Utilizing heterogeneity to allocate ML tasks for increased efficiency

Master's thesis in Computer science and engineering

Lukas Carling
Max Villing

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Utilizing heterogeneity to allocate ML tasks for increased efficiency

Lukas Carling
Max Villing

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Utilizing heterogeneity to allocate ML tasks for increased efficiency

Lukas Carling
Max Villing

© Lukas Carling, Max Villing, 2023.

Supervisor: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering
Advisor: Muhammad Waqar Azhar, Department of Computer Science and Engineering
Examiner: Risat Pathan, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Utilizing heterogeneity to allocate ML tasks for increased efficiency

Lukas Carling, Max Villing
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

There is a growing interest in using heterogeneous hardware and resource allocation to boost the efficiency of software applications. Proper use of both imposes additional burdens on software development. We look at characterizing some common machine learning tasks with regards to CPU-GPU systems, specifically for the NVIDIA Orin, in order to try and predict what conditions will give the highest performance and energy-efficiency. We then take an iterative approach for allocating said tasks to hardware and selecting resources based on our characterization, with either performance or energy-efficiency as a goal. We find that while there is room for improvement on per-task predictions there are various possibilities to gain significant benefits to performance and energy by properly utilizing hardware heterogeneity and resource allocation. Additional exploration of domain specific accelerators such as tensor cores shows significant potential for accelerating convolutions.

Keywords: heterogeneous hardware, resource allocation, convolutional neural network, NVIDIA, Jetson ORIN, tensor cores, machine learning.

Acknowledgements

We would like to express our gratitude to everyone who has supported us during this project. We want to thank our supervisor(s) Pedro and Waqar from Chalmers who have given valuable suggestions and supported us through out work. Additionally we would also like to thank Jessica Villing for assistance during report-writing.

Lukas Carling & Max Villing

Gothenburg, 2023-06-21

Contents

List of Figures **xi**

List of Tables **xii**

1 Introduction **2**
- 1.1 The Need for Heterogeneity . . . 2
- 1.2 Problem Formulation and Research Question(s) . . . 3
  - 1.2.1 Purpose . . . 3
  - 1.2.2 Contributions . . . 3
  - 1.2.3 Scope . . . 4

2 Background **5**
- 2.1 Heterogeneous Systems . . . 6
  - 2.1.1 Application requirements . . . 6
  - 2.1.2 Heterogeneous Components . . . 6
  - 2.1.3 Memory Systems . . . 7
- 2.2 Related Work . . . 7

3 Theory **9**
- 3.1 Datatypes . . . 9
- 3.2 Governors . . . 9
- 3.3 Frequency scaling . . . 9
- 3.4 Understanding hardware characteristics . . . 11
  - 3.4.1 Central Processing Unit (CPU) . . . 11
  - 3.4.2 Graphical Processing Unit (GPU) . . . 12
- 3.5 Tensor cores . . . 12
- 3.6 Layers . . . 13
  - 3.6.1 Channels . . . 13
  - 3.6.2 Convolution . . . 13
  - 3.6.3 Pointwise . . . 14
  - 3.6.4 Depthwise . . . 14
  - 3.6.5 Average Pooling . . . 14
  - 3.6.6 Softmax . . . 15
  - 3.6.7 Fully Connected . . . 15
  - 3.6.8 ReLu . . . 15
  - 3.6.9 Batch Normalization . . . 15

4 Implementation 16
4.1 Heterogeneous System Characteristics 17
4.2 Task Scheduling 17
4.2.1 Machine Learning (ML) 19

5 Design 20
5.1 Hardware 20
5.2 Software 20
5.2.1 Layers 21

6 Method 22
6.1 Languages and Frameworks 22
6.2 ORIN 22
6.3 Scripts 22
6.4 Building a CNN model 23
6.4.1 ResNet-11 23
6.4.2 MobileNet 23
6.4.3 Batch Normalization 24
6.5 Manual Resource Allocation 24
6.6 Metrics 25
6.6.1 Performance 25
6.6.2 Throughput 26
6.6.3 Energy 26
6.7 Monitoring and measuring 26
6.7.1 Performance Counters 26
6.7.2 CPU Shielding 27

7 Results 28
7.1 ResNet Layers Characteristics 28
7.2 MobileNet 29
7.2.1 Pointwise Convolution 29
7.2.2 Depthwise Convolution 31
7.2.3 Batch Normalization 33
7.2.4 ReLu 35
7.2.5 Remaining Tasks 38
7.2.6 Overall Assessment 38
7.3 Layer Trends 39
7.4 Impact of Frequency 40
7.5 The impact of tensor cores 41

8 Discussion 43
8.1 Frequency scaling 43
8.2 Implementation of kernels on CPU and GPU 43
8.3 A homogeneous execution 43
8.4 Manual Configurations 44
8.4.1 Varying Frequency 44
8.5 Differences Between Computers 44

8.6 Ethics 45
8.7 Sources of error 45

9 Conclusion **47**

Bibliography **48**

List of Figures

3.1 Result (time) of running a 2048x2048 GEMM on CPU with varying frequencies. 10
3.2 Result (energy) of running a 2048x2048 GEMM on CPU with varying frequencies. 11
3.3 Pseudo code for a convolutional layer. 14
3.4 C++ implementation for a fully connected layer 15

7.1 Line graph illustrating the performance of each pointwise convolution 30
7.2 Line graph illustrating the efficiency of each pointwise convolution . . 31
7.3 Line graph illustrating the performance of each depthwise convolution 32
7.4 Line graph illustrating the efficiency of each depthwise convolution . . 33
7.5 Line graph illustrating the performance of each batch normalization . 34
7.6 Line graph illustrating the efficiency of each batch normalization . . . 35
7.7 Line graph illustrating the performance of each batch normalization . 36
7.8 Line graph illustrating the efficiency of each batch normalization . . . 37
7.9 Bar graph illustrating the overall performance of the network 39
7.10 Bar graph illustrating the overall efficiency of the network 40
7.11 Time and energy of tensor cores compared to cuda cores. 41
7.12 Fractional difference (y-axis) for running GEMM with tensor cores and comparing with CPU implementation for validation over varying matrix size (x-axis). 42

List of Tables

5.1 Hardware details for the target architecture (Chalmers) and personal machines used for testing. 20

7.1 Table showing the performance for the remaining tasks in the MobileNet (measured in ms) 38
7.2 Table showing the energy consumption for the remaining tasks in the MobileNet (measured in mw) 38
7.3 Different layers and their corresponding execution time on CPU and GPU for a N = 1024 ResNet. CPU frequency is 2.2 GHz and GPU frequency is 1.3 GHz. 41

Glossary

Computing Terminology

Due to the breadth of the computing world many common terms have multiple different definitions for different use cases. In this paper, we will be using these common terms in the following ways:

Convolutional Neural Network (CNN) A neural network that makes extensive use of convolution. Common in computer vision applications.

General Matrix Multiply (GEMM) Algorithm for performing matrix multiplication.

Machine Learning (ML): A application composed of different heterogeneous layers that process data to produce a deterministic result. Typically exhibit large degrees of parallelism.

Task: A component of work that makes up part of an application. Tasks are either in dependent meaning they can be executed in any order or dependent meaning they require some other tasks to be completed before they can be started. In this project, a task will be considered to be a layer of a CNN.

Scheduling: The act of deciding when and on what hardware tasks should be executed on in order to meet deadlines.

Dispatching: The act of tasking a specific compute unit with executing a given task.

Many Integrated Cores (MIC): A term introduced by Intel, referring to a series of micro-architectures that integrated many physical cores onto a single integrated circuit.

Graphics Processing Unit (GPU): A specialized processor originally designed to accelerate graphics rendering.

Compute Unit (CU): A piece of hardware capable of performing computations on data given to it.

Instruction Set Architecture (ISA): A part of the model that defines how software controls the compute unit it is running on.

Arithmetic Intensity: The ratio of arithmetic operations to memory operations in a computational task or algorithm.

1 Introduction

Heterogeneous systems are of great interest in the future creation of high performance efficient applications. However, making proper use of heterogeneity introduces an additional burden on the developer who needs to familiarize themselves with the characteristics of their particular hardware and how to best fit their software to it. We wish to look into the development of a framework that can dynamically make these decisions during runtime. In order to accomplish this we investigate the performance characteristics of ML applications and analyze manual configurations in order to develop insights that can power future heuristics and frameworks.

1.1 The Need for Heterogeneity

With the coming end of the Moore era and Dennard Scaling, the quest for increased computer performance has to take a different turn. We can no longer rely on ever more powerful processing cores. Instead parallelism has become more and more common as the focus shifts to doing more things at once instead of doing one thing faster [1]. But even parallelism has its limit, Amdahl's law provides a cap on how much an application can be sped up via parallelization. This is where heterogeneous systems enter the picture.

Heterogeneous system trade the general performance of homogeneous systems for specific types of performance improvements. Dedicated heterogeneous accelerators can perform certain types of work much faster. At its most extreme we see Application Specific Integrated Circuits (ASICs), which are purpose built for a specific application and can perform no other type of work. But less specific forms also exist, Graphics Processing Units (GPUs) for example are accelerators focused on performing large amounts of parallel operations at once without the general computing capabilities of traditional CPUs.

1.2 Problem Formulation and Research Question(s)

"Which improvements can hardware heterogeneity have on ML applications utilizing heuristics?"

"Will automatic mapping of a ML application to heterogeneous hardware based on predetermined characteristics allow for a noticeable energy reduction while maintaining QoS?"

"To what extent can the performance of a ML application be increased while preserving minimal energy costs exploiting hardware heterogeneity?"

"Which metrics are relevant for measurements when exploring ML applications on heterogeneous hardware?"

"What are the problems / trade-offs with a heterogeneous approach for ML applications?"

1.2.1 Purpose

We look at characterizing various commonly used tasks in Convolutional Neural Networks such as convolution or batch normalization. We look at how they perform on CPUs and GPUs absent any device specific optimizations in order to as accurately as possible determine their different performance characteristics. Based on this we construct theoretically optimal combinations of frequency and hardware to run a given task as fast or efficiently as possible. We then evaluate these theoretically optimal configurations against standard configurations to see if they actually realize their theoretical advantage. We will then try to use those findings to lay the foundations for the development of heuristics that can aid in mapping software tasks to heterogeneous hardware. At no point will we modify hardware directly, we are looking purely at the problem from a software perspective.

1.2.2 Contributions

In this thesis we have investigated the performance and energy-efficiency of resource allocation for a CPU-GPU heterogeneous system. The contributions from our work are outlined below.

We have evaluated performance and energy metrics in the context of resource allocation for a heterogeneous system. The involved metrics are execution time, throughput, energy-efficiency, arithmetic intensity among others. By analyzing these values we gained an insight in how they have an impact time and energy.

We have explored and attempted to create a foundation for the characterization of resource allocation which can be applied in a heuristic. Multiple runs have been done in order to understand how for example frequency scaling or different hardware has an impact on energy and time.

Overall the research from this project is supposed to give an insight in how various tasks could be allocated to heterogeneous hardware via for example a heuristic with

the aim to reduce energy and execution time.

1.2.3 Scope

Due to the breadth of the field and the limited time and resources available to us this thesis will only focus on the characterization of ML layers to create and test theoretically optimal configurations with regards to performance and energy consumption. By looking at the results from these configurations we then discuss insights that can be used for the future development of heuristics for resource allocation in heterogeneous systems. The original intent was for us to explore that as well but that proved to be infeasible in the time provided.

Additionally, while we will not completely disregard the accuracy of the neural networks we are running, our primary interest lies in how the act of running them can be improved or made more efficient. We will note when we think our results introduce some level of inaccuracy to their execution but we are not looking to make "better" convolutional neural networks. We are simply using them as a useful test case for how hardware characteristics can affect task execution.

2

Background

Heterogeneity is a word commonly thrown around in the modern computing world and has many potential definitions. In order to avoid ambiguity and provide clarity in this text we will define a heterogeneous system as any computer setup that makes extensive use of compute hardware with significant different characteristics and capabilities from one another that make them best suited for different tasks. The archetypical example (and the one we will focus on for the rest of this text) is a dual CPU-GPU setup where most work is conducted on the CPU but some is offloaded on the GPU to take advantage of its massive parallelism and optimization for floating point operations. The idea is that such a system will be able to achieve high performance at a lower cost than a homogeneous system which uses one all-purpose compute unit. In a more traditional homogeneous system one would need to upgrade the capabilities of the system as a whole to gain more performance, which can be highly inefficient if say one only needed to perform faster floating point division and now has to use a CPU which is overkill for the majority of work expected of it. In a heterogeneous system one could instead procure a piece of specialized hardware (commonly called a accelerator) for the needed task and keep using the other hardware, saving on both monetary (since the specialized hardware would overall be less capable than the homogeneous) and power (accelerators often consume less energy since they don't need the all purpose capabilities of a CPU such as caches) costs.

Similarly we will also define a heterogeneous framework as a programming framework intended to help developers best make use of heterogeneous hardware in one form or another. This does not require the framework to cover all kinds of heterogeneous hardware (and indeed such a thing would be nearly impossible given how diverse such hardware can be) nor does it require the framework to be able to handle concurrent use of multiple heterogeneous devices. Instead we are talking about the likes of CUDA, a framework NVIDIA provides that allows developers to easily write code that will be run on NVIDIA GPUs and to define when and how the GPUs will be used as well as how to transfer memory to and from them. A heterogeneous framework is merely a software framework meant to help development on a non-CPU platform under our definition.

2.1 Heterogeneous Systems

While a great many types of heterogeneous accelerators have been developed their use imposes a significant burden on software developers compared to more traditional homogeneous hardware. Efficient mapping of tasks with differing characteristics to hardware with differing properties is a non trivial problem and one that has significant implications for both performance and efficiency. As such there is great interest in the development of frameworks that can aid in this process and reduce the burden of heterogeneous development. Some examples of frameworks are:

Single ISA Frameworks (OpenMP, pthreads, MPI) Multiple ISA Frameworks (OpenCL / OpenACC, CUDA, StarPU)

Here we differentiate frameworks based on whether their compute units use a single shared ISA or multiple different ones. Single ISA mainly support heterogeneity by allowing homogeneous compute units to better make use of parallelism. While this makes them flexible as they are not bound to any specific type of hardware their general nature means they can't extract the performance of more specialized options. Multiple ISA frameworks are those where different compute units use different ISAs to perform their operations. Typically this involves having a normal CPU running operations aided by a different accelerators it can dispatch tasks to. CUDA for example is a framework for having a CPU send tasks and data to NVIDIA GPUs. This makes multiple ISA frameworks powerful since they can make full use of accelerators but also makes them less generic, CUDA requires that a NVIDIA GPU be used while OpenMP does not care about the hardware that uses it.

2.1.1 Application requirements

Certain applications, for example video playback, have certain service performance requirements. Processing faster than these requirements add no extra value to the user. Recent work in the department has looked at these areas and produced several research papers such as SLOOP [2], SaC [3], and Task-RM [4] . Several research artifacts of run-time systems for each of these projects were developed. So for certain applications performance is not the primary concern and heterogeneity may be used to gain energy efficiency or to improve some other metric.

2.1.2 Heterogeneous Components

Modern heterogeneous processors consist of a variety of different specialized computing elements, such as general-purpose cores of various type, GPUs, vector co-processors, and accelerators. While the proper use of these elements can greatly improve the performance of a system another important consideration is the needs of the application.

Among the accelerator options GPUs in particular are popular. The primary purpose of the GPU is to provide high-throughput data-parallel computing with a large number of threads, which is ideal for large-scale data-parallel applications requiring

high computational density and simple logic branching. The CPU features a complicated logic control unit and a large-capacity cache, has a short data transmission latency, is adaptable to a variety of tasks, and excels at complex logic operations. By pairing the two together the CPU+GPU architecture can combine the strength of both, using the GPU to process data-intensive parallel tasks while the CPU handles complex logical transaction processing. This allows the CPU and GPU to fully exploit their respective advantages and maximize the utilization of heterogeneous systems, saving on computational expenses and energy. [5]

2.1.3 Memory Systems

Another important facet of heterogeneous systems is the type of memory they use. Generally speaking they can either make use of a shared memory space or distributed memory. In shared memory systems all compute units have access to the same memory and can read and write to it at their leisure. While this guarantees that there is no overhead from having multiple copies of memory it also means all units share memory bandwidth and complicated control logic might be necessary to ensure everyone has access to the memory without the risk of one unit overwriting the work of another. In distributed memory systems compute units have their own memory units and use communication to share information with one another. While this ensures there is no possibility of compute units overwriting each others memory and that bandwidth won't be a limiting factor it also means a lot of data has to be duplicated across systems and adds a communication overhead should different compute units need to cooperate.

Nowadays it is common for mixed memory systems to be used, where compute units have distributed memories but there is also some form of shared memory, either across the whole system or for groups of compute units.

2.2 Related Work

As discussed in [6] homogeneous multicore systems have shown to provide adequate performance for certain tasks, but as Amdahl's law states when the available parallelism decreases the homogeneous architecture cannot provide the desired speedups. Instead heterogeneous systems are explored as certain components excel at different scenarios. The authors of [6] claim that heterogeneous computing (HC) is an alternative to the homogeneous limitation but that there are still issues and areas to be researched.

Previous work relating to resource allocation [7] also states that resource allocation is vital for improving system performance when it comes to big data processing which ML quickly can become. The authors of [7] also claim that resource capacities are unused while others are exhausted which previous algorithms did not consider. This leads to resource wastage [7] which can be reduced with proper characterization and improved algorithms or heuristic which take this point in to consideration.

While allocation can be done manually it would be ideal to have a framework or a

system which dynamically allocates resources. An example of such an idea is StarPU which is a runtime system for heterogeneous multicore architectures that provides support for CPU/GPU implementations. StarPU handles the allocation of tasks to specific compute units and allows the programmer to focus on the algorithmic aspects of their work. One particular feature of StarPU is that it supports different implementations of a task for different hardware, meaning a programmer can write optimized code for all their different processing units and not have to worry than a CPU optimized version will be used on a GPU or vice versa. StarPU also maintains a dynamic estimation of the performance of a task on different compute units based on past executions. [8]

3

Theory

3.1 Datatypes

In the project different data types will be used to represent matrices. This chapter's purpose is to give an insight in which ways the types differ and how this affects the results. The main focus has been on 16-bit and 32-bit floating point values but 32-bit integers and 64-bit double precision values have been studied as well. There are variances in energy and area when different arithmetical operations are done, for example the energy cost of doing a 16-bit floating point multiplication is approximately 30% as expensive as performing a 32-bit one [9].

In the context of a RTX 3060 Ti the peak theoretical speed is 16.2 TFLOPS for both 16 and 32-bit [10] but there would still be a rather large energy efficiency potential by doing half-precision operations instead. The bandwidth of the RTX 3060 Ti is 448.0 GB / s [10] which would allow for $5.6 * 10^{13}$ 64-bit, $1.12 * 10^{14}$ 32-bit or $2.24 * 10^{14}$ 16-bit values per second. Assuming the algorithm is highly memory bound this can have a large impact on performance.

3.2 Governors

Governors are software modules or algorithms that are implemented in the operating system to manage and control the frequency of the CPU. Their responsibility is to dynamically alternate the frequency to accommodate the workload of the machine [11]. Since this project is focused on the Linux operating system the following list is an example of Linux governors [11].

- **Performance** sets the frequency of the CPU to the highest available frequency.
- **Powersave** Sets the CPU statically to the lowest frequency available.
- **Ondemand** Sets the CPU frequency depending on the current system load.

3.3 Frequency scaling

Generally when talking about hardware such as a CPU or a GPU there is a specification on what their frequency is. That does not mean that the component runs

at that speed at all times, for instance there are governors controlling the desired frequency of the CPU [12] meaning it can change depending on workload or other factors. The main idea of frequency scaling is to optimize the performance and power consumption depending on the current workload. Although this is done automatically by the aforementioned governors, Linux allows for manual configuration of frequency on a software-level. In figure 3.1 there is an example of what running a GEMM with different frequencies on the CPU looks like in terms of time.

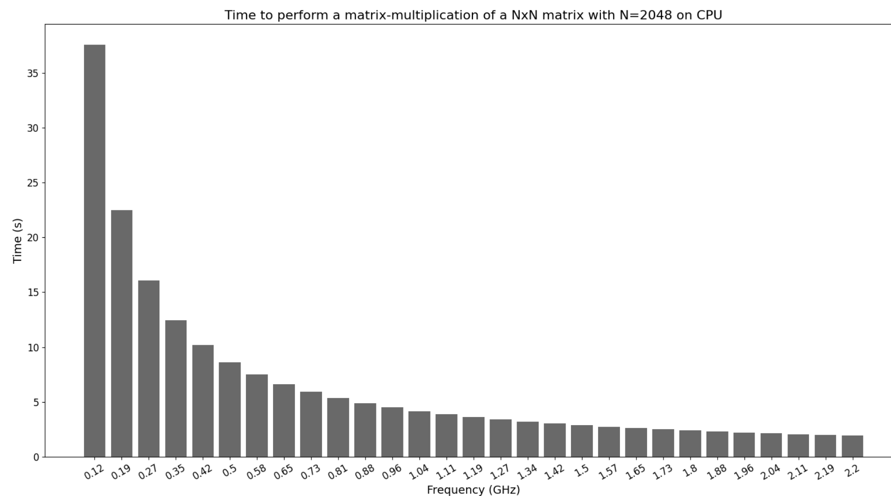

Figure 3.1: Result (time) of running a 2048x2048 GEMM on CPU with varying frequencies.

While a significant difference can be seen in time between the first two frequencies the larger frequencies seem to not yield as much of a performance increase, which seems similar to $-log(x)$. This means that although frequency is increased there is no corresponding performance gain. With a larger frequency the energy consumption is also increased which means that the total energy might be higher for larger frequencies even if they are faster. This can be seen in figure 3.2

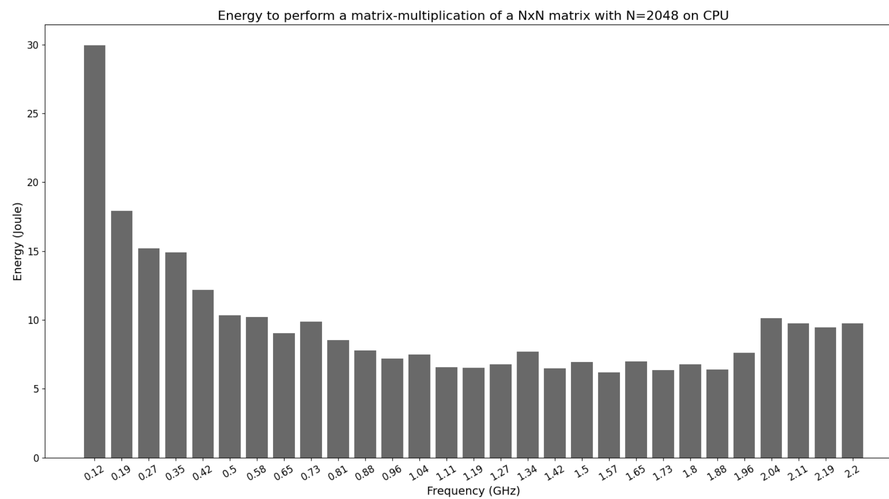

Figure 3.2: Result (energy) of running a 2048x2048 GEMM on CPU with varying frequencies.

3.4 Understanding hardware characteristics

It has been specified that this project focuses on a CPU + GPU system and although those terms might be widely adopted this chapter will explain further into detail what characteristics are of interest for each component.

3.4.1 Central Processing Unit (CPU)

Early computers were sometimes referred to as "fixed-program computers" as they had to be physically rewired in order to do different tasks. In order to tackle the time and effort problem with rewiring, "stored-program computers" were introduced with the idea to allow a computer to perform a certain number of instructions. In combination with this new type of computer the term Central Processing Unit (CPU) became more widely adopted as a computer could now execute programs without the need for rewiring. This could be why the CPU is often called the brain of the computer because it executes the instructions of a computer program such as arithmetic, logic, controlling and I/O operations [13].

In contrast to the GPU, the CPU is more general purpose as there is a need to perform more arbitrary operations for the functionality of a computer. Instead of specializing in great parallelism, although with 8 or 12 cores it does offer some, it specializes in doing various operations. The peak performance for a CPU would be something similar to: $3.8 * 8 * 2 * 8 = 486.4$ GFLOPS with 8 cores and for 1 core it would be approximately: $3.8 * 1 * 2 * 8 = 60.8$ GFLOPS [14].

3.4.2 Graphical Processing Unit (GPU)

Graphical Processing Units (GPUs) were originally developed to handle complex, real-time calculations required for rendering graphics and images. Different from CPUs which mainly focus on general-purpose computations, GPUs are specialized in doing great work in parallel which allows for a vast number of tasks to be completed simultaneously. The GPU consists of thousands of smaller processing cores called shader units, which is what creates the possibility for such parallelism [15]. For the NVIDIA cards which were mainly used in this project the equivalent would be called CUDA cores [16].

To give an example the RTX 3060 Ti has 4864 shading units and 152 tensor cores operating at a frequency between 1410 and 1665 MHz [10]. Although the cores run at roughly 40% of the speed a CPU core operate at, the parallelism would allow for a peak performance of approximately $1.4 * 4864 * 2 = 13.6$ TFLOPS if all shader units were working simultaneously. If only one core could run the performance would instead be closer to $1.4 * 1 * 2 = 2.8$ GFLOPS [14].

3.5 Tensor cores

Since the NVIDIA ORIN is equipped with an integrated Ampere GPU which includes two Graphic Processing Clusters (GPCs), there are up to 16 Streaming Multiprocessors (SMs). Each SM contains 128 CUDA cores and four 3rd generation tensor cores [17] for the architecture used in this project (Ampere). Tensor cores allow for a higher throughput and faster execution time by utilizing half-precision or mixed-precision computing [18].

The tensor cores operate on 16-bit floating point values and perform a full-precision (64-bit) multiplication and accumulate the results and store them in a 32-bit floating point value.

This means that by converting 32-bit floating point (FP) values to 16-bit values, execution time could be reduced at the cost of the accuracy you lose when converting the values. Although there is research [19] suggesting that accuracy can be restored while still surpassing 32-bit FP peak performance.

A 3rd generation tensor core operates on matrices of sizes where the rows and columns are multiples of four. This is because the matrix is broken down into 4x4 matrices which are operated on simultaneously by utilizing a technique called "warp-level matrix operations" [20].

A limitation with tensor cores is that they operate on matrices which size is a multiple of eight meaning that if the input matrix has other dimensions tensor cores can not be fully utilized. For smaller matrices this issue becomes more apparent when tensor cores are not used efficiently but as the matrix size increases the difference becomes less noticeable. It is still something one should have in mind when preparing data though.

3.6 Layers

Considering an unknown function $f_a : \mathbb{X} -> \mathbb{Y}$ a neural network decomposes the function in to a number of more simple functions called layers ($f_a = f_1 * f_2 * f_3 ... * f_N$) [21]. The different tasks which will be mapped to hardware can be divided into these layers which are the building blocks of a model. A model might use ten layers which means there will be ten tasks to be assigned heterogeneously to a hardware component. This section will further explain how the different layers work, including additional factors which become relevant in the analysis later in the report. The layers in this scenario are all two-dimensional (2D).

3.6.1 Channels

Not a layer per see but a common term used when discussing layers. Channels indicates the z depth of a matrix. A common use case for having depth in convolutions is to have each channel represent a RGB value for a given pixel. These can then be combined to express the true color of a pixel. In neural networks it is common to deepen input to even more channels or to compress several channels into one. This allows for the network to more precisely learn information about the input data.

3.6.2 Convolution

The convolutional layer, is considered to be an essential block for a CNN where it receives an input image and outputs a new image. Assuming the input image can be represented as $C^{(m-1)}$ (consisting of A_m channels) and the output image as $C^{(m)}$ (of B_m channels) the output can be calculated [21] using:

$$A_o^{(m)} = g_m \sum_k W_{ab}^{(m)} * A_k^{(m-1)}$$

where the multiplication is considered to be the convolution operation [21]. In the scope of this project this layer has been implemented in both CUDA and C++. An example pseudo code can be found in figure 3.3

```
1   // N represents the width / height of a NxN matrix.
2   for i from 0 to rr do
3       for j from 0 to rc do
4           for k from 0 to ic do
5               for l from 0 to oc do
6                   tmp = 0
7
8                   for m from pl to pr do
9                       for n from pt to pb do
10                          x = j + m
11                          y = i + n
12                          if x > 0 and x < ic and y > 0 and y < ir do
13                              // Calculate index1 and index2 with variables.
14                              tmp += input[index1] * weight[index2]
15                          end if
16                      end for
17                  end for
18
19                  // Calculate index3 with variables
20                  result[index3] = tmp
21
22              end for
23          end for
24      end for
25  end for
```

Figure 3.3: Pseudo code for a convolutional layer.

3.6.3 Pointwise

A pointwise convolution is a special form of convolution where the weights are of the form 1x1xN where N is the channel depth of the input. A pointwise convolution has no stride and the kernel is overlaid over every element of the input. Pointwise convolutions are used in conjunction with depthwise convolution to achieve nearly the same result as normal convolution with only a slight decrease in accuracy thanks to performing much fewer multiplications than a normal convolution. Typically multiple 1x1xN kernels will be used, this will produce a output of channel depth M where M is the amount of kernels used.

3.6.4 Depthwise

A depthwise convolution is a special form of convolution that applies a single weight of dimensions NxM to each input channel. In normal convolution the filter has the same channel depth as the input and and allows for channels to be freely mixed to produce the output whereas in depthwise convolution each channel is kept separate. The combination of depthwise convolution followed by pointwise convolution produces results similar to normal convolution at a slight accuracy cost and a lower amount of operations (how lower depends upon the weight dimensions).

3.6.5 Average Pooling

An average pooling operation moves an NxM filter across the input matrix and finds the average sum of the values in the input currently overlapping with the filter. The stride of a Average Pooling layer determines how many "steps" the filter moves after each operation, a stride of 1 would mean that the filter moves 1 tile to the right. Average pooling can be used to "shrink" an input matrix while still retaining some information from each element in it. Typically average pooling is seen near the end

of a neural network but it can be used anywhere. A related layer is the so called Max Pooling which outputs the maximum value instead of the average.

3.6.6 Softmax

Softmax is an operation used to convert data from one form to another. This is typically done to shrink values to fit better on a scale or otherwise make output more usable. In the case of softmax it converts a row of the input data into a normalization of the sum of the exponentials of each number in the row. Commonly softmax is used at the very end of a neural network so that the processed data can be easily expressed in terms of probabilities.

3.6.7 Fully Connected

A fully connected layer is a operation which maps every element of the input to every element of the output. Figure 3.4 illustrates its pseudo code. Fully connected layers are useful fgr learning non linear information regarding the input.

```
for (int i = 0; i < weight_rows; i++) {
    t sum = 0;
    for (int j = 0; j < weight_cols; j++) {
        sum+= input[j] * weight[(i * weight_cols) + j];
    }
    result[i] = sum;
}
```

Figure 3.4: C++ implementation for a fully connected layer

3.6.8 ReLu

ReLu is a simple classification function that converts all non negative values in the input into 0 while leaving the rest untouched. ReLu is typically run after other layers and prevents problems stemming from negative values. In the context of most neural networks where each element in the output will correspond to a likelihood a negative value has the same bearing as 0 in the end.

3.6.9 Batch Normalization

Batch normalization is a function that normalizes and then rescales all values in the input on a per-channel basis. This is done by first normalizing each value, then adding a arbitrary value and then multiplying it with a different arbitrary value. These arbitrary values can be learnt as training progresses and their role is to convert the values to a new basis. This prevents any value from growing too extreme and speeds up the training of neural networks as a result.

4

Implementation

As heterogeneous computing is an incredibly broad field and we have limited time and resources we will only be looking at systems that use a CPU for most computations and a GPU as a accelerator. It is of course desirable for a system to make full use of its resources at all times in order to achieve maximum performance. However there are constraints that makes this impossible. Some tasks in a program are inherently sequential and can't be performed in parallel, which as Amdahl's law states "the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used", meaning a system with four cores might be limited to running on a single core to run sequentially. Specialized hardware may be able to accelerate them but a system will never be able to use all its resources on them. In addition there are also problems that prevent full resource utilization even for parallel tasks and these challenges grow as the systems do the same. They come from both the hardware side and the software side.

Hardware wise heterogeneous systems contain multiple computing devices with different system architectures, instruction sets, and programming models. Thus heterogeneous systems frequently have different programming models from CPUs. The instruction set architectures of compute elements (Programs) may differ, resulting in binary incompatibility. Compute elements, similarly, may interpret memory in many ways. This includes endianness, calling convention, and memory layout, and is determined by the architecture and compiler used. A solution to this problem is to have the CPU control logic and dictate program flow while delegating tasks to accelerators. This greatly simplifies the system and reduces the risk of conflicts at the cost of making the CPU responsible for smooth functioning [5].

On the software end parallel work requires communication between compute units lest work be wasted. In addition there is the problem of memory access. Where distributed memory systems need time to load data into memory, shared memory systems risks different compute units undoing each others works [5]. This introduces the need for synchronization and its associated logic. Hence even in a hypothetical application that is 100% parallelizable a system with 4 times as many resources would not see a 4 times speed up as some computing time is dedicated towards making the parallel work possible in the first place [5].

For our own software we will mak use of OpenMP for the CPU and CUDA for the GPU, both are frameworks well suited for developing parallel software and have

extensive support networks avaible.

In the following subsections we will look at heterogeneous systems in more detail and explain why they are of relevance to us.

4.1 Heterogeneous System Characteristics

In a heterogeneous system, CPUs with identical architectures may have underlying micro-architectural variances that result in varying degrees of performance and power consumption. Capability asymmetries combined with opaque programming models and operating system abstractions can sometimes lead to performance predictability issues, particularly with mixed workloads. In order to overcome this issue, researchers have proposed a model to analyze the performance of heterogeneous multi-core systems to fairly divide computing jobs. [5]

While splitting data on homogeneous platforms is frequently straightforward, it has been demonstrated that the problem is NP-Complete in the general heterogeneous case. However if the amount of partitions is kept to a modest level it is possible to achieve optimal partitioning that fully balances the load and minimize communication volumes. Further research is necessary to obtain optimal partitioning in a heterogeneous platform. [5]

To handle data splitting resource managers (RM) can be used. They select configurations that minimizes energy under the constraint of meeting a specified Quality-of-Service (QoS) demand, for example, a periodic computational deadline. Heterogeneous multicore architectures, such as ARM big/LITTLE and Intel QuickIA, typically expose a rich configuration space including different processor core types differing in issue width and out-of-orderness, voltage/frequency levels, and the number of cores of a certain type. These con- figurations expose a variety of performance/energy characteristics. [3]

We believe that heterogeneous objectives are the norm in multiobjective optimization rather than the exception. Nevertheless, it is still a largely unexplored topic to understand how each different type of heterogeneity causes specific difficulties to existing multi-objective techniques. [22]

One unconventional issue with heterogeneous systems is how to validate their performance. Given they are specialized towards certain tasks it can be hard to draw comparisons to other systems. In this case we believe it best to compare with homogeneous approaches for solving the same problem. [22]

4.2 Task Scheduling

One challenge with using accelerators is granularity. The more specialized the accelerator the harder it is to use them for a wide variety of general algorithms. As such, while accelerators are desirable and necessary for performance improvements, accelerators with too specific of a focus may be undesirable as the cost of developing

them as they may not be suited for enough different situations and systems. A balance must be struck between specificity and generalizability for a hardware to be practical in real life scenarios.

Another issue is that since accelerators might share data that the general purpose CPU is generating/consuming, maintaining coherency and consistency of the memory poses a challenge.

Specifically, we recognize the difficulty of run-time task scheduling in systems with multiple hardware accelerators: where a task is an abstraction level for a set of instructions that define a primitive, which tends to repeat across the program. User threads can be made up of multiple tasks and those tasks might be best performed by some accelerators in the system. The threads can have fine-grained parallelism leading to lots of dependencies among the tasks, coarse-grained parallelism that requires minimal interactions, or data parallelism that makes them completely independent. Furthermore, control flow changes may change the dependency structure completely during run-time. This clearly indicates the necessity of a run- time task scheduler for such heterogeneous systems, and a rich line of works based on run-time APIs address this challenge.

While the usage of custom schedulers in accelerators has been explored before, the domain of hardware based task scheduling for heterogeneous systems is mostly unexplored and is where we look to focus our efforts. [23]

To avoid the problem of an overly specialized system we will be looking at runtime based task scheduling. In this type of scheduler the runtime defines the efficiency and ability of the environment. It determines the target architectures supported, task scheduling objectives, scheduling methodologies, support for fault tolerance etc. A large number of task-based programming environments have been developed over the past decades, with even established languages like C++ integrating tasks for shared memory parallelism. Cilk, OpenMP, Intel TBB. These were built for shared memory systems but the past few years have seen task-parallel models being built for heterogeneous hardware such as StarPU and distributed systems such as chapel, x10 and HPX.

We observe common denominations among all task-parallel programming environments. They require the programmer to present the application code in a new language, or at best annotate sections of the code for analysis. The former clearly affects portability, while the latter might not give the runtime enough information to extract fine-grained parallelism. Also, to the best of our knowledge, none of aforementioned task-parallel models with software/hardware scheduler caters to heterogeneous systems with specialized accelerators. Our survey findings suggest decoupling programming and task management aspects of a task-parallel programming model to enable portability, and to develop task management hardware for modern day systems.

The core idea of having a large number of accelerators that are shared across applications is realizable only when the run-time scheduling is realized in the system to effectively share the resources.

4.2.1 Machine Learning (ML)

While this move towards heterogeneity has been occurring there has been a great surge of interest in ML applications. These applications are made up of distinct "layers" with differing characteristics and resource requirements. ML applications are also often highly computationally demanding. Given this it is should come as no surprise that there is great interest in combining ML with heterogeneous accelerators.

We wish to look at general question of how best to map a heterogeneous application to heterogeneous hardware for the specific case of a ML application for some desired metrics.

5

Design

5.1 Hardware

For testing the primary computer used was the Chalmers ORIN. It is equipped with an integrated Jetson ORIN GPU and a ARM Cortex-A78AE. The ORIN chip posses both the standard CUDA cores found on NVIDIA GPU but also Tensor Cores, compute units specialized for half precision matrix multiplication. The code was developed on our personal computers. In addition, some testing was also conducted on our personal machines to get a wider spread of data. Table 5.1 shows the exact hardware on each machine.

Computer	OS	CPU	GPU
Chalmers	Linux	ARM Cortex-A78AE v8.2	Integrated Orin Ampere GPU
Max'	Windows 10	AMD Ryzen 7 5800X	NVIDIA GeForce RTX 3060 TI
Lukas'	Windows 10	AMD Ryzen 7 3700X	NVIDIA GeForce RTX 2070

Table 5.1: Hardware details for the target architecture (Chalmers) and personal machines used for testing.

5.2 Software

The software was coded in C++ while making heavy use of OpenMP and CUDA for development of layer code targeted towards the CPU and GPU respectively. For measuring performance a variety of methods were used. Wall clock time was measured with the omp_get_wtime() method, additionally CUDA supports the recording of events with cudaEventRecord, this was used to measure how long time execution took on chip while wtime allowed the measurement of the computation time plus the overhead added by memory transfer to and from the GPU. Furthermore the C library Prof allowed us more in-depth performance metrics, counting executed cycles, cache accesses and the like. By combining these measures we could evaluate the performance of the CPU and GPU kernels. For efficiency we made use of a tool called TegraPower that checked power consumption of the CPU and/or GPU at user defined intervals. By averaging its findings we received an estimate for the

power consumption across the whole program execution. A more accurate reading could have been conducted with direct access to the hardware but as we had to use the ORIN computer remotely this was deemed infeasible.

5.2.1 Layers

It was of paramount importance that the implementations being run on the CPU and GPU were as similar as possible in order to ensure the results were meaningful. This meant we could not necessarily run the most efficient implementation of any one layer on either the GPU or CPU if they did not have a clear counterpart on the other hardware, we were interested in testing the fundamental differences in characteristics of the platforms, not which one had had the most effort put into the developments of efficient algorithms. As a result our implementations of the layers are very simple, each one being an almost one-to-one translation of the basic concepts discussed in Chapter **??**. That way the only performance differences would stem from the hardware differences and how they handle parallelism. It is worth noting though that in practical circumstances the results would differ as workplaces would use differing implementations of these algorithms.

6
Method

6.1 Languages and Frameworks

Development was done primarily in C++, Python and CUDA to write kernels for the different ML layers to use. In order to best use the heterogeneous resources it was crucial to have multiple different implementations of the kernels that would be best suited for different hardware. Hence each kernel would have a CPU and GPU implementation even if one of the two would work best on one or the other. This was done since in real world scenarios it might not always be possible to run software on the most fitting hardware, say for example if it was occupied by a higher priority task that would not finish before a looming deadline.

6.2 ORIN

The primary testing platform was a Chalmers computer featuring a Jetson ORIN module. It was chosen due to the heterogeneity provided by the ORIN being a good candidate for this kind of work and it having powerful performance. Software was developed on other computers and then transferred to the ORIN in order to test it.

As part of the tests on the ORIN cset, a Python application, was used. Cset allows for programs and tasks to be run in isolated environments, shielding them from the OS interfering by scheduling background tasks to run on the same cores executing our applications. Cset additionally allowed for specific cores to be used when running software, granting further control over the used resources. Finally commands were also used to restrict CPU frequency, in order to avoid the Linux governors interfering with tests by modifying the frequency.

6.3 Scripts

In order to ensure tests were as reproducible as possible, heavy use of shell scripts was made. The scripts would specify what programs were to be run, what resources were to be used and set up relevant environmental variables. In particular the scripts made heavy use of cset to precisely control resource use, by allowing us to directly specify what frequencies the cpu and gpu should operate at as well as which cpu

cores should be used. In addition the scripts also defined how many threads OMP should use.

6.4 Building a CNN model

When kernels implementations on CPU and GPU have been created for CNN there are various architectures of interest. The first one to be discussed in this report is the ResNet model which offers high scalability and is frequently applied at a supercomputing scale [24]. Previous work [25] has already shown that the scalability of ResNet-50 on heterogeneous systems offer a potential speedup while maintaining accuracy.

6.4.1 ResNet-11

In order to characterize different types of layers and compare CPU and GPU performance a simple neural network application was needed. As such a simple Residual Neural Network (ResNet) was set up. A ResNet is a neural network where certain layers "skip" ahead and link up with layers further down the chain. This is done in order to avoid issues of accuracy degradation by making better use of the input data [26]

In this work, ResNet-11 is used, where the number indicates the number of layers used in the implementation. Hence our model use 11 layers while the previously mentioned ResNet-50 has 50 layers. It consists of 3 sets of 3 convolutional layers and 1 average pooling layers. After these sets there is a softmax layer that put together the final output. These layers were implemented both in C++ and in CUDA so that the performance characteristics of the network may be established on both the CPU and GPU. Furthermore these implementations were tested with both floating point numbers and doubles to best characterize the applications.

6.4.2 MobileNet

In order to gain useful results it was important to be able to test out multiple different types of tasks. Thus in addition to the ResNet we also opted to include a MobileNet. Like ResNet MobileNet is also a convolutional neural network but it's implemented in a very different way. MobileNet makes extensive use of depthwise and pointwise convolution to significantly cut down on the amount of operations it needs to perform compared to a standard CNN. It ius structured as follows: a normal convolution layer that takes in a 224x224x3 input followed by 26 alternating depthwise and pointwise convolution layers. Each of the convolution layers is followed by a batch normalization and then a relu before the next convolution begins. After the 27 convolution layers comes an average pooling layer followed by a fully connected layer and then ending with a softmax. Throughout this entire process the input is compressed from 224x224x3 into 1x1x1000. We were particularly interested in MobileNet due to the lower amount of amount of operations but also because of its use of Batch Normalization.

6.4.3 Batch Normalization

Batch normalization is an operation where each element in a matrix is subjected to a subtraction of the mean for its channel followed by a division by the standard of deviation (also for its channel) before then being multiplied by a arbitrary number and then getting a arbitrary addition. The point of all this is to make the training of neural networks faster by rescaling and recentering every input element, preventing the formation of extreme values. Implementing this on the CPU was easy but it posed a challenge to do on the GPU. We need to know the global mean and standard of deviation for each element but CUDA code is not meant to use global synchronization so either each thread would have to calculate this separately (creating a truly immense overhead) or a different solution is required. What we settled on was Creating three kernels, one that performs reduction across channels, one that calculates part of the standard of deviation and one that performs the actual batch normalization. Then by repeatedly launching the first 2 kernels and using the launches as a global synchronization we could complete batch normalization without too much of an overhead.

6.5 Manual Resource Allocation

After we had collected a sufficient amount of characterization data it was time to experiment with finding "optimal" configurations of hardware, frequency and thread count with regards to either performance or energy consumption. To do this we gathered all our characteristic data for ResNet and MobileNet into two files, wrote a python script that could identify the fastest and most efficient options for each task (tasks are either a layer or an operation such as batch norm or relu). With this data in hand we could identify hypothetical optimal configurations. We also used the data to construct lists showing how much extra time the most efficient option incurred and how much inefficiency the fastest option incurred. With the data in hand we then constructed bash scripts that would manually run each task with its optimal hardware, frequency and (in case of CPUs) thread count. In order to make as fair a test as possible we also made sure to initialize the CUDA memory system before running each task so as not to give GPUs an undue penalty (the memory initialization could affect their execution time by up to a factor of 5). We opted to run each task separately as we could not control GPU or CPU frequency from inside our application and had to do that via bash scripting. These optimal configurations were then contrasted with static all-CPU and all-GPU configurations were the Linux governors were able to vary the frequency as they wished. To make the results as comparable as possible these static configurations were still executed task by task via scripts on the same code as the manually allocated configurations ran on.

6.6 Metrics

When something is to be measured, there is a great importance in understanding what is being measured and how it is of relevance. In the scope of this project the main focus points will be metrics in the domain of performance and energy for a computer system. For a computer there is no elementary single metric to characterize the performance as it can be argued that performance is multi-dimensional [27], and to accurately determine performance we need to investigate multiple values. Therefore variables such as number of instructions, operations, last-level cache misses and similar will also be measured using performance counters. These values will assist in devising a heuristic or determining why performance can be gained or lost.

6.6.1 Performance

A fundamental metric for any computer system's performance is the execution time [28], which is the time required to execute a given task. A computer A with a lower execution time than computer B would with this interpretation therefore be considered to have a better performance. An important note is that execution time is nondeterministic which means that to have a more accurate value, the execution time should be measured multiple times and at least report the mean [28].

The formula for the total execution time of a task can be defined with the instruction count(IC), cycles per instruction(CPI), and cycle time(Tc):

$$T_{\mathrm{exe}} = IC * CPI * Tc$$

Speedup can be an abstract definition and to avoid uncertainty it will be defined as the following:

$$S_{\mathrm{i}} = \frac{T_{\mathrm{R,\ i}}}{T_{\mathrm{A,\ i}}}$$

where S_i is the speedup of the time for a process i a machine A has over machine B. To further receive a more accurate result when comparing speedups between different machines and programs there are various ways of calculating the mean. Three common approaches are the arithmetic, geometric and harmonic mean which are defined as the following respectively[27]:

$$Arithmetic, \sum \frac{T_{\mathrm{n,\ i}}}{N}$$

$$Geometric, \sqrt[N]{\prod T_{\mathrm{n,\ i}}}$$

$$Harmonic, \frac{N}{\sum \frac{1}{n}}$$

6.6.2 Throughput

In addition to execution time there are other metrics used for performance analysis and one commonly used is throughput [28] which is defined as number of tasks per time unit. This can be useful when investigating areas such as bandwidth where it is of interest to compare how many bits are transmitted per second. In the scope of this project throughput will be discussed when it comes to tensorcores as they offer a higher throughput[29]. The throughput becomes an interesting metric when memory is an important factor of determining performance.

6.6.3 Energy

A big part of this project will focus on the performance but energy is also a highly relevant metric which will be investigated. Energy can be characterized in different ways and some examples are to measure it in joules, the power (joules / second), the energy combined with the time it takes as a product (Energy-Delay product, EDP) and joule + quality of service (QoS). The main focus will be on energy in joules but the discussion will include input on how the other metrics are relevant when developing a heuristic.

Power has two main components of interest, the static and the dynamic power. The dynamic power has become the main determinant for power consumption and can be represented using this formula [30]:

$$P_{\text{dynamic}} = \alpha C V^2 f$$

As explained in [31], the dynamic power varies similar to the cube of the maximum frequency, which is why it becomes a major interest to investigate. To give an illustration, if a system has a frequency f and a speed S, one could either (1) multiply the system four times or (2) increase the frequency fourfold. This would in both cases yield a 4x S but the dynamic power in case (2) would be 16x as large as the dynamic power in case (1).

6.7 Monitoring and measuring

When there is a need to evaluate a program there are various ways of measuring performance. A program made in C++ could measure the time of a given task to execute from start to finish with *std::clock()*. In the scenario of this project there are more metrics which are to be taken in to account. Therefore there is a need to further explore for solutions to measure the given metrics.

6.7.1 Performance Counters

This project is aimed for a Linux system which makes Performance Counters for Linux of interest. The Red Hat organization has an article describing how *perf* is a user-space tool which can be used to analyze collected performance data [32]. As per

the perf manual page it is confirmed that the tool can analyze executed instructions, cache-misses, CPU cycles and other data points of interest. [33]. While perf is an attractive tool for measuring, it is run in the Command-Line Interface (CLI) and targets the entire application. In the case of this project it is of relevance to understand how each part of the program runs. By isolating a part of the program and running it, the specific part can be analyzed in greater detail. For example, in a ML application each layer could be analyzed to understand how long each layer is run and perhaps where the bottleneck is. To invoke such calls in C or C++ code there is a manual page [34] describing how *perf_event_open()* works.

6.7.2 CPU Shielding

In multi-processor realtime systems (RTS), there could be a need to isolate specific CPU cores. This is due to the fact that when running a process on a core the CPU could halt or interrupt said process in order to prioritize another [35]. In the case of this project, there is another reason. When running experiments on the cores and working with very small numbers for time measurements, each interruption or swaps with another process can have a big impact on the results. Introducing CPU shielding will allow for the processes to run isolated and therefore guarantee a rapid response while providing a deterministic environment, thus coming closer to an actual real time scenario unimpacted by other processes [36].

7

Results

In this chapter the results from running our tests will be presented. Additionally we present some of our insights (more in chapter 8) that we gained from the data.

7.1 ResNet Layers Characteristics

Going into the tests we expected the GPU to have a clear advantage over the CPU when it came to performing convolution as it consists of matrix multiplication and was expected to be highly parallelizable. There was more uncertainty on average pooling and softmax since different setups could yield a better performance per joule, while it offered parallelism it was at a lower degree compared to the convolution.

In general the testing confirmed our suspicions, the GPU had a decisive advantage for all convolution layers (if the memory initialization delay was factored in, see GPU Kernel Implementation) and it also outperformed the CPU for average pooling, though to a lesser extent. Softmax was a lot more competitive however, while on average the GPU had a better performance across all tests it noticeably performed much worse if forced to use double point precision and there were some occasions where the CPU outperformed it. In the setup where double precision was used and both hardware components ran at maximum frequency we could see the most significant difference. This can be explained by the fact that the GPU become memory bound and also by the fact of how many FLOPS per clock cycle per core the hardware could do.

Comparing the ampere architecture which supposedly can only do 1/32 of a double-precision per clock cycle and core the ARM Cortex-A78 (see table 5.1) series which can do 8 per core per cycle there is a large difference in how many FLOPS they can do per second [37]. With 12 cores and each core doing 8 per cycle we would have $12 * 8 = 96$ 64-bit Floating-point operations per clock cycle (FLOPC), while for the GPU we would have $128 * 16 * 1/32 = 64$ 64-bit FLOPC. This is without taking memory bandwidth for GPU, time for a cycle (frequency) or overhead for using multiple cores into account but should give an idea on why CPU performs double-precision with higher efficiency if maximum parallelism is used for both hardware components.

Even when comparing the two from the perspective of energy consumption the GPU still came out ahead, it might have had a higher power requirement but its

superior execution time more than made up for it. In general ResNet would seem like a poor fit for heterogeneous hardware as it is nearly always better to run it on just the GPU. The only exception is for softmax, which for sufficiently large inputs and double precision works better on the CPU. However give that MobileNet also includes a Softmax layer and is far less homogeneous, as well as our limited time and resources, we opted to forego creating any manual configurations for ResNet and focus our testing on MobileNet.

7.2 MobileNet

We had already done extensive characterization of the ResNet by the time we worked on the MobileNet so we already had some hard data to base our assumptions on. We expected the GPU would still prove superior for convolution but we were more unsure on the added batch normalization as it needed global synchronization. In order to assess the potential of heterogeneity we made two different task mappings. The first used the most efficient option for each task and the second the highest performance option for each task. We called these configurations OptEnergy and OptTime respectively and were compared against each other. Additionally we also recorded performance data for using just a CPU and just a GPU for every task and letting their frequency vary, we called these configurations VarCPU and VarGPU respectively. The bulk of the work of the MobileNet consisted of multiple depthwise and pointwise convolutions, batch normalization and ReLu. Finally it should be noted we executed each task five times and averaged the results to minimize the impact of individual variations in execution. We will begin our analysis by looking at the results from individual task executions before looking at the combined results of the configurations across the entire network.

7.2.1 Pointwise Convolution

In our MobileNet there were 13 pointwise convolution layers. Figure 7.1 showcases the performance characteristics of each configuration for each of the convolutions. The x axis indicates which convolution is being done and the y axis shows the execution time for the convolution (in ms). All configurations performed as we expected with OptTime executing faster than all other, OptEnergy only trailing it slightly (OptEnergy in most cases used the second or third fastest possible frequency) and VarCPU and VarGPU trailing behind by a significant margin. Our previous testing had indicated that a GPU would always work better than on a CPU for this kind of task and we expected that the varying frequency would inflict a performance penalty as it took some time to find the optimal setup. We also note that for the 9th convolution OptEnergy outperformed OptTime by the slightest margin but this was a case where our tests had indicated the same frequency was ideal for both performance and efficiency so this is nothing more than individual variations between task executions. Next we turn our attention to Figure 7.2 which shows the corresponding energy consumption for each of the above convolutions. Again the results conformed to our expectations with OptEnergy being more efficient than OptTime and both outperforming VarCPU and VarGPU by a significant margin.

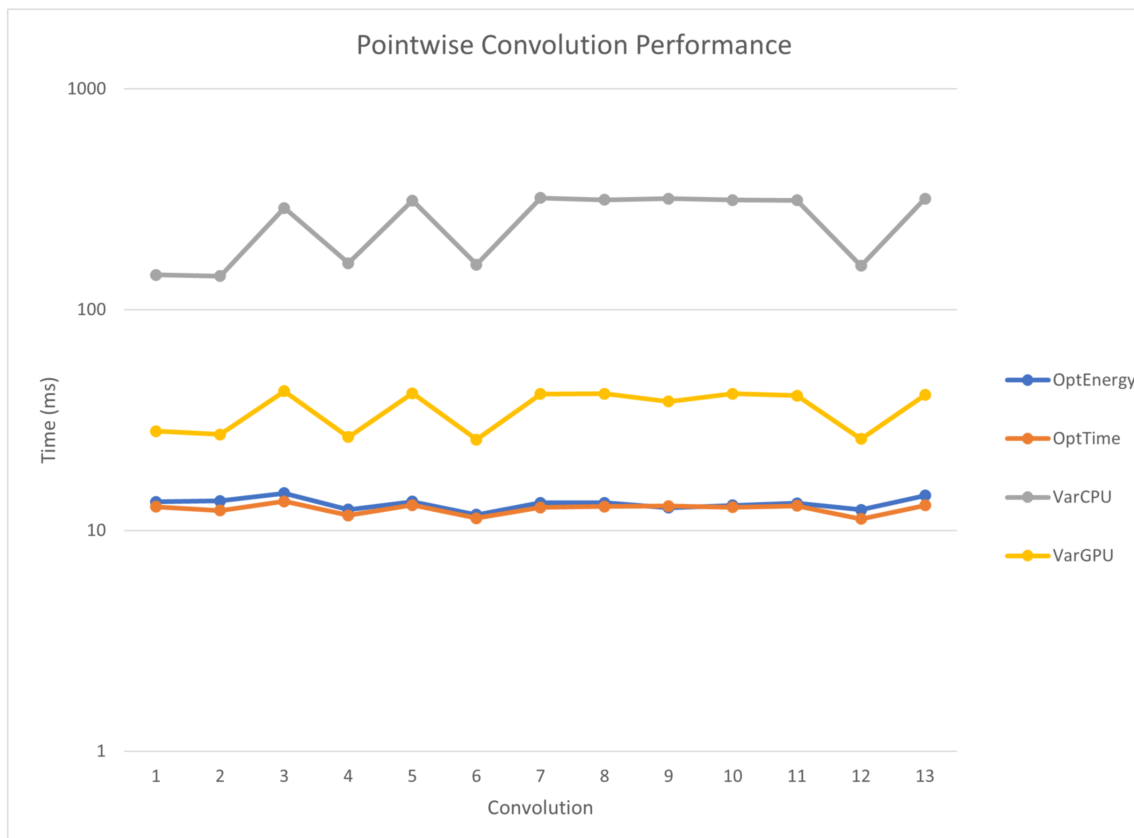

Figure 7.1: Line graph illustrating the performance of each pointwise convolution

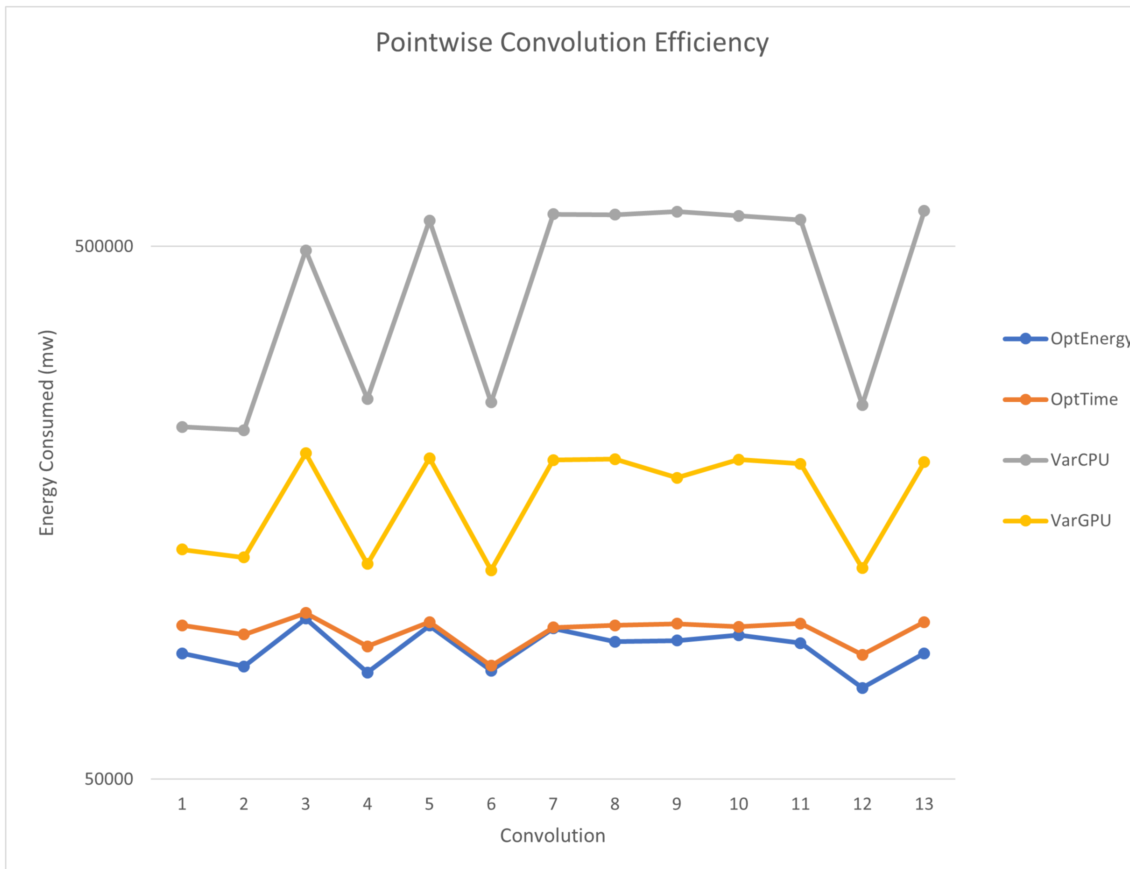

Figure 7.2: Line graph illustrating the efficiency of each pointwise convolution

The only other noteworthy thing is that the CPU had consistently less power usage than the GPU but consumed vastly more energy due to its much slower execution time.

7.2.2 Depthwise Convolution

We also had 13 depthwise convolutions in our MobileNet, as depthwise and pointwise convolutions together approximate the results of normal convolution. Moving onto depthwise convolution we expected the findings from pointwise convolution to hold with our own configurations outperforming the varying ones at every point. However by looking at Figure 7.3 it was clear this was note the case. The graph shows the performance of each convolution and we see that in the majority of cases we were outperformed by both VarCPU and VarGPU. When we performed the tests to select our manual configurations we had assumed that fixed frequencies would always outperform varying frequencies and so did not test them. Clearly there was some other factor that meant our chosen frequencies were suboptimal. Figure 7.4 shows the energy consumption for each task and the same story plays out again. While we were more efficient than VarGPU in all cases VarCPU dominated all convolutions except the last. Further research into this type of convolution is needed in order to find optimal configurations for it.

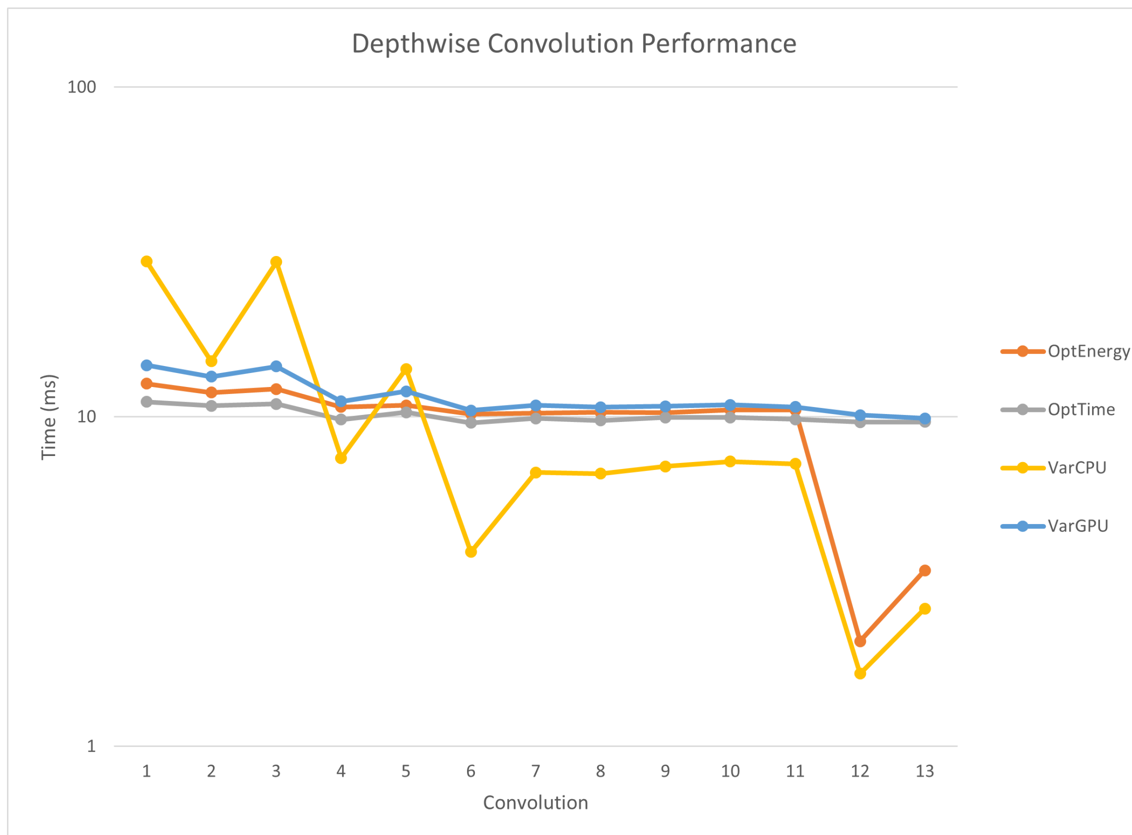

Figure 7.3: Line graph illustrating the performance of each depthwise convolution

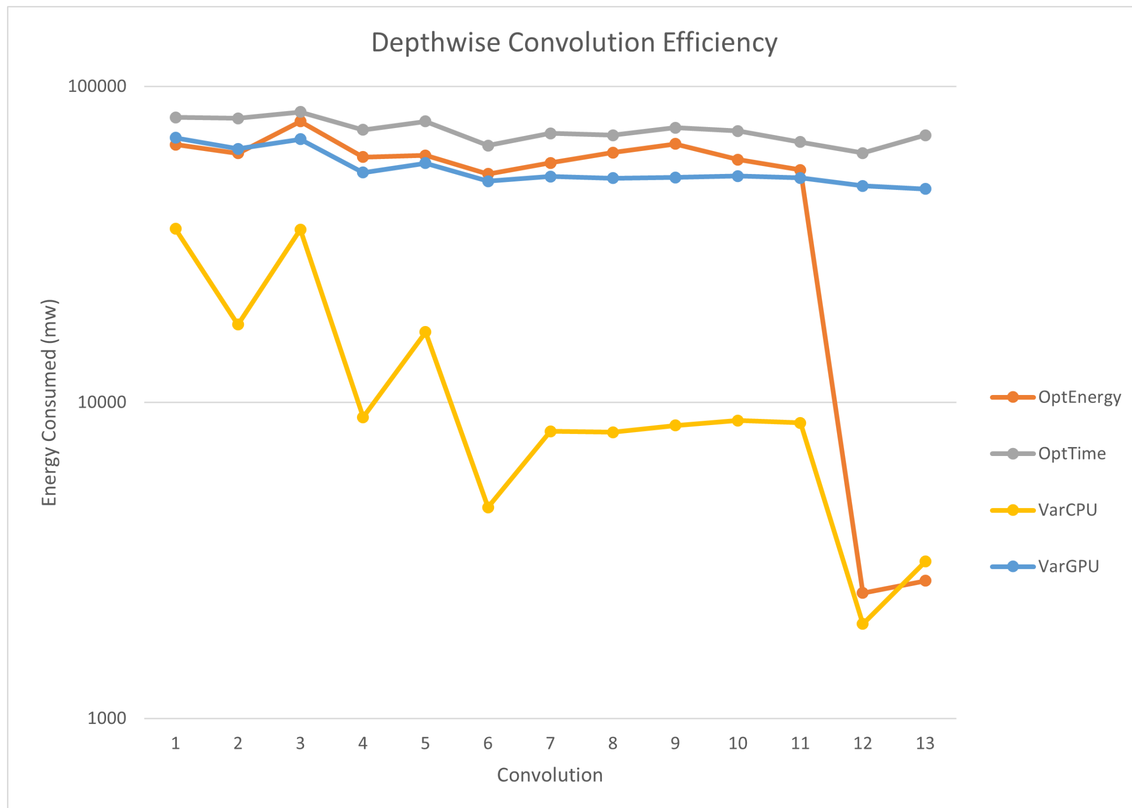

Figure 7.4: Line graph illustrating the efficiency of each depthwise convolution

7.2.3 Batch Normalization

After every convolution (including the normal one at the beginning of the network) a batch normalization task was performed. Figure 7.5 shows the performance of each batch normalization task expressed in milliseconds. While in general our configurations performed about as we had expected from the pointwise convolution tests we do see a case (layer 3) where VarGPU outperformed OptEnergy, though this is not concerning since performance is not the goal of OptEnergy. It is however one of the rare scenarios where OptTime and OptEnergy ran on different hardware, the former using a CPU and the later a GPU. Batch normalization is one of the few cases where the CPU consistently outperformed the GPU (alongside depthwise convolution). This is likely due to the overhead added by the need for global synchronization needed to perform the task, our GPU implementation only being able to do so by launching a kernel multiple times. Moving on Figure 7.6 showcases the energy consumption of each layers batch normalization. Here we have another mixed bag of results, in most cases OptTime was more efficient than OptEnergy but was itself less efficient than VarCPU in several cases. We had no methods of directly measuring energy consumption during the tests and instead had to approximate it so it seems to be that either our previous tests where we selected the consumption failed to account for some factor that significantly affected efficiency or the measurements from these tests were tainted somehow or perhaps a combination of the two. Either case we see CPU based configurations dominating the GPU which seems entirely logical as the CPU in general has less power consumption and was in fact faster

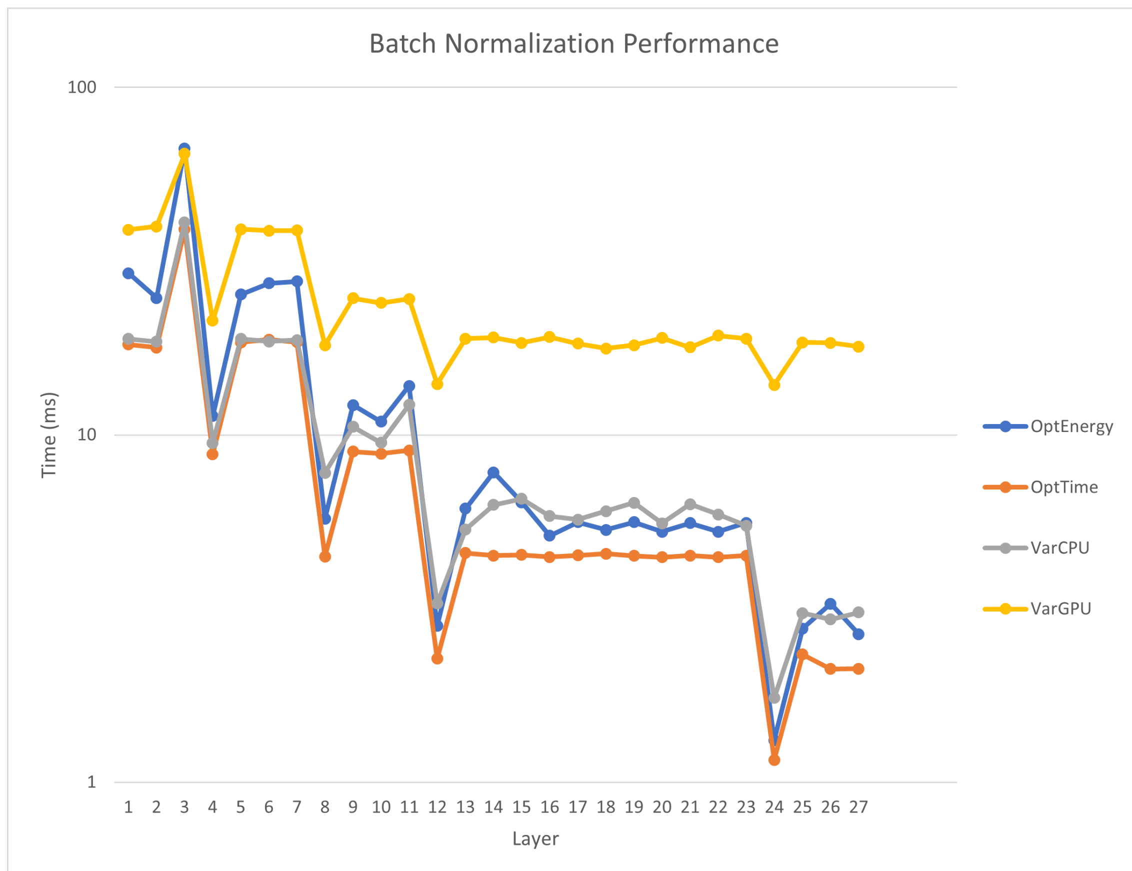

Figure 7.5: Line graph illustrating the performance of each batch normalization

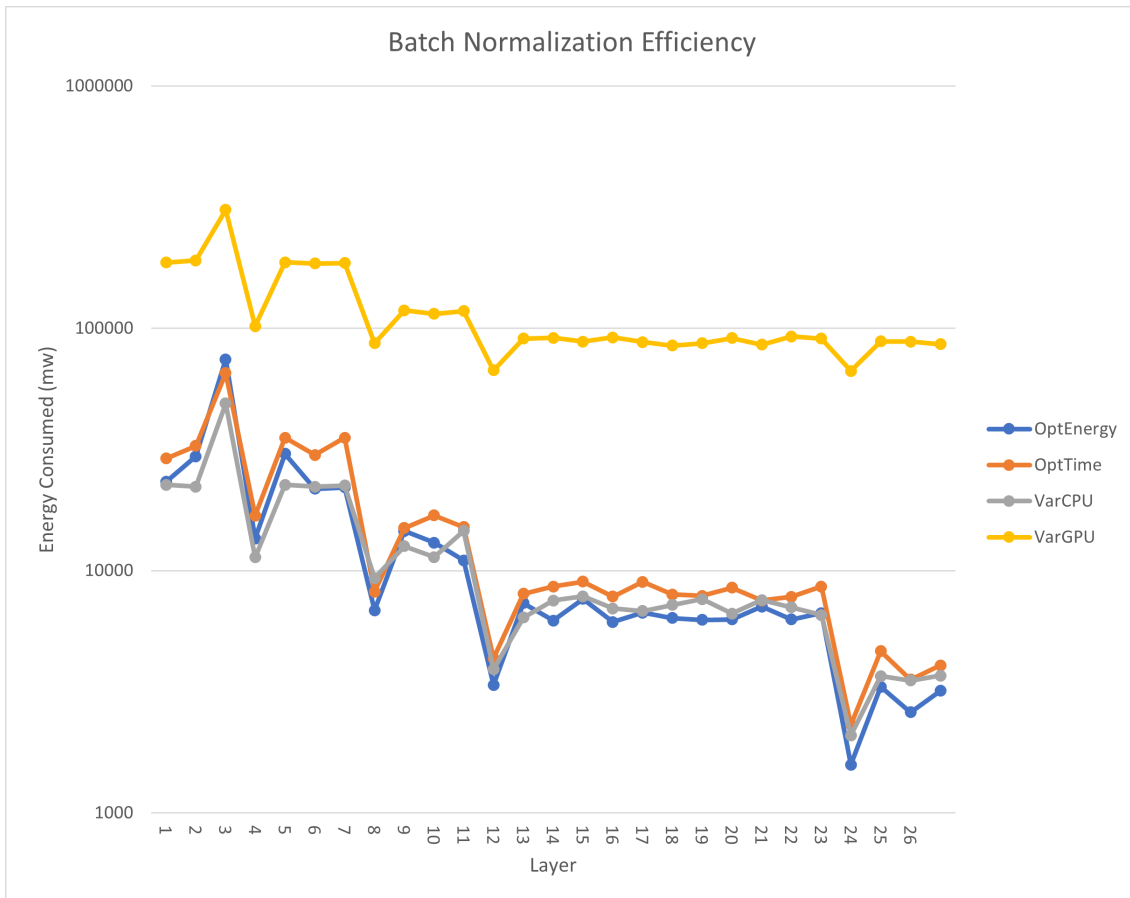

Figure 7.6: Line graph illustrating the efficiency of each batch normalization

than the GPU for these tasks.

7.2.4 ReLu

Next we turn to ReLu, arguably the simplest of all the tasks in the network. One ReLu task was performed after every batch normalization. Figure 7.7 showcases the performance of each ReLu operation in our network, with each occuring after a batch normalization. This is another mixed result as at every later OptTime is outperformed by both OptEnergy and VarCPU with the exception of layer 24. It seems that our tests wrongly allocated it to the GPU in all cases except that one. Given the computational simplicity of ReLu we believe the overhead added by launching a GPU kernel made them non competitive with the CPU based options. Looking at the efficiency graph in Figure 7.8 we see similarly mixed results. While OptEnergy is the most efficient option in the majority of cases it is outdone by VarCPU at a few occasions. More research is needed to get consistently better results but the overall trend followed our expectations in this regard at least.

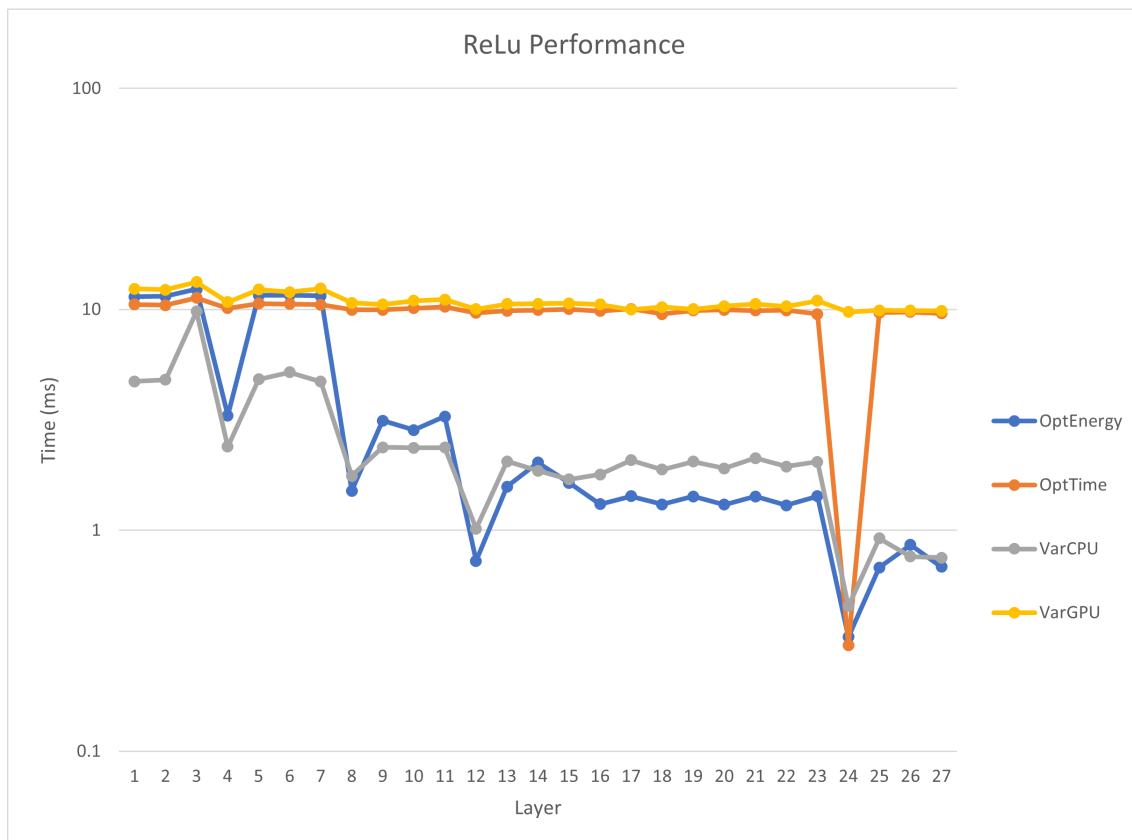

Figure 7.7: Line graph illustrating the performance of each batch normalization

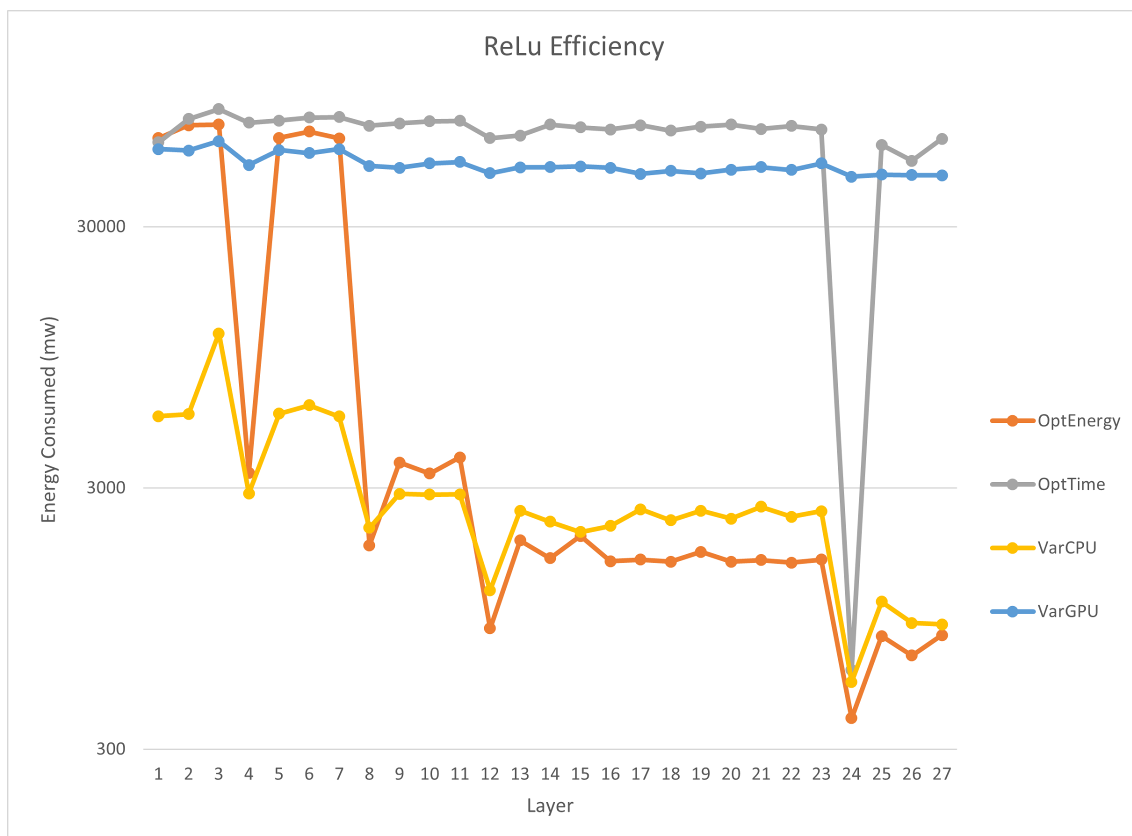

Figure 7.8: Line graph illustrating the efficiency of each batch normalization

7.2.5 Remaining Tasks

In addition to the above types of tasks our MobileNet also had one normal convolution (at the very beginning) and an average pooling, fully connected and softmax layers (at the very end). Unlike the above tasks there was only a single instance of each of these so we will deal with them as a group. Table 7.1 showcases the performance of these tasks expressed in milliseconds. We see that in all cases except for the fully connected layer OptTime outperformed all other other configurations. Meanwhile table 7.2 showcases the energy consumption of each configuration expressed in milliwatts. While most results are as expected the standout exception is the initial convolution where OptEnergy was by far the worst option in regards to both performance and efficiency. Our selection criteria seem to be wholly unsuited to normal convolution and additional refinement is necessary.

Task	OptEnergy	OptTime	VarCPU	VarGPU
Convolution	171.825	11.19624	115.97739999999999	14.34776
Average Pooling	0.5765013999999999	0.3260362	0.4275536000000001	9.816714000000001
Fully Connected	6.274162	10.914460000000002	4.847085999999999	21.25232
Softmax	0.06956166	0.041247980000000004	0.054503974999999996	9.690654

Table 7.1: Table showing the performance for the remaining tasks in the MobileNet (measured in ms)

Task	OptEnergy	OptTime	VarCPU	VarGPU
Convolution	205502.69999999998	67009.4964	184891.17108	68697.07488
Average Pooling	459.47161579999994	650.1161828	511.3541056000001	47002.426632
Fully Connected	5000.507114	78387.65172000001	5797.114855999999	101747.60723200001
Softmax	55.440643019999996	82.240222524	65.1867541	46398.851352000005

Table 7.2: Table showing the energy consumption for the remaining tasks in the MobileNet (measured in mw)

7.2.6 Overall Assessment

Next we look at the MobileNet as a whole. While our configurations did not exceed in their role for every single task the overall trend seemed to be in their favor. When looking at the network as a whole we introduce two additional configurations, OptEnergyVar and OptTimeVar. These are not real configurations we ran tests with but instead the results of taking the best hardware from VarCPU and VarGPU for every tasks with regards to either energy consumption or execution time respectively. This is done in order to illustrate how much benefit is gained simply from running on hardware best suited to a particular task without pre defined resource allocation. We did not believe a comparison of only OptEnergy, OptTime, VarCPU and VarGPU would be fair as the later two would be severely penalized at task they are not ideal for.

Figure 7.9 shows the overall performance of each configuration. As expected OptTime and OptEnergy both vastly outperformed VarCPU and VarGPU but the comparison with OptEnergyVar and OptTimeVar is significantly more interesting. While both are outperformed by OptTime and OptEnergy the difference is much smaller.

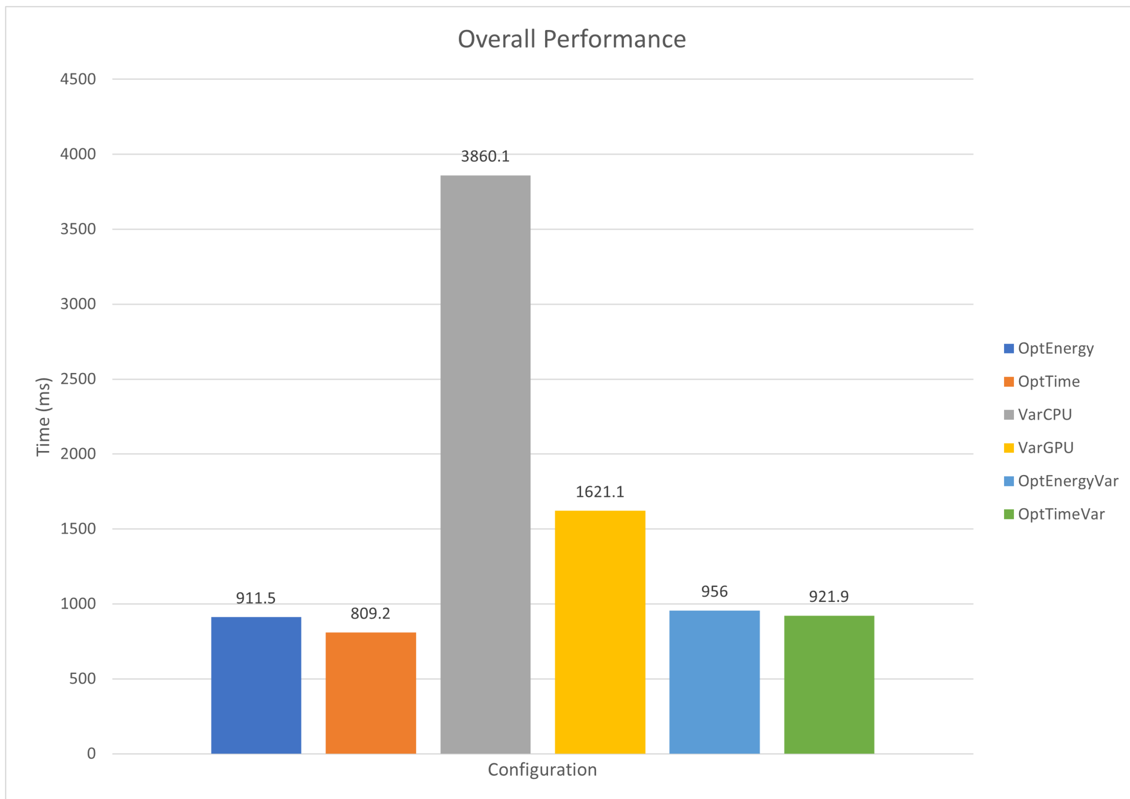

Figure 7.9: Bar graph illustrating the overall performance of the network

The biggest difference is OptTime having 85% the execution time of OptEnergyVar and the gap being smaller in all other cases. This would suggest that unless performance is at an absolute premium it might be best to identify which hardware a particular piece of software works best on and then not worry about resource allocation. Moving on Figure 7.6 illustrates the energy consumption of each configuration. Again we see a significant gap between OptEnergy and OptTime compared to VarCPU and VarGPU. And again OptEnergyVar and OptTimeVar only lag slightly behind, with the gap being even closer this time. Additionally we see that OptTime consumed significantly more energy than OptEnergy var for a not at all proportional increase in performance (per Figure 7.9). As OptTimeVar does not have such an outsized loss in efficiency this would suggest that while the resource allocation is increasing performance it does so at significantly diminishing returns. Again we see that resource allocation confers minor benefits compared to those from hardware heterogeneity to begin with.

7.3 Layer Trends

In general the more "independent" the calculations on a layer were the better the GPU performed. By this we mean the less information any one calculation needs to know about the data set the better it will be on the GPU. For example the GPU dominated the CPU in all cases when it came to convolution and convolution requires no global information, it is just a matrix multiplication with the weight and

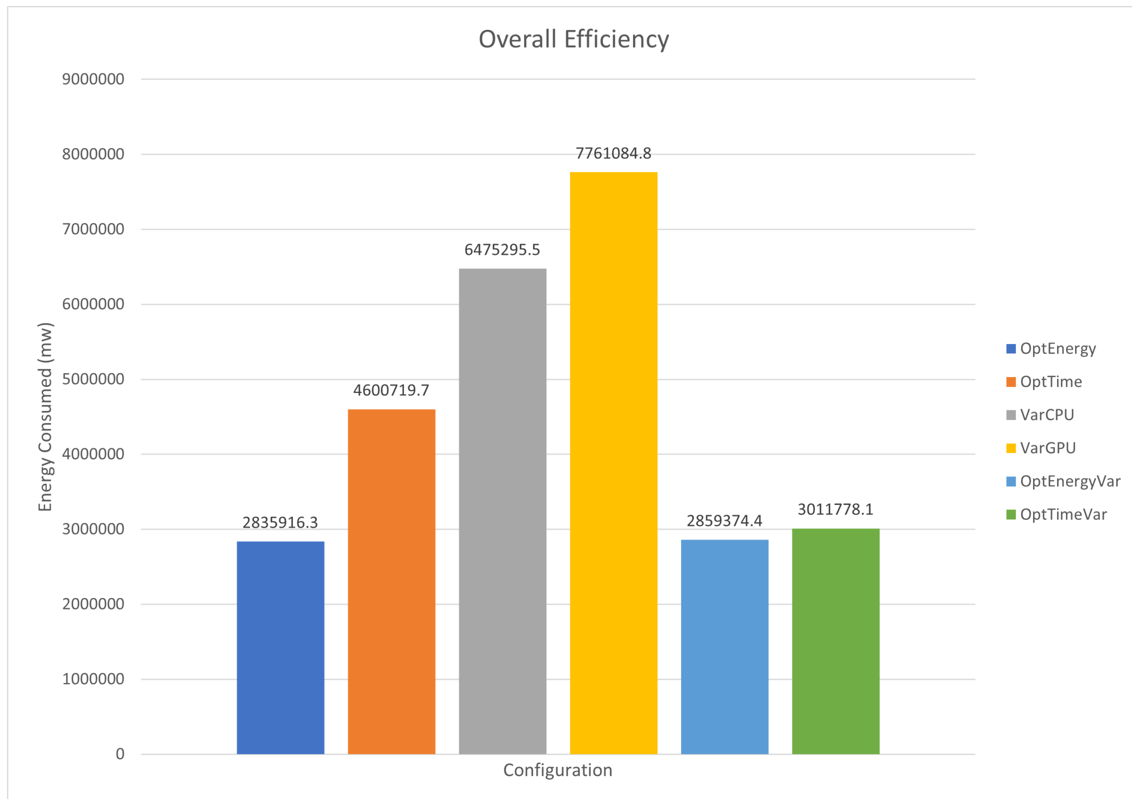

Figure 7.10: Bar graph illustrating the overall efficiency of the network

a small portion of the input matrix. Meanwhile the GPU performed much worse on softmax which requires knowledge of the whole row being operated on and was inferior to the CPU for batch normalization which requires the global mean and standard of deviation.

7.4 Impact of Frequency

Although the highest frequency is expected to operate the fastest, nearly always it was not the most energy efficient. Often the second or third highest option would give the best trade off between performance and power consumption. This could be explained by multiple factors where arithmetic intensity, bandwidth and the total power cost are some of them. Is it noteworthy to mention that the results seen in chapter 3.3 of running a GEMM on varying frequencies became similar to a negative $log(x)$, where larger frequencies did not have a proportional larger advantage relative to the power consumption.

If the total power is defined as $P_{\text{total}} = P_{\text{static}} + P_{\text{dynamic}}$ and the power consumption ranged between $[750, 2500]$ watt for CPU and $[4500, > 10000]$ watt for GPU, the static component can be considered to be in close proximity to the lower limit. As a consequence of this, the GPU needs to be 4x as fast, running at the lowest frequency in order to compete with the CPU in energy-efficiency. Apparent from table 7.3, there are indeed layers where the combined time for the GPU is 4x faster than the

CPU. It is also quite clear that the memory bandwidth is the largest contributor to execution time for the GPU, hence why AI is an important metric as it indicates when one component could be advantageous over another.

Layer	CPU time	GPU time (exec time + memory bandwidth time)
Initial convolution:	424.781	6.32659 + 104.433
Average Pooling:	79.3912	3.0648 + 9.71631
Dense:	89.3684	3352.56 + 3411.4

Table 7.3: Different layers and their corresponding execution time on CPU and GPU for a N = 1024 ResNet. CPU frequency is 2.2 GHz and GPU frequency is 1.3 GHz.

7.5 The impact of tensor cores

To compare the tensor- with cuda-cores tests were run on a General Matrix Multiplication (GEMM) with sizes differing between 16x16 and 1024x1024 matrices. Key values of interest are the time, power and energy (P*T). The results can be seen in figure 7.11.

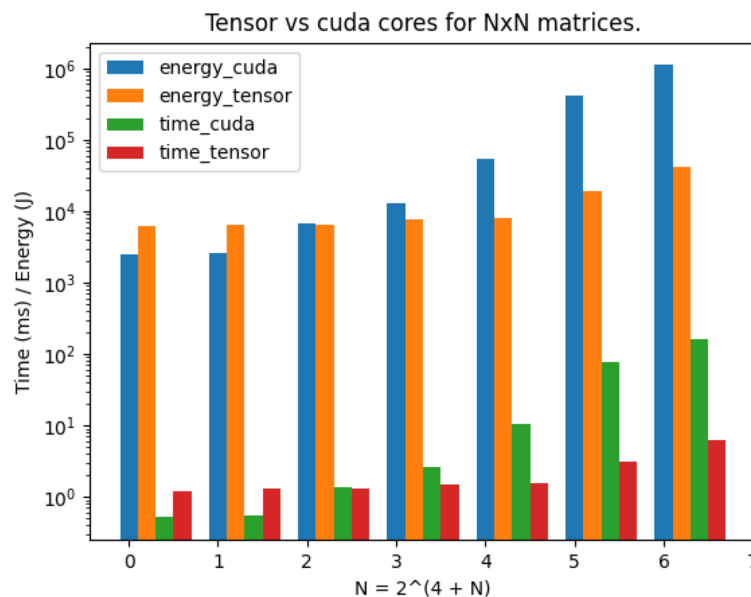

Figure 7.11: Time and energy of tensor cores compared to cuda cores.

There is a clear advantage of running CUDA cores for lower matrix-sizes as the tensor cores have both higher energy and execution time. When N grows larger the trend seems to indicate that tensor cores gain an advantage since the increase in time and power is lower relative to the one for cuda cores.

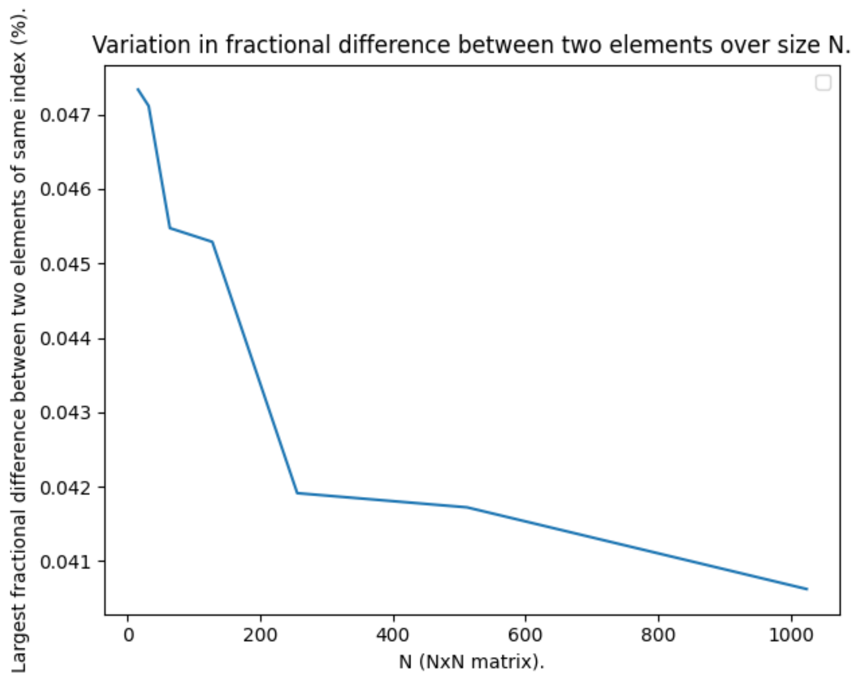

Figure 7.12: Fractional difference (y-axis) for running GEMM with tensor cores and comparing with CPU implementation for validation over varying matrix size (x-axis).

For margin of error it should be noted that although the difference is not very large, there is an impact of mixed-precision or half-precision computation. In our case the largest fractional difference (LFD) between GEMM on cuda and tensor cores can be seen in figure 7.12. The values range are within the bounds of: $[0.047\%, 0.041\%]$. Doing one GEMM with a LFD of 0.047% can be argued to not have a large impact on the result. Although it is worth mentioning that if it is assumed that the LFD is maintained for each GEMM, meaning that each GEMM has a maximum loss of 0.047%, the final impact can be rather large. If ResNet-11 performs 11 GEMM operations and MobileNet performs 27, this could result in a maximum loss of $(1 - 0.00047)^{11} \approx 0.9948$ for ResNet and $(1 - 0.00047)^{27} \approx 0.987$ for MobileNet.

8
Discussion

8.1 Frequency scaling

The trend when changing frequencies is that at a certain a point the higher frequencies did not yield as much of a significant speedup while having a higher energy cost. Hence why the fastest frequency was not always the most optimal for energy-efficiency. The idea is that if a hardware component can be 2x faster for 2x the power the $\Delta energy$ would be zero. Instead as we can see in the GPU could in many cases have a higher proportional speedup compared to the increased energy cost. This brings the question, why is the total energy still larger? One explanation would be that if the program is memory-bound, if the AI is not large enough, we would not benefit from a peak performance increase since we will not have any use of it. The energy-costs still go up although we cannot benefit from the increased performance which is why the total energy might not go down. Similar to a very fast car being stuck in traffic, even if it can go fast it is limited by other factors and might be more expensive to drive.

8.2 Implementation of kernels on CPU and GPU

As noted we performed tests using our custom implementations of the GPU and CPU code. This was done in order to make sure both hardware were performing the exact same work and that the only differences in characteristics would stem from the underlying hardware. But of course this is not how modern kernels are used. They are much more closely tailored to their hardware using various tricks and shortcuts to speed up calculations by for example exploiting CUDA memory architecture. In addition we are by no means C or CUDA experts and better implementations may exist. Both of these should be factored into future heuristics and frameworks. Perhaps a hardware known to have a superior but non-comparable implementation of a kernel could be weighed more heavily to factor in the implementation advantage in addition to the advantage stemming from its heterogeneous characteristics.

8.3 A homogeneous execution

The first CNN implementation was based on the ResNet-11 [26] model and was adapted to the scope of this project. A major thing about ResNet is the number of

convolutional layers executed. Considering the great possibility of parallelization of such a layer the GPU would always outperform the CPU which led to a *homogeneous execution*. Although there were still performance and energy improvements with frequency scaling these results were different from what was originally expected. With GPUs being specialized in floating-point operations it was expected that the GPU would have an advantage for convolutional layers with FP values but other data types such as double-precision (64-bit) were initially thought to be a better competitor as CPUs have more double-precision ALUs.

8.4 Manual Configurations

One issue we faced when testing our theoretically optimal configurations is that we could find no tools to directly control the CPU and GPU frequencies from inside the application. We had to resort to using external tools to set the frequencies before launching the program and execute each layer as a separate application. While we were able to compensate for this to ensure there was minimal overhead it was still not optimal and not close to the "natural" circumstances where you'd want to assign software to hardware on the fly. Thus a hypothetical future framework would either need to be closely integrated with the OS to give it control over the frequencies or have some other way to seamlessly affect them. To a lesser extent this also occurred with the OMP thread counts though it is possible to directly restrict those while executing software, we simply chose not to do it and have it be a part of the script since we were already setting the frequency there anyways.

8.4.1 Varying Frequency

As we saw much of the benefits of our configurations came from them utilizing heterogeneity to begin with, not from their fixed frequencies. When we picked the best option out of the CPU or GPU for every task we saw combined results that closely matched our optimized configuration. Unless efficiency or performance is of paramount importance in many situations it might be much more expedient to establish whether a task works best on the CPU or the GPU and then let the frequency dynamically adjust to their needs. This would also significantly cut down on the need to worry about memory bounded situations as the frequency could naturally adjust to minimize energy consumption during waiting periods. Overall further research is needed into finding optimal frequencies but our findings do suggest fixed frequencies might be better suited for situations where performance is paramount, which makes sense as frequency is one of the most important aspects of execution time.

8.5 Differences Between Computers

Our testing was conducted on the Chalmers ORIN computer and our personal desktops for a sample size of three. We all had different hardware so it was a somewhat diverse spread but it is still far short of what is needed for truly generalizable find-

ings. Further work would need to test on a wider variety of hardware in order to develop a more universal heuristic. In addition it might be worth investigating how identical hardware runs on computers with different OS/task schedulers to more closely monitor the impact they have on layer characteristics. Owing to our limited resources we were not able to make progress on that front.

8.6 Ethics

Although it might be an obvious topic it is relevant to discuss as when AI or ML advances there is a potential job displacement because AI might have the chance to automate tasks which are traditionally performed by humans. For example if the accuracy for autonomous cars will be considered sufficient and fast-enough, jobs such as taxi or truck drivers might be replaced with robots. Therefore it is relevant to ensure a just transition of affected workers and addressing potential socioeconomic impacts.

While a 98% accuracy might not have a large impact on predicting if an image of a number is the correct number, problems arise when solutions with lower accuracy are used in areas where human lives are involved. When autonomous cars became of interest various questions were risen, among them was the famous trolley problem.

In the context of this project those issues are highly relevant as CNNs or other machine learning models are used in autonomous vehicles [38] or in image recognition. For instance it could have some major drawbacks if a car only does 98/100 decisions correctly or if facial recognition in law enforcement mark innocent people as guilty [39]. Therefore it is important to note that perhaps tensor cores should not be used in those domains unless further action is taken to maintain accuracy.

Complexity increases when a heterogeneous system is to be fully utilized which brings up the dilemma of intellectual property and ownership. Many components and technologies are supposed to work together where it might be problematic with ownership and rights for the developed technology or algorithm. It is key to determine fair attribution and appropriate rights for collaboration as we are supposed to engage and improve together but under fair use.

8.7 Sources of error

As always with software research there is the potential for errors to taint the results. Beyond the usual hardware failures that could taint the results we see two potential causes of errors. The first is the way in which we measured energy consumption. As we did not have physical access to the computer the tests were run on we had to rely on software to measure power consumption which may be imprecise. Our consumption metrics are as such best understood as rough estimates and since they are based off of periodic pools of power usage it is possible the reality differed. We find this unlikely since we used repeated runs of the tests to provide an average measurement but it is still worth noting. The other source is the implementation of

the algorithms we tested. While we did our best to make sure the same algorithms were run on both the CPU and GPU there is always the possibility we made some error in the code which would taint the results. If further verification of the results is desired it would likely be a good idea to look at the software on a lower level and analyze the instructions being run on both the CPU and GPU.

9

Conclusion

Based on our tests we conclude that proper utilization of hardware heterogeneity and resource allocation can bring significant benefits when it comes to both efficiency and performance. However the two are not equal, the benefits of adding resource allocation to hardware heterogeneity are less than those gained by using hardware heterogeneity itself. This is important to keep in mind when making decisions about how to improve existing systems with heterogeneity, unless maximum efficiency is needed the effort to identify optimal resource allocation may not be worth it and significant advantage can be gained just from identifying what tasks are best suited for what hardware. Further research into how to identify optimal resource allocation may alleviate this concern.

We also stress that these results will not fully match up to real life circumstances due to our focus on making the GPU and CPU implementations as comparable as possible. In any real life applications we expect CPU and GPU implementations would use device specific optimizations to obtain even more performance and efficiency but we expect the general trend of our findings to hold as we rarely found situations where the CPU and GPU were nearly equal. One almost always dominated the other so we expect device specific optimizations to widen existing gaps rather than narrow them.

While most of our work was spent on comparing cuda cores and traditional CPU cores, there is a great opportunity to implement solutions for tensor cores in order to achieve even greater energy-efficiency and performance for matrix operations. We see that although the mixed-precision applied in tensor cores reduce accuracy which might not be beneficial for all domains, such as areas where high accuracy is critical, the benefit from using tensor cores is highly relevant if high-precision is not a must.

Finally we believe there is significant room for improvement and further refinement of this area. We have done an introductory look at the subject but there is significant room for further refining resource allocation, improving power measurements and broadening the scope to more types of hardware. In particular we think it would be interesting to look at how heuristics could use ours and similar findings to predict what hardware and resources would best suit a particular task and how other types of ML applications could best be characterized.

Bibliography

[1] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Scientific Programming*, vol. 18, no. 1, pp. 1–33, 2010, ISSN: 1058-9244, 1875-919X. DOI: 10.1155/2010/540159. [Online]. Available: http://www.hindawi.com/journals/sp/2010/540159/ (visited on 02/08/2023).

[2] M. W. Azhar, P. Stenström, and V. Papaefstathiou, "SLOOP: QoS-supervised loop execution to reduce energy on heterogeneous architectures," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 4, pp. 1–25, Dec. 31, 2017, ISSN: 1544-3566, 1544-3973. DOI: 10.1145/3148053. [Online]. Available: https://dl.acm.org/doi/10.1145/3148053 (visited on 02/08/2023).

[3] M. W. Azhar, M. Pericàs, and P. Stenström, "Sac: Exploiting execution-time slack to save energy in heterogeneous multicore systems," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP '19, Kyoto, Japan: Association for Computing Machinery, 2019, ISBN: 9781450362955. DOI: 10.1145/3337821.3337865. [Online]. Available: https://doi.org/10.1145/3337821.3337865.

[4] M. W. Azhar, M. Pericàs, and P. Stenström, "Task-RM: A resource manager for energy reduction in task-parallel applications under quality of service constraints," *ACM Transactions on Architecture and Code Optimization*, vol. 19, no. 1, pp. 1–26, Mar. 31, 2022, ISSN: 1544-3566, 1544-3973. DOI: 10.1145/3494537. [Online]. Available: https://dl.acm.org/doi/10.1145/3494537 (visited on 02/08/2023).

[5] S. Gajendra and P. Prashant, "Current trends in heterogeneous systems: A review," *Trends in Computer Science and Information Technology*, vol. 7, no. 3, Nov. 24, 2022, ISSN: 26413086. DOI: 10.17352/tcsit.000055. [Online]. Available: https://www.peertechzpublications.com/articles/TCSIT-7-155.pdf (visited on 02/08/2023).

[6] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang, "Heterogeneous computing: Challenges and opportunities," *University of Southern California*, 2009.

[7] W.-C. Chung, T.-L. Wu, Y.-H. Lee, K.-C. Huang, H.-C. Hsiao, and K.-C. Lai, "Minimizing resource waste in heterogeneous resource allocation for data stream processing on clouds," 2020.

[8] *StarPU: A Unified Runtime System for Heterogeneous Multicore Architectures — starpu.gitlabpages.inria.fr*, https://starpu.gitlabpages.inria.fr/, [Accessed 24-Apr-2023].

[9] O. Sentieys, “Approximate computing for dnn: Improving performance and energy efficiency of deep-learning hardware accelerators with controlled arithmetic approximations,” in *CSW 2021 - HiPEAC Computing Systems Week*, Lyon, France, Oct. 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03494932.

[10] May 2023. [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-rtx-3060-ti.c3681.

[11] Linux Kernel Documentation, *CPU Frequency Scaling Governors*, https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt, Accessed: May 5, 2023.

[12] ArchWiki, *CPU frequency scaling*, https://wiki.archlinux.org/title/CPU_frequency_scaling, Accessed: May 8, 2023.

[13] C. H. Tee, “Chapter 1 - a historical perspective on industrial production and outlook,” in *Digital Manufacturing*, C. D. Patel and C.-H. Chen, Eds., Elsevier, 2022, pp. 1–56, ISBN: 978-0-323-95062-6. DOI: https://doi.org/10.1016/B978-0-323-95062-6.00009-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780323950626000097.

[14] *Theoretical peak*, https://www.sciencedirect.com/topics/computer-science/theoretical-peak, Accessed: May 13, 2023.

[15] Intel, *What is a gpu?* https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html, Accessed: May 13, 2023.

[16] NVIDIA, *What's the difference between a cpu and a gpu?* https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/, Accessed: May 13, 2023.

[17] N. D. Blog, “Programming tensor cores in cuda 9,” *NVIDIA Developer Blog*, 2017. [Online]. Available: https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/.

[18] NVIDIA, *Nvidia jetson agx orin technical brief*, Online, 2021. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf.

[19] H. Ootomo and R. Yokota, “Recovering single precision accuracy from tensor cores while surpassing the fp32 theoretical peak performance,” *The International Journal of High Performance Computing Applications*, vol. 36, no. 4, pp. 475–491, 2022. [Online]. Available: https://doi.org/10.1177/10943420221090256.

[20] NVIDIA, *NVIDIA Volta architecture*, Online, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[21] R. Ñanculef, P. Radeva, and S. Balocco, “Chapter 9 - training convolutional nets to detect calcified plaque in ivus sequences,” in *Intravascular Ultrasound*, S. Balocco, Ed., Elsevier, 2020, pp. 141–158, ISBN: 978-0-12-818833-0. DOI: https://doi.org/10.1016/B978-0-12-818833-0.00009-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128188330000096.

[22] R. Allmendinger and J. Knowles, *Heterogeneous objectives: State-of-the-art and future research*, Feb. 26, 2021. arXiv: 2103.15546[cs,math]. [Online]. Available: http://arxiv.org/abs/2103.15546 (visited on 02/08/2023).

[23] K. Hegde, A. Srivastava, and R. Agrawal, *HTS: A hardware task scheduler for heterogeneous systems*, Jun. 29, 2019. arXiv: 1907.00271[cs]. [Online]. Available: http://arxiv.org/abs/1907.00271 (visited on 02/08/2023).

[24] *Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions - Journal of Big Data — journalofbigdata.springeropen.com*, https://journalofbigdata.springeropen.com/articles/10.1186/s40537-021-00444-8, [Accessed 24-Apr-2023].

[25] *Scaling Up a Multispectral Resnet-50 to 128 GPUs — ieeexplore.ieee.org*, https://ieeexplore.ieee.org/abstract/document/9324237, [Accessed 24-Apr-2023].

[26] C. Shorten, *Introduction to ResNets — towardsdatascience.com*, https://towardsdatascience.com/introduction-to-resnets-c0a830a288a4?gi=947c406f8014, [Accessed 24-Apr-2023].

[27] J. E. Smith, "Characterizing computer performance with a single number," *Commun. ACM*, vol. 31, no. 10, pp. 1202–1206, Oct. 1988, ISSN: 0001-0782. DOI: 10.1145/63039.63043. [Online]. Available: https://doi.org/10.1145/63039.63043.

[28] D. J. Lilja, "Measuring computer performance, a practitioner's guide," Cambridge University Press, 2005, pp. 17–19.

[29] NVIDIA Developer Blog, *Programming tensor cores in cuda 9*, https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/, Accessed: May 13, 2023, Nov. 2017.

[30] J. G. Proakis and D. G. Manolakis, "Digital signal processing," in *The Electrical Engineering Handbook*, R. C. Dorf, Ed., 2nd, Boca Raton, FL: CRC Press, 2005, pp. 263–280, ISBN: 978-0849318054.

[31] M. McCool, J. Reinders, and A. Robison, "Structured parallel programming," in *Structured Parallel Programming*, 1st, Amsterdam: Elsevier, 2012, pp. 39–75, ISBN: 978-0-12-415993-8.

[32] *Red hat enterprise linux 6 - performance tuning guide*, Red Hat, [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/developer_guide/perf.

[33] *Perf: Linux profiling with performance counters*. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main%5C_Page.

[34] *Man() perf_event_open()*. [Online]. Available: https://man7.org/linux/man-pages/man2/perf%5C_event%5C_open.2.html.

[35] *Cpu shielding capability*. [Online]. Available: https://elinux.org/CPU%5C_Shielding%5C_capability.

[36] *Shielded cpus: Real-time performance in standard linux*. [Online]. Available: https://www.linuxjournal.com/article/6900.

[37] WikiChip, *Flops*. [Online]. Available: https://en.wikichip.org/wiki/flops.

[38] Neptune.ai, *Self-driving cars with convolutional neural networks (cnn)*, Blog post, Year of Publication. [Online]. Available: https://neptune.ai/blog/self-driving-cars-with-convolutional-neural-networks-cnn.

[39] D. Leslie, "Understanding artificial intelligence: Ethics and safety," 2019. [Online]. Available: https://www.turing.ac.uk/sites/default/files/2019-08/understanding_artificial_intelligence_ethics_and_safety.pdf.

Appendix

The codebase used for this thesis can be found at https://git.chalmers.se/waqarm/ra_thesis.

ResNet

The ResNet can be run by compiling the resnet8CUDA.cu file in the CUDA_Lib folder. We compiled it with gcc with the -lgomp option. The following command line options are used when running the program:

- –weights: Specifiers where the weights file for the ResNet can be foudn, default location is ../examples/weights. For a given layer that uses weights (convolutional layers) its weight files follow the format _layerIndex_inputChannel_outputChannel.csv where the contents are a 2D comma seperated matrix representing one input channel and output channel combination.
- –gpu: Specifies that a run should be executed on the GPU. Note that at least one of –cpu and –gpu must be set for program to execute. If both are set the GPU will be executed first followed by the CPU.
- –cpu: Specifies that a run should be executed on the CPU. Note that at least one of –cpu and –gpu must be set for program to execute. If both are set the GPU will be executed first followed by the CPU.
- –double: Specifies that one run with double point precision for all values should be executed Note that at least one of –float and –double must be set for program to execute. If both are set a float run will be executed followed by a double run.
- –float: Specifies that one run with floats for all values should be executed Note that at least one of –float and –double must be set for program to execute. If both are set a float run will be executed followed by a double run.
- -rows n: Specifies that the network should operate on a matrix with n rows, default value is 3.
- -cols n: Specifies that the network should operate on a matrix with n columns, default value is 3.
- -in_channels n: Specifies the number of input channels, default value is 3.
- -out_channels n: Specifies the number of output channels, default value is 1.
- –Layer: Specifies that only the selected layer should be executed instead of the whole network. Takes a number between 0 and 12. Requires that a corresponding input be present in "./dumps/input/"
- –dump_output n: Dumps the output value of every executed operation in "./dumps/output/". Main intended purpose is to verify accuracy of GPU and CPU implementations.

- –dump_output n: Dumps the input values of every executed operation in "./dumps/input/". Main intended purpose is to generate valid input files for use with the –Layer option.

MobileNet

The MobileNet can be run by compiling the mobilenet.cu file in the CUDA_Lib folder. We compiled it with gcc with the -lgomp option. The following command line options are used when running the program:

- –weights: Specifiers where the weights file for the ResNet can be foudn, default location is ../examples/weights_mobilenet. For a given layer that uses weights (convolutional layers) the format is weights_layerName where the contents are CSV matrix representing all the channels for that layer. Each row represents a row in the matrix and the values in the row are laid out in (x,y,z) format where x indicates a row element, y indicates a input channel and z a output channel. The program weight_generator.py can automatically generate weight files in this format.
- –gpu: Specifies that a run should be executed on the GPU. Note that at least one of –cpu and –gpu must be set for program to execute. If both are set the GPU will be executed first followed by the CPU.
- –cpu: Specifies that a run should be executed on the CPU. Note that at least one of –cpu and –gpu must be set for program to execute. If both are set the GPU will be executed first followed by the CPU.
- –double: Specifies that one run with double point precision for all values should be executed Note that at least one of –float and –double must be set for program to execute. If both are set a float run will be executed followed by a double run.
- –float: Specifies that one run with floats for all values should be executed Note that at least one of –float and –double must be set for program to execute. If both are set a float run will be executed followed by a double run.
- –Layer: Specifies that only the selected layer should be executed instead of the whole network. Takes a number between 0 and 12. Requires that a corresponding input be present in "./dumps/input/"
- –dump_output n: Dumps the output value of every executed operation in "./dumps/output/". Main intended purpose is to verify accuracy of GPU and CPU implementations.
- –dump_output n: Dumps the input values of every executed operation in "./dumps/input/". Main intended purpose is to generate valid input files for use with the –Layer option.
- -runs n: Specifies how many runs with the above options should be executed. Default value 1. Intended purpose is to test if performance characteristics differed between runs.

For running individual MobileNet tasks the file mobilenet_manual_allocator_setup.cu is used. It uses the same option as above with 3 additions.

- -layer n: Specifies what layer the task belongs to. Takes values between 0 and 29. Note that any task beyond the first initial convolution requires a corresponding input file.
- –batch_norm: If the layer is a convolutional layer (0-26) this runs the batch normalization task on that layer instead of the convolution. Has no effect for layers 27-29.0
- –relu: If the layer is a convolutional layer (0-26) this runs the relu task on that layer instead of the convolution. Has no effect for layers 27-29.

Output Format

For both ResNet and MobileNet performance data for each task is output in the following way

- task_name,time_1,time_2: For CPU execution only time_2 will not appear and time_1 indicates the time the the whole task took to complete. For GPU execution time_1 is the time the task took to execute on the GPU while time_2 also includes the time it took to transfer the memory to and from the GPU.
- Average Power: Average power consumption during task execution.

Weight Generator

weight_generator.py, located in the examples folder, is a handy tool for quickly generating weights files that can be used by the ResNet and MobileNet programs. The generated weights are random and have no resemblance to weights files generated by actual training of neural networks but are useful if one only wishes to test performance data. The program takes in 4 arguments in the following order:

- name: Specifies the name the file will have (the format will be weights_name.csv). A string.
- cols: How many columns will the generated weight file have. A integer.
- rows: How many rows will the generated weight file have. A integer.
- in_channels: How many input channels will the generated weight file have. A integer.
- out_channels: How many output channels will the generated weight file have. A integer.

Following this format a typical execution could look like this "python weight_generator.py example_name 5 5 3 3".

CPU Isolation

There are various scripts in the git repository for running applications isolated on a set of CPU cores. The baseline can be found at

```
ra_thesis/examples/debug/runIsolated.sh
```

which takes five arguments:

1. **Frequency:** the desired frequency to run the CPU cores on.
2. **Threads:** the desired number of threads to run the implementation with.
3. **Application:** the path to the desired application to run.
4. **Input:** parameters for the application.
5. **GPU frequency:** the desired frequency for GPU was also an option to set if needed.

An example of how **runIsolated.sh** could be run can be found in

```
ra_thesis/examples/debug/run_all_N.sh
```

but a simple example is:

```
./runIsolated.sh 115200 4 ./app.out "--param1 --param2" 1300500000
```

Tensor

The tensor implementations can be found in

```
ra_thesis/examples/debug/src/tensorexample.cu
```

which was compiled with

```
nvcc tensorexample.cu -lcublas -arch=sm_80 -o tensorexample.out
```

but a makefile has also been included (in the same directory) for simplicity. The only argument currently is: **output** which prints out data. The file contains a lot of functions but the relevant ones are highlighted in the main function.