that due to the large speed-up of the optimization process, some loss of quality in the produced solutions can be accepted. Since significant time is already saved by using simulated results during optimization, a slower solution can still lead to shorter total time – i.e. including both the optimization and the actual training of the network. The speed-up is estimated to be in the range of about three orders of magnitude. Using the example from which this estimation was derived, there was a speed up from 55 seconds, which is practically instantaneous, to 19.5 hours. By using the simulator for optimizing the device placement of this training process, the training itself could commence 19.5 hours earlier than if using real benchmarks. Even for a training process spanning several days, an immense speed-up would be required in order to justify using this amount of extra time for the optimization process.

5.2 Contributions

This report has presented a new execution simulator for deep neural networks, building on previous efforts by [Qi et al., 2016] and Addanki et al. [2019]. The simulator allows the estimation of execution time for a given neural network when placed in a specified way onto a given device graph. It supports the main layer types present in typical convolutional neural networks, which account for a large portion of modern state-of-the-art deep learning models. The simulator allows for the simulation of multiple concurrent batches of data, enabling the simulation of a simple type of pipeline parallelism. Due to the simulator being independent from the hardware it simulates, and three orders of magnitude faster than benchmarking, it facilitates the efficient solution of the device placement problem without requiring access to the training hardware. Experiments indicate that solutions produced when optimizing against the simulator are valid for application in the real world.

This report has also introduced two evolutionary algorithms for solving the device placement problem. To the best of my knowledge, this is the first application of evolutionary algorithms to the device placement problem. Results of the experiments indicate that these types of algorithms are highly capable of solving this problem. Tests also show that the ability of the MAP-Elites algorithm to produce multiple quality solutions can help the system to find a good solution when transferring solutions from a simulator-based optimization process to the real world. In this way, it alleviates any negative impact the simulator may have on produced solutions.

Combined, the execution simulator and the optimization algorithms form a novel system for solving the device placement problem.

5.3 Future Work

In this project, evolutionary algorithms for solving device placement problems have been evaluated against a baseline of classic algorithms and trivial solutions, indicating that they are highly capable of providing improved results. However, reinforcement learning has also been applied to this problem in multiple previous instances. A natural next step is therefore the direct comparison of the methods introduced in this report with the current state-of-the-art reinforcement learning-based methods for solving the device placement problem.

This report also introduces a new execution simulator that yields multiple advantages when used in the solution of the device placement problem. In this project, the simulator has been used together with evolutionary algorithms. However, the nature of the simulator allows it to be used with any optimization algorithms. A possible direction for future work is therefore to look at the integration of the simulator with other optimization methods – for example reinforcement learning.

The simulator currently supports regular feed-forward networks as well as most convolutional network designs. Together, this constitutes a large fraction of current deep learning models. However, there are other large groups of networks – in particular, the recurrent neural networks and attention networks frequently applied to natural language processing. Expanding the list of layer types supported by the simulator would therefore enable it to process a larger range of models.

The simulator is able to simulate hardware independent of access to it. In fact, it is able to simulate hardware independent of the very existence of that hardware. This enables not only the evolution of placement of neural networks onto the given hardware, but the evolution of the hardware itself. By co-evolving hardware configurations and the placement of networks onto said hardware, it may be possible to discover hardware configurations that have not been conceived by human engineers, and that may possibly outperform current hardware for deep learning applications. It is likely that the simulator would need further improvement in order to facilitate such a process, as higher granularity and accuracy may be required.

Bibliography

Addanki, R., Venkatakrishnan, S. B., Gupta, S., Mao, H., and Alizadeh, M. (2019). Placeto: Learning generalizable device placement algorithms for distributed machine learning. *arXiv preprint arXiv:1906.08879*.

Andreassen, E. L. (2019). Optimizing device placement for deep learning using heuristic search and execution time modeling. Project report in TDT4501, Department of Computer Science, NTNU – Norwegian University of Science and Technology.

Andreassen, E. L. (2020a). Exprimo: A performance model for deep neural networks. `https://github.com/Lagostra/exprimo`.

Andreassen, E. L. (2020b). Exprimo: Source code for experiments. `https://doi.org/10.5281/zenodo.3878186`.

Bäck, T. (1992). Self-adaptation in genetic algorithms. In *Proceedings of the first european conference on artificial life*, pages 263–271. The MIT Press, Cambridge, MA.

Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. (2012). Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE.

Corrêa, R. C., Ferreira, A., and Rebreyend, P. (1999). Scheduling multiprocessor tasks with genetic algorithms. *IEEE Transactions on Parallel and Distributed systems*, 10(8):825–837.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., and Salakhutdinov, R. (2019). Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee.

Eiben, A. E., Smith, J. E., et al. (2015). *Introduction to evolutionary computing.* Springer, 2nd edition.

Gao, Y., Chen, L., and Li, B. (2018). Spotlight: Optimizing device placement for training deep neural networks. In *International Conference on Machine Learning*, pages 1662–1670.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning.* MIT press.

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., and Gibbons, P. (2018). Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377.*

Hastings, W. K. (1970). Monte carlo sampling methods using markov chains and their applications.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Hou, E. S., Ansari, N., and Ren, H. (1994). A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed systems*, 5(2):113–120.

Huang, H., Cheng, P., Xu, H., and Xiong, Y. (2020). Simulating performance of ml systems with offline profiling. *arXiv preprint arXiv:2002.06790.*

Jia, Z., Lin, S., Qi, C. R., and Aiken, A. (2018a). Exploring hidden dimensions in parallelizing convolutional neural networks. *arXiv preprint arXiv:1802.04924.*

Jia, Z., Zaharia, M., and Aiken, A. (2018b). Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358.*

Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882.*

Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997.*

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Lehman, J. and Stanley, K. O. (2011). Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 211–218.

Mahajan, D., Girshick, R., Ramanathan, V., He, K., Paluri, M., Li, Y., Bharambe, A., and van der Maaten, L. (2018). Exploring the limits of weakly supervised pretraining. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 181–196.

Mirhoseini, A., Goldie, A., Pham, H., Steiner, B., Le, Q. V., and Dean, J. (2018). A hierarchical model for device placement.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. (2017). Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2430–2439. JMLR. org.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

Mouret, J.-B. and Clune, J. (2015). Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035.

PyTorch (2020). torchvision. `https://github.com/pytorch/vision`.

Qi, H., Sparks, E. R., and Talwalkar, A. (2016). Paleo: A performance model for deep neural networks.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8).

Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Synnaeve, G., Xu, Q., Kahn, J., Grave, E., Likhomanenko, T., Pratap, V., Sriram, A., Liptchinsky, V., and Collobert, R. (2019). End-to-end asr: from supervised to semi-supervised learning with modern architectures. *arXiv preprint arXiv:1911.08460*.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

Wu, A. S., Yu, H., Jin, S., Lin, K.-C., and Schiavone, G. (2004). An incremental genetic algorithm approach to multiprocessor scheduling. *IEEE Transactions on parallel and distributed systems*, 15(9):824–834.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.

Zhou, Y., Roy, S., Abdolrashidi, A., Wong, D., Ma, P. C., Xu, Q., Zhong, M., Liu, H., Goldie, A., Mirhoseini, A., et al. (2019). Gdp: Generalized device placement for dataflow graphs. *arXiv preprint arXiv:1910.01578*.

Appendices

Appendix A

Additional Experiment Results

For some of the experiments presented in this report, aggregation and statistical analysis of the results is impractical. The results section of the report therefore presents a selection of results that are representative and aids the analysis. Results from additional runs of the experiments are included in this appendix.

A.1 E4 — Transfer of Solutions During Training

In this experiment, presented in Section 4.3.4, optimization of device placements for different networks were initially performed against the execution simulator for a given number of generations, before the population was transferred to the real hardware, and the optimization process continued with benchmarked scores. In the figures below, the best execution times in the population is plotted against the generation. The vertical dashed line indicates the point at which the transfer from simulator to real hardware was made.

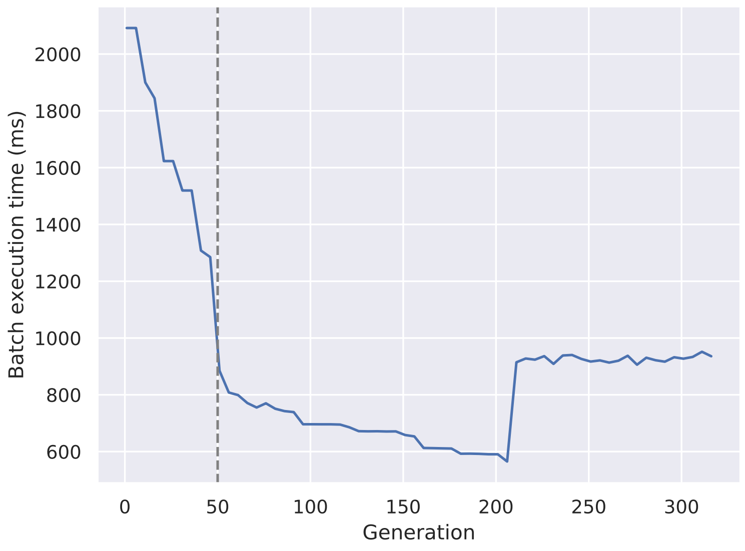


Figure A.1: Full run of GA optimizing ResNet-50 for Malvik with 50 simulated and 300 benchmarked generations, including spike in batch time due to increased load on server around generation 210.

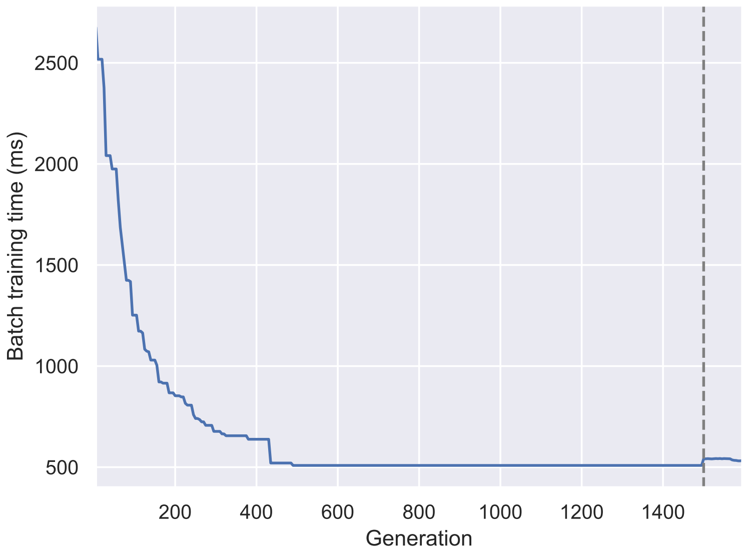


Figure A.2: Optimization of ResNet-50 on Malvik, optimizing for 1500 generations against the simulator, and 100 generations against benchmarked times.

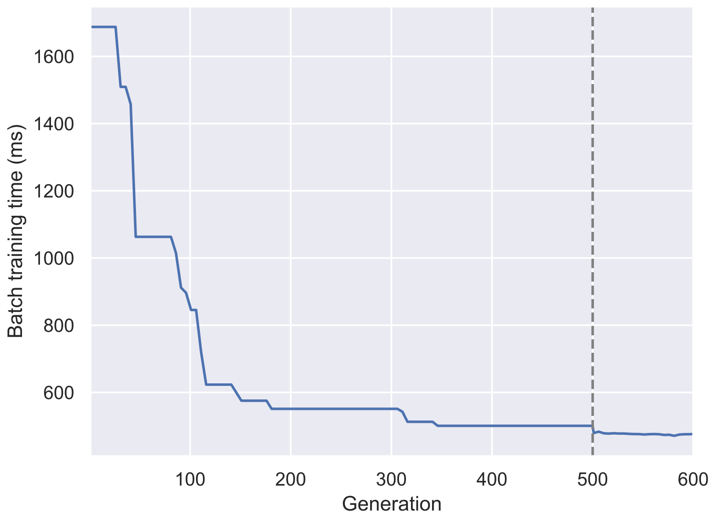


Figure A.3: Optimization of ResNet-50 on Malvik with limited available memory, optimizing for 500 generations against the simulator, and 100 generations against benchmarked times.

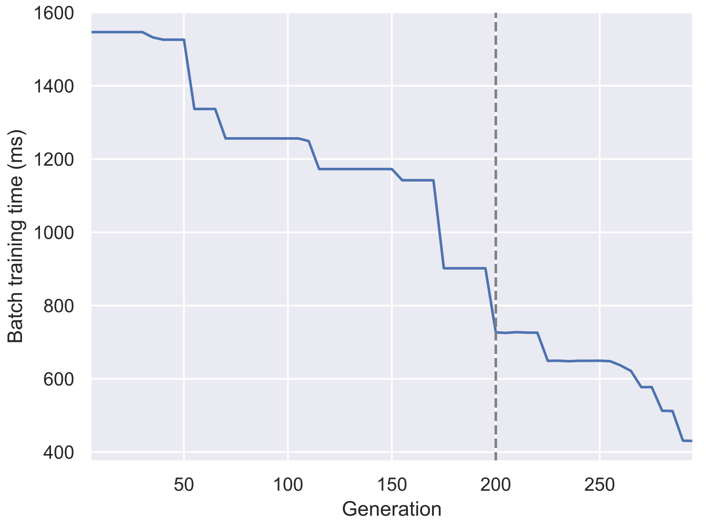


Figure A.4: Optimization of ResNet-50 on Malvik with limited available memory, optimizing for 200 generations against the simulator, and 100 generations against benchmarked times.

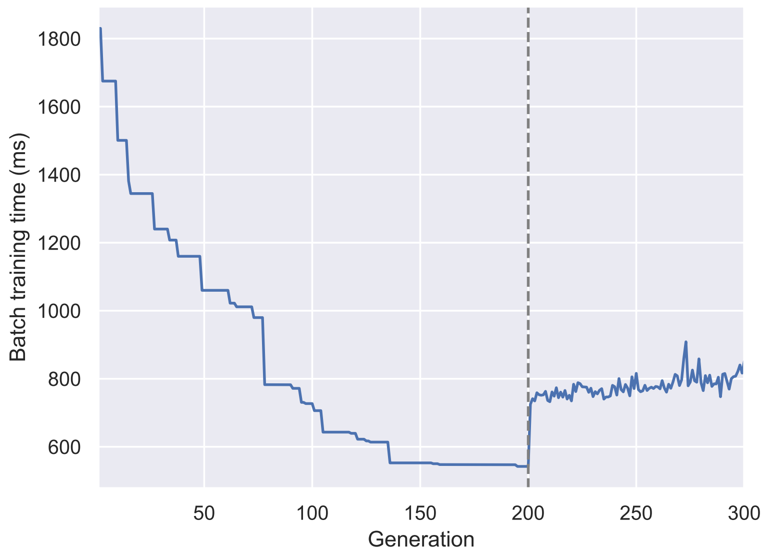
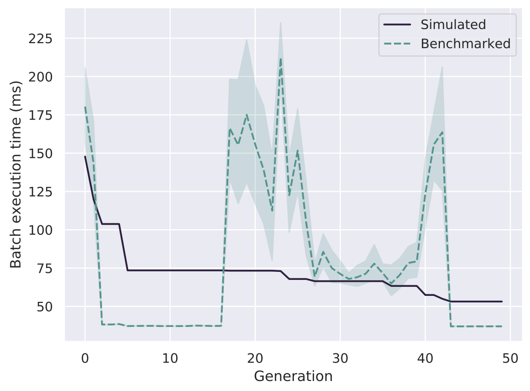


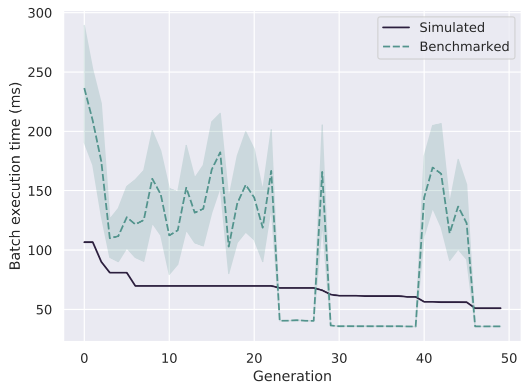
Figure A.5: Optimization of Inception on Malvik with limited available memory, optimizing for 200 generations against the simulator, and 100 generations against benchmarked times. The jump at the transfer point is larger than for ResNet-50 runs because the numerical accuracy of the simulator is lower for this network, as observed in Section 4.3.5.

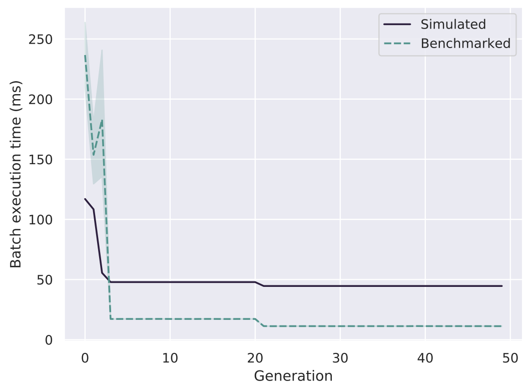
A.2 E5 — Comparison of Simulation and Benchmarks

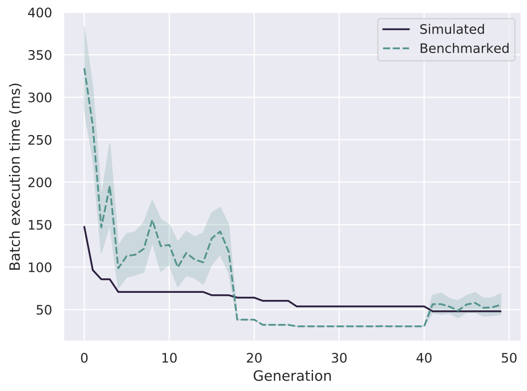
In this experiment, presented in Section 4.3.5, a variety of configurations for a single network were created by running an optimization process using a genetic algorithm, and checkpointing the best network of generations at a given interval. These configurations were then transferred to the real hardware, and the batch execution time was benchmarked. The result plots show a comparison between simulated and benchmarked batch execution times. Runs for different networks are separated in the subsections below.

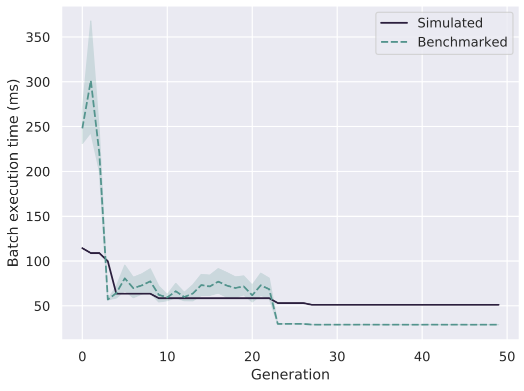
A.2.1 AlexNet

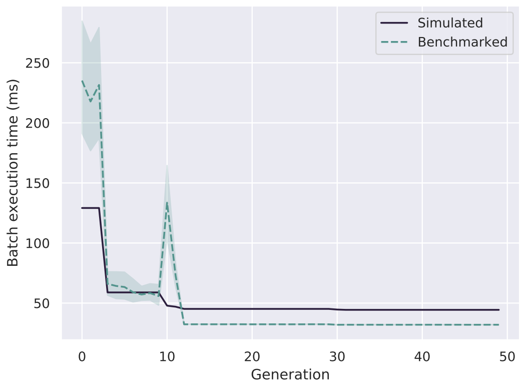


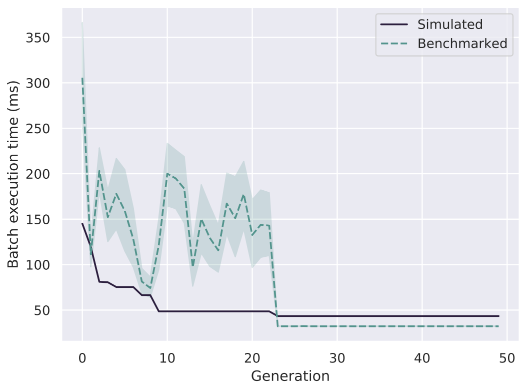


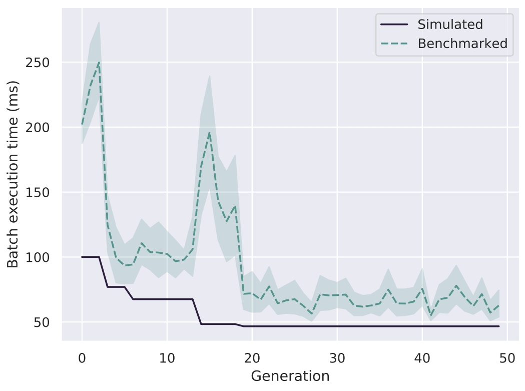


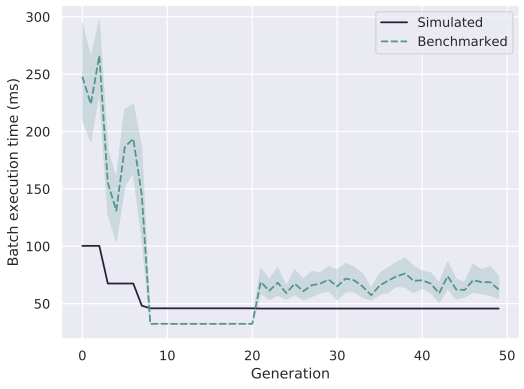


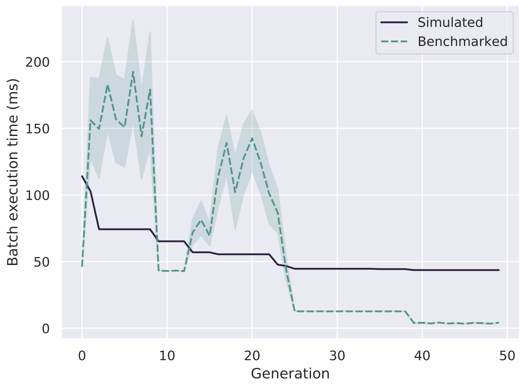

Simulated
Benchmarked
Batch execution time (ms)
Generation


Simulated
Benchmarked
Batch execution time (ms)
Generation

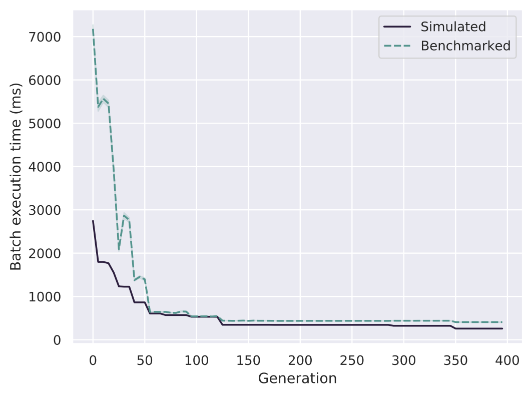

Simulated
Benchmarked
Batch execution time (ms)
Generation

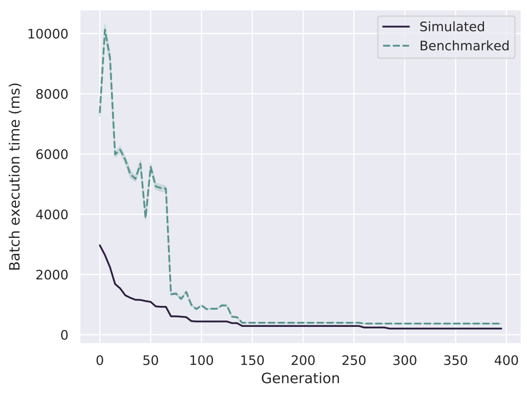

Simulated
Benchmarked
Batch execution time (ms)
Generation

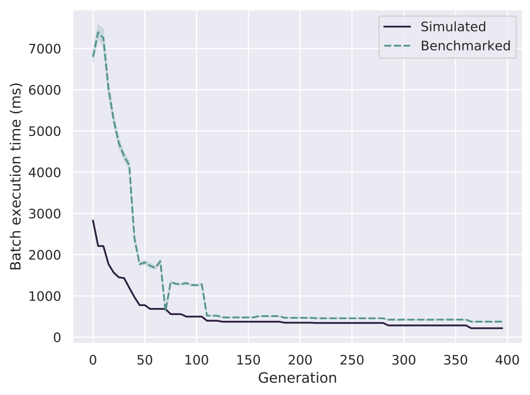

Simulated
Benchmarked
Batch execution time (ms)
Generation

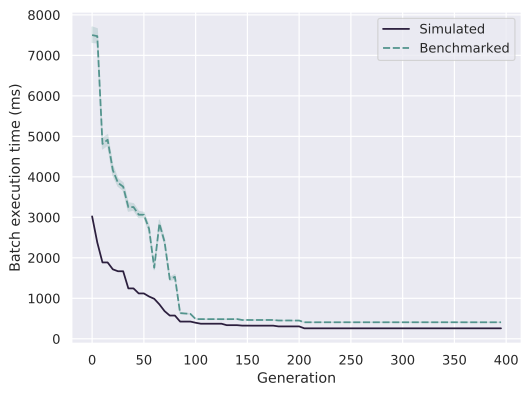

Simulated
Benchmarked
Batch execution time (ms)
Generation

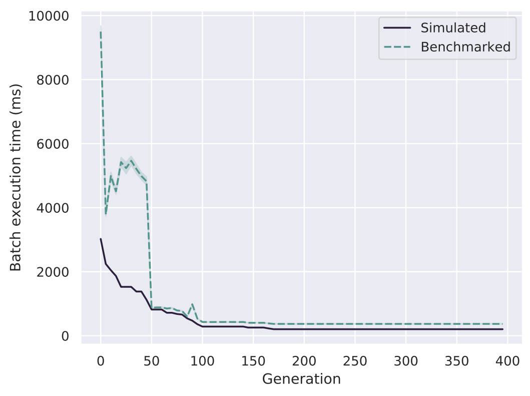
A.2.2 ResNet-50

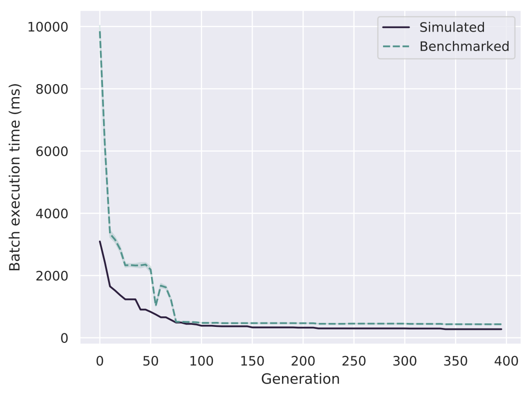


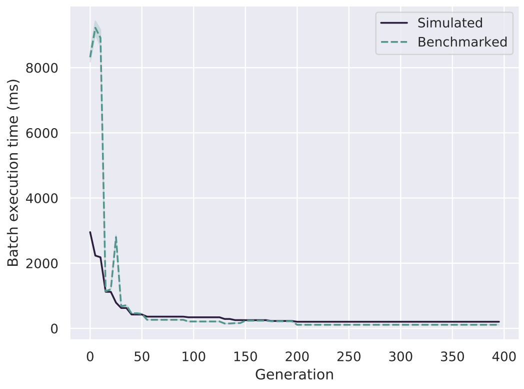


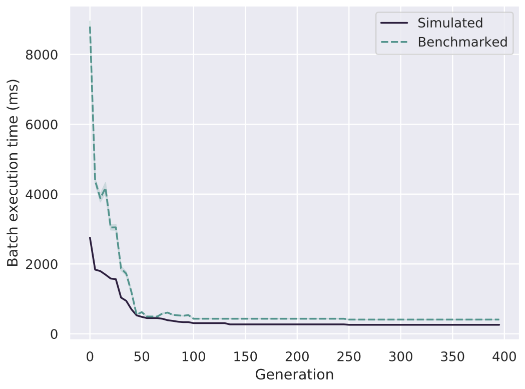


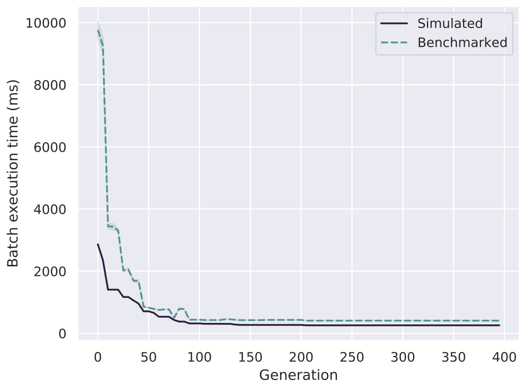


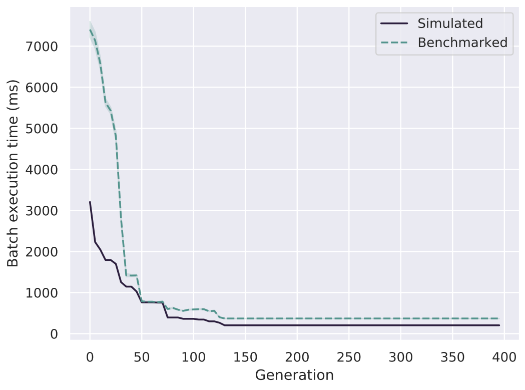




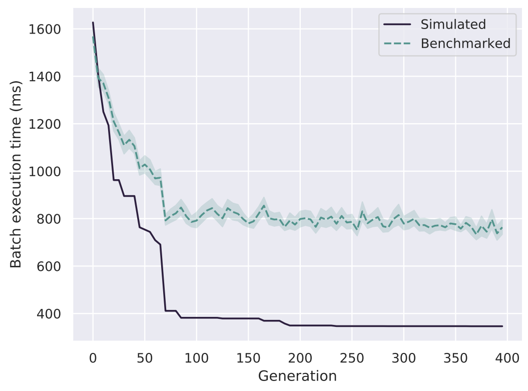

Simulated
Benchmarked
Batch execution time (ms)
Generation

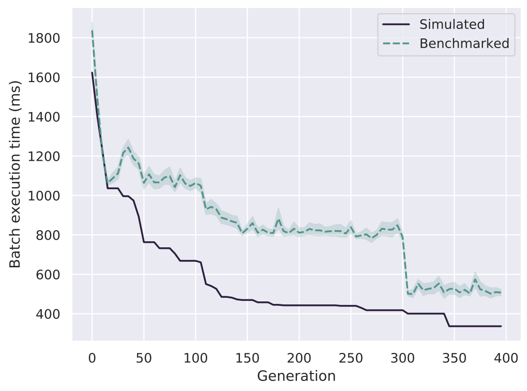

Simulated
Benchmarked
Batch execution time (ms)
Generation

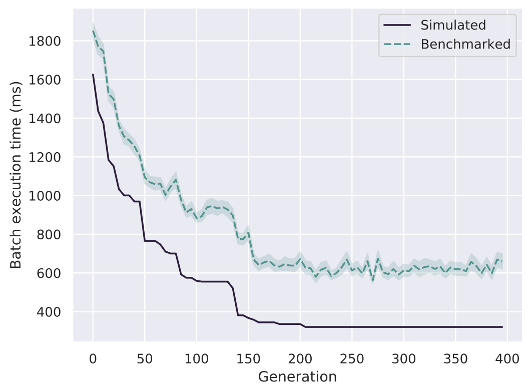

Simulated
Benchmarked
Batch execution time (ms)
Generation

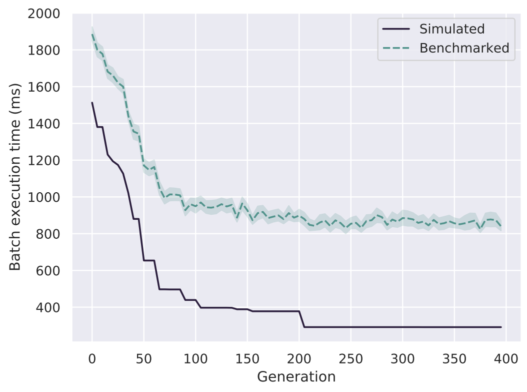

Simulated
Benchmarked
Batch execution time (ms)
Generation

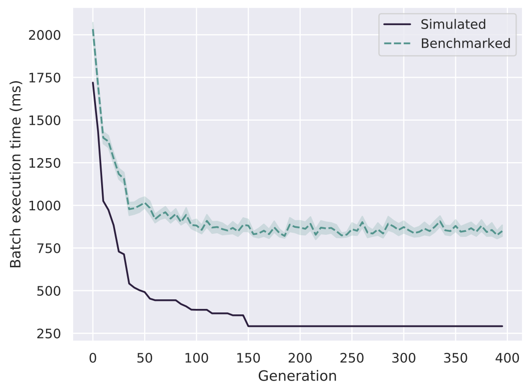
A.2.3 Inception V3

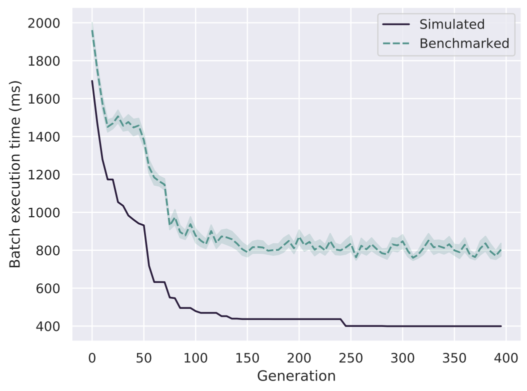


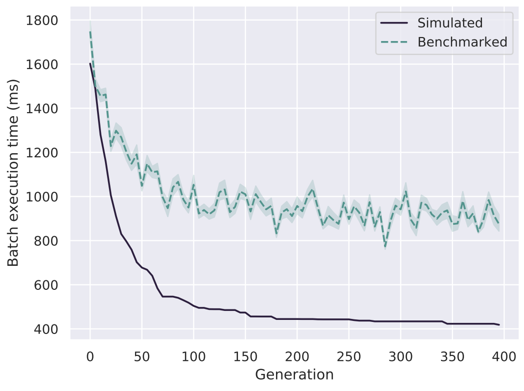


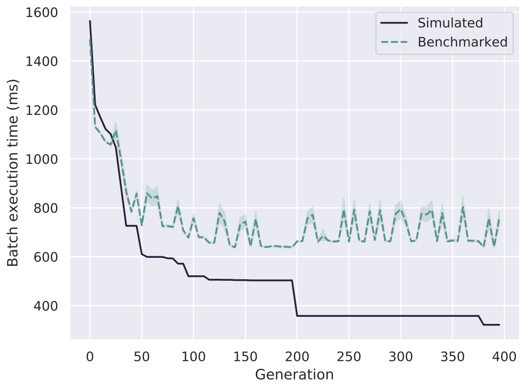


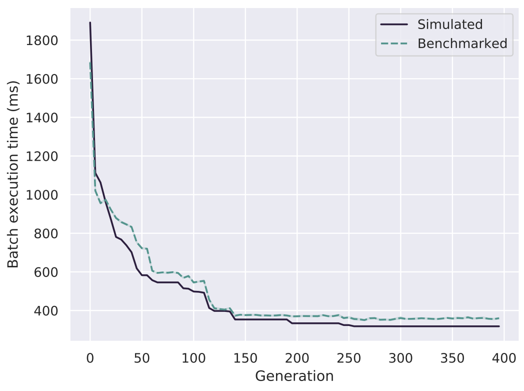


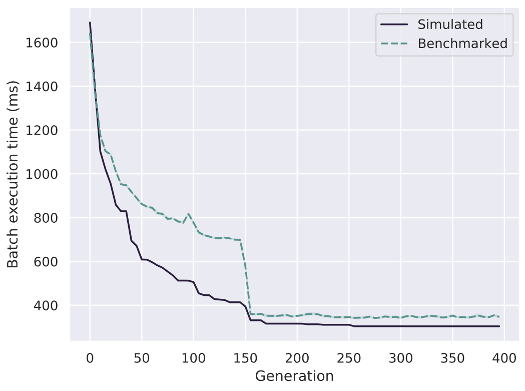





Simulated
Benchmarked
Batch execution time (ms)
Generation


Simulated
Benchmarked
Batch execution time (ms)
Generation


Simulated
Benchmarked
Batch execution time (ms)
Generation


Simulated
Benchmarked
Batch execution time (ms)
Generation