Abschlussarbeit im Studiengang IMIT (MSc)

Analysis and Comparison of Performance and Power Consumption of Neural Networks on CPU, GPU, TPU and FPGA

Christopher Noel Hesse

258297

hessech@uni-hildesheim.de

Betreuer:
Prof. Dr. Klaus Schmid, SSE
Dr. Holger Eichelberger

Arbeitsgruppe Software Systems Engineering • Institut für Informatik
Universität Hildesheim • Universitätsplatz 1 • D-31134 Hildesheim

Eigenständigkeitserklärung

Erklärung über das selbstständige Verfassen von "Analysis and Comparison of Performance and Power Consumption of Neural Networks on CPU, GPU, TPU and FPGA"

Ich versichere hiermit, dass ich die vorstehende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der obigen Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich in jedem Fall durch die Angabe der Quelle bzw. der Herkunft, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet und anderen elektronischen Text- und Datensammlungen und dergleichen. Die eingereichte Arbeit ist nicht anderweitig als Prüfungsleistung verwendet worden oder in deutscher oder einer anderen Sprache als Veröffentlichung erschienen. Mir ist bewusst, dass wahrheitswidrige Angaben als Täuschung behandelt werden.

Hildesheim, den 18. Juni, 2021

Christopher Noel Hesse

Abstract

In this work, we analyze the performance of neural networks on a variety of heterogenous platforms. We strive to find the best platform in terms of raw benchmark performance, performance per watt and performance per Euro. To reach this goal, we focused on convolutional neural networks and created several micro- and macrobenchmark applications and used a state-of-the-art real-world network, YOLOv3. We parametrized the benchmarks to analyze the effect of input size, kernel size and other variables on the performance and efficiency.

Our results show that a system using FPGA accelerators is about 7x to 45x faster than a comparable system using high-end GPUs, while consuming about 10% of the power. Novel heterogenous architectures like the Apple M1 integrated SoC offer between 3-5x better performance while drawing 10-20% of the power compared to existing consumer hardware with x86 CPUs and third-party GPUs.

We conclude that the FPGA is the most effective accelerator for neural networks in heterogenous systems. It outmatches powerful GPU server class hardware in both performance and efficiency. The Apple M1 SoC offers the best performance per Euro in our tests.

Acknowledgements

I would like to thank my thesis advisor Dr. Holger Eichelberger for his continuous efforts in helping me complete this work. He always steered me in the right direction and made sure that my work holds up to the high standards of university research. I was encouraged to present parts of my work at the 11th Symposium of Software Performance 2020, where it was met positively. Furthermore, Dr. Eichelberger established the link with the Institute of Microelectronic Systems (IMS) of the Leibniz University Hannover, which allowed me to add more systems to the array of test platforms.

I would also like to thank Gia Bao Thieu and Guillermo Payá-Vayá of the IMS. They reproduced my experiments on their GPU and FPGA servers which provided valuable data to my analysis efforts.

Finally, I would like to express my gratitude to my family for supplying me with graphics art used in Figures in this work. They also provided me with motivation and much-welcomed distraction during the dire times of the worldwide Corona crisis.

Contents

List of Figures v

List of Tables vi

Listings vii

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Goals . . . 2
- 1.3 Structure . . . 3

2 Related Work 4
- 2.1 Heterogenous Computing . . . 4
- 2.2 Neural Networks . . . 5

3 Heterogenous Computing 6
- 3.1 History . . . 6
  - 3.1.1 CPU . . . 6
  - 3.1.2 GPU . . . 8
- 3.2 Accelerators . . . 9
  - 3.2.1 GPU . . . 9
  - 3.2.2 TPU . . . 10
  - 3.2.3 FPGA . . . 11
- 3.3 Heterogenous Architectures . . . 12
  - 3.3.1 Offloading . . . 13
  - 3.3.2 Parallelism . . . 16
  - 3.3.3 Memory Hierarchy . . . 17
- 3.4 Programming Stacks . . . 19
  - 3.4.1 CUDA . . . 20
  - 3.4.2 OpenCL . . . 23
  - 3.4.3 TensorFlow . . . 26
- 3.5 Challenges . . . 27

4 Neural Networks 29
- 4.1 The Neuron . . . 29
  - 4.1.1 Biological Inspiration . . . 29
  - 4.1.2 Mathematical Model . . . 31
- 4.2 Artificial Neural Networks . . . 35
- 4.3 Convolutional Neural Networks . . . 36
  - 4.3.1 YOLO . . . 39

5 Approach **43**
5.1 Metrics . . . 43
5.1.1 Latency . . . 43
5.1.2 CPU . . . 43
5.1.3 RAM . . . 45
5.1.4 Accelerator . . . 46
5.1.5 Power . . . 47
5.2 Benchmarks . . . 47
5.2.1 Microbenchmarks . . . 48
5.2.2 Macrobenchmarks . . . 50
5.3 Tools . . . 51
5.3.1 perfstat . . . 52
5.3.2 Nvidia-smi . . . 53

6 Setup **54**
6.1 Server . . . 54
6.2 Desktop . . . 56
6.3 Edge . . . 57

7 Results **58**
7.1 Full data example . . . 58
7.2 Microbenchmarks . . . 60
7.2.1 2D convolution . . . 61
7.2.2 2D depthwise convolution . . . 63
7.2.3 2D max pooling . . . 64
7.2.4 ReLU activation . . . 64
7.3 Macrobenchmarks . . . 65
7.3.1 CNN inference . . . 65
7.3.2 CNN training . . . 66
7.3.3 YOLOv3 inference . . . 67
7.4 Discussion . . . 68

8 Closing thoughts **77**
8.1 Conclusions . . . 77
8.2 Lessons learned . . . 78

A Appendix **79**
A.1 Results . . . 79

Bibliography **88**

List of Figures

3.1 CPU vs. GPU architecture based on [37] . . . 9
3.2 TPUv3 architecture based on the Google TPU docs . . . 10
3.3 Heterogenous Architecture Tiers . . . 12
3.4 Offloading example . . . 14
3.5 Taxonomy of parallelism in hardware . . . 16
3.6 CPU memory hierarchy . . . 17
3.7 Naive matrix transposition . . . 19
3.8 CUDA architecture . . . 20

4.1 Biological Neuron . . . 30
4.2 The Neuron . . . 31
4.3 Linear Activation Function (plot by WolframAlpha) . . . 32
4.4 Sigmoid Activation Function (plot by WolframAlpha) . . . 33
4.5 ReLU Activation Function (plot by WolframAlpha) . . . 33
4.6 Leaky ReLU Activation Function (plot by WolframAlpha) . . . 33
4.7 Decision boundaries: linear (left) and arbitrary (right) . . . 35
4.8 Feed-Forward (left) and Recurrent (right) networks . . . 35
4.9 Deep Neural Network . . . 36
4.10 Image convolution . . . 37
4.11 Max pooling . . . 38
4.12 YOLO object detection taken from [39] . . . 40
4.13 YOLOv3 *full* vs. *tiny* at 416x416 . . . 41

5.1 CNN microbenchmarks . . . 48
5.2 Regular vs. depthwise convolution . . . 50
5.3 Synthetic CNN layers . . . 51

7.1 $S1_{GPU}$ maxpool2d benchmark results . . . 59
7.2 $S1_{GPU}$ maxpool2d GPU benchmark . . . 60
7.3 $S1_{GPU}$ maxpool2d GPU usage . . . 61
7.4 $S1_{GPU}$ maxpool2d GPU power . . . 62
7.5 2D convolution iteration time distribution . . . 64
7.6 2D depthwise convolution iteration time distribution . . . 66
7.7 2D max pooling iteration time distribution . . . 68
7.8 ReLU iteration time distribution . . . 70
7.9 CNN inference iteration time distribution . . . 71
7.10 $S1_{CPU}$ RAM stats . . . 72
7.11 $S1_{CPU}$ CPU stats . . . 72
7.12 $S1_{CPU}$ benchmark iterations . . . 73
7.13 YOLOv3 inference iteration time distribution . . . 75
7.14 $D2_{GPU}$ YOLOv3 benchmark iterations . . . 75

7.15 $D1_{GPU}$ YOLOv3 benchmark iterations 76

List of Tables

5.1 Linux CPU System metrics 44
5.2 Linux CPU Process metrics 45
5.3 Linux RAM System metrics 46
5.4 Linux RAM Process metrics 46
5.5 Accelerator metrics 47
5.6 Microbenchmarks 49
5.7 Macrobenchmarks 52
5.8 YOLOv3 performance 52
5.9 YOLOv3 TensorFlow vs Darknet 53
5.10 Nvidia-smi metrics 53

6.1 Nvidia DGX-1 specifications 54
6.2 FPGA server specifications 55
6.3 IMS GPU servers 55
6.4 IMS GPU servers (cont.) 55
6.5 IMS FPGA server 56
6.6 HP Omen 15-dh0007ng specifications 56
6.7 Apple Mac Mini M1 2020 specifications 57
6.8 Nvidia Jetson Nano Developer kit specifications 57

7.1 System matrix 58
7.2 2D convolution 63
7.3 2D depthwise convolution 65
7.4 2D max pooling 67
7.5 ReLU 69
7.6 CNN inference 71
7.7 CNN training 73
7.8 YOLOv3 inference 74

A.1 tf-conv2d 79
A.2 tf-conv2d-dwise 82
A.3 tf-max-pool2d 83
A.4 tf-relu 85
A.5 tf-cnn-inference 85
A.6 tf-cnn-train 86
A.7 tf-yolov3-inference 86

Listings

3.1 Pseudocode for multiply-add 8
3.2 Pseudocode for thresholding kernel 14
3.3 CUDA vector addition 21
3.4 OpenCL vector addition 23
3.5 TensorFlow vector addition 26

5.1 /proc/stat contents 43
5.2 /proc/1/stat contents 44
5.3 /proc/meminfo contents 45
5.4 perfstat monitoring loop pseudocode 52

1 Introduction

Neural networks are capable of solving today's problems, such as facial recognition, image classification, object detection, speech synthesis or even autonomous driving. They often outperform traditional approaches in their domain. For example in face recognition, neural networks tend to generalize better to new data than rigid part models that train on predetermined regions of the face, such as jaws, lips, nose, eyes.

With neural networks becoming more and more complex to solve more complex issues, the demand for compute power rises accordingly. Since many networks tend to boil down to highly parallelizable algorithms, CPU hardware is not a good fit. Modern CPUs are designed to be highly complex, with multiple layers of shared caches, instruction prefetching logic or branch prediction logic built-in. Thus, the complexity and cost of a CPU is rather high and the core count is low. Modern multi-core CPU hardware offers no more than tenths of cores for consumers or hundreds of cores for enterprise level users.

Today, much of the software infrastructure for neural networks in general and deep learning in particular revolves around Graphics Processing Units (GPU). These accelerators have been around for a long time and are well understood. Early GPUs were designed only for graphics processing using a fixed-function pipeline in hardware. In the late 2000s, this changed and GPU hardware became GPGPU (general purpose computation) capable. This allowed programmers to run arbitrary code on the GPU, thus being able to exploit the massive core count found on this class of hardware.

The issue with GPU hardware is again the complexity and the cost. Modern GPU hardware as built by Nvidia, AMD or Intel still has function blocks that are dedicated to graphics processing. These circuits are useless for machine learning and only drive up the cost. Another point to consider is the type of data that can be computed. GPUs traditionally operate on floating point numbers. Many neural networks can be trained or converted into a *quantized* representation, which means it uses integer values exclusively. The impact of this process on output accuracy varies from network to network. But for select use-cases such as object detection on embedded systems, this impact may be negligible.

More recently, big corporations have started to look towards other hardware platforms to run neural networks on. Google for example designed its custom, in-house Tensor Processing Unit (TPU) hardware. This kind of hardware specializes solely in matrix and vector math, the most common operations used in machine learning and neural networks. Given the layout and design of the TPU, it is said to be a very efficient and performant accelerator for such applications. Many companies do not have the knowledge or the money at their disposal to design custom Application-Specific Integrated Circuits (ASIC) though.

FPGA vendors such as Intel or Xilinx often tout their hardware as being more efficient than CPU and GPU class hardware for machine learning applications. FPGA stands for Field Programmable Gate Array. FPGAs have been around for a long time already, being used for telecommunication, audio or video signal processing and more. The benefit of FPGA hardware is that they consist of generic array logic, allowing them to be reprogrammed to run a specific algorithm in hardware. FPGA hardware can be built to meet application requirements, e.g. floating-point logic or integer logic support.

In this work, we analyze the performance of neural networks on an array of heterogenous systems. We cover server, desktop and edge class hardware. Furthermore, we collect monitoring data from the hardware, allowing us to infer on hardware capability exploitation. We also measure the power draw of the test systems where possible. To do so, we do not

rely on software readings but connect a measurement device directly between the power outlet and the respective system. Our benchmark suite encompasses convolutional neural network applications, including several micro- and macrobenchmarks.

1.1 Motivation

When employing state-of-the-art neural networks in projects, engineers and managers often face the task of choosing a suitable hardware platform to run on [1]. This applies to cases ranging from edge computing to desktop class machines as well as high performance computing including supercomputers. Most of the time, there will be numerous constraints impacting the decision making process such as performance criteria, efficiency or cost. To some degree, the same constraints affect the development as well, e.g. in terms of the time wasted by a developer waiting for models to be trained. Scouting for viable hardware and accompanying software solutions can be a daunting and time-consuming task. Most of the time however, absolute chip performance, efficiency, etc. is not the only factor one must take into account. Rather, the entire heterogenous computing package (including memory transfer latencies and other overhead) dominates the project outlook.

Neural networks are a good fit for measuring and comparing heterogenous system performance. They are modeled after the human brain, where millions of neurons are interconnected. Accelerators used in heterogenous systems usually operate on hundreds or thousands of simple cores, compared to the highly complex core design used in CPUs. Thus, accelerator cores mandate workloads with a high degree of parallelism for optimal exploitation.

At implementation time, one needs to be aware of the different programming models used to implement algorithms on the various accelerators. There is a wide breadth of software implementation standards, some proprietary (like Nvidia CUDA) and some open-source (like OpenCL). Moreover, some of the standards target only a single hardware platform such as GPUs, whereas others enable a single implementation on multiple targets.

At a high-level, there are multiple established frameworks which provide abstraction layers to implement algorithms independent of the underlying programming model of a CPU, GPU or others. This will often save precious developer time and reduce required entry skill. At the same time though, it may reduce the roofline performance, as it has to be able to cater to multiple vendor backends at compile- or runtime.

1.2 Goals

This thesis aims to answer the following research questions:

With regards to select neuronal network use-cases,

1. which platform is the most performant in terms of iterations per time?
2. which platform is the most efficient in terms of iterations per energy?
3. which platform is the most cost-effective in terms of iterations per money?

An iteration in this case describes a single part of code that is executed in a loop during benchmarking. For example, in case of neural network inference, *iteration* refers to a single forward pass. In case of training, it means a single training iteration using all training images.

To be able to answer these questions, we first had to establish a set of performance and energy metrics to measure. Second, we had to find a way to acquire the metrics in question

by means of extensive benchmarking and monitoring. As a last step, we needed to analyze and aggregate the acquired data to be able to draw conclusions.

1.3 Structure

This work is structured as follows: first, we take a deep dive in the topic of heterogenous computing in Chapter 3. We reiterate on the history of processor development and go on to explore GPUs as one of the first upcoming coprocessor designs. As a follow-up, we talk about other kinds of accelerators used for machine learning and neural networks especially. In the final part of the chapter, we explore various modern programming environments for heterogenous systems.

In Chapter 4, we talk about neural networks. We begin by explaining the development from understanding the biological neuron as part of the human brain to artificial neural networks. Afterwards, we show the inner workings of convolutional neural networks in particular. Finally, we present the YOLOv3 network as a real-world state-of-the-art model.

Chapter 5 details the approach we took to evaluate the performance neural networks on heterogenous systems. We define various metrics which we collect during the benchmark runs, that allow us to reason about hardware usage in relation to changing benchmark parameters. Afterwards, we present the benchmarks we developed. Finally, we list some of the tools used to acquire monitoring data such as hardware usage, power draw and more.

In Chapter 6, we list the system specifications of the machines used in our experiments.

We present the benchmark results in Chapter 7. At the end of the chapter, we discuss the observed results.

Finally, Chapter 8 concludes our work with lessons learned and closing thoughts.

2 Related Work

Without a doubt, there has been a resurgence of interest in neural networks in the last decade. Yet, much of the research work is focused on the performance (in the sense of accuracy) of the networks. In the context of our performance analysis, we typically use the term *performance* to describe the computing capabilities of a system. New architectures are developed each year to outperform existing state-of-the-art models on tasks like speech synthesis, object detection, handwriting recognition and more.

The remainder of this chapter is structured as follows. First, we present work on generic heterogenous computing performance in Section 2.1. Next, Section 2.2 is focused on performance evaluations of neural networks in particular.

2.1 Heterogenous Computing

Chung et al. present a model for single-chip heterogenous computing in [9]. They argue that combining traditional (CPU) with unconventional cores (e.g. GPU, FPGA) allows for greater energy efficiency. They find that such unconventional cores must be able to exploit sufficient parallelism in the workload. Furthermore, the off-chip bandwidth poses a first-order threat to performance and efficiency gains according to the authors. Finally, they argue that less efficient but more flexible hardware designs such as GPUs or FPGAs are competitive with custom logic designs, given a parallelism grade of 90-99%. Finally, they conclude that their model makes a strong case for unconventional cores in heterogenous computing architecutres.

In [31], Mittal and Vetter survey a range of heterogenous computing techniques. In doing so, they exclusively look at CPU-GPU systems. They find that programming algorithms for such systems is still tedious, often mandating custom code for each processor type. On top of that, they state that one must first identify suitable subtasks to exploit the accelerator efficiency and performance gains. Mittal and Vetter conclude that ultimately, fully automatic toolchains would be required so that more application domains can profit from the benefits of heterogenous computing.

Schulte et al. describe AMD's vision for exascale computing in [42]. With AMD being a GPU manufacturer, they argue in favor of GPUs to be used for heterogenous computing architectures. Their reasoning for going with GPU hardware as the accelerator of choice is that their customers desire not to pay for custom components only useable for high-performance computing.

In [8], Che et al. present Rodinia, a benchmark suite to evaluate heterogenous computing capabilities. They focus on multi-core processors and support CPU and GPU hardware. The authors claim to cover parallel communication, synchronization and power consumption, providing valuable architectural insight. They explicitly hint at the importance of memory-bandwidth limitations in heterogenous computing architects.

Danalis et al. developed the Scalable Heterogenous Computing (SHOC) benchmark suite [11]. They state their initial focus is on systems featuring GPU hardware and multi-core processors for compute acceleration. The benchmark suite uses the Khronos OpenCL API and Nvidia's proprietary CUDA to implement the individual test programs. Similar to our work, the SHOC suite is composed of microbenchmarks and higher level applications.

In [18], Hu and Rossbach present ALTIS, a GPGPU benchmark suite. They argue that the benchmark suites we presented earlier, Rodinia and SHOC, were aimed at last-gen GPU hardware regarding workload and capabilities. In contrast, ALTIS is touted as featuring

modern workloads including deep neural networks, graph analytics and crypto-currencies. The authors adapted selected benchmarks from the other suites and developed new applications as well. One interesting decision by the authors is to focus on CUDA platforms exclusively, meaning it can only be used to evaluate systems with Nvidia GPU accelerators.

2.2 Neural Networks

Bianco et al. evaluate a wide range of deep neural network architectures in [3]. They use two heterogenous systems in their work, one workstation equipped with a Nvidia Titan X (Pascal) and one Nvidia Jetson TX1 developer kit. Their analysis features performance indices for accuracy, model complexity but also computational complexity, memory usage and inference time. The authors find that model complexity can be used to directly estimate the total memory utilization of an application. They also state that even the networks with low model complexity are barely able to run on the embedded test system, requiring too much GPU memory.

In [22], Karki et al. present Tango, a benchmark suite for accelerators featuring deep neural network applications. The benchmarks are implemented in CUDA and OpenCL, which means that they will be able to run on most modern GPU hardware. The authors evaluate their benchmark suite on three systems, one equipped with a server class GPU, one using a mobile GPU and one that uses a mobile FPGA. They find that the mobile FPGA platform is more power efficient than the mobile GPU platform.

Zhu et al. implemented a benchmark suite to analyze the training of deep neural networks in [52]. Their benchmark applications cover a range of modern machine learning domains such as image recognition, speech recognition, machine translation and more. Furthermore, they developed a novel toolchain to perform performance analysis, focusing on profiling the memory usage of data structures in neural networks.

3 Heterogenous Computing

Heterogenous computing is already prevalent in most of today's devices. For example, most consumer devices feature dedicated silicon for graphics rendering and processing. Even for small office PCs, software rendering is not sufficient to provide a decent user experience so they are equipped with graphics processing units (GPUs). In the mobile world, devices are resource constrained and operate on a tight power budget, e.g. smartphones. Network equipment usually features dedicated circuitry to offload checksum computation, a seemingly small yet recurring task, instead of running on the main central processing unit (CPU).

In this chapter, we first explore the history of select compute hardware in Section 3.1. Section 3.3 introduces heterogenous architectures, the concept of offloading computation and describes common memory hierarchies. Once the architectures are introduced, we go on to present the currently used programming models in Section 3.4 and link them to the architectures they can run on. Finally, we discuss the challenges faced by industry professionals and researchers when trying to select a heterogenous system for a given project in Section 3.5.

3.1 History

After reading the introduction of this chapter, one may ask themselves why heterogenous systems are necessary in the first place. After all, CPUs became more powerful (in terms of compute power) and less power-hungry ever since the first computer was developed. Why do we not focus on further improving CPUs, since they can handle any possible workload and the programming is rather easy? We try to answer this question by exploring the history of CPU and GPU hardware. After this, we motivate the development of advanced accelerator hardware.

3.1.1 CPU

Traditionally, CPUs were the only kind of hardware applicable for running general purpose computations. Ultimately, when we think of *performance* we mostly think about the time it takes the hardware to complete a given task. In games for example, the main metric of performance is the *frame time*, aka the time it takes for a single frame to be rendered. It is important to understand which parts of the hardware influence the performance one is able to achieve when running a program. For CPUs, one of the most relevant key properties is its frequency. The frequency determines the number of cycles the CPU can execute in a given time period. For example, a frequency of three megahertz (MHz) would indicate that a CPU can complete 3.000.000 (three million) cycles per second. To understand how that relates to actual program code being executed, it is important to understand what a cycle actually is. A CPU is designed to execute so-called *instructions*. When writing C code for example, the compiler translates the high-level code into the low-level assembly language. That assembly code (also called machine code) contains all the instructions necessary to perform the requested operations. We deliberately leave out some details regarding registers and such here for brevity. The key takeaway here is that the time needed to run a program is directly related to the number of instructions the CPU can execute per second (which in turn is directly proportional to the number of cycles it can execute).

From the above, one may understand that all we need to do to run programs faster is to just keep increasing the frequency (disregarding memory transfer times and such). While true in theory, there are physical limits as to how high the frequency can be for a given chip.

$$P = CpfV_{dd}^2 \tag{3.1}$$

Kogge et al. [23] state the equation for power density as shown in equation 3.1, where P is the power density in watts (per area), C is the capacitance, p is the transistor density, f is the CPU frequency and V_{dd} is the supply voltage. We can already see that the frequency f is directly proportional to the output power density. Brodtkorb et al. state that modern CPUs are already approaching the physical limits with current cooling technology [5]. Thus, increasing the frequency will not work in the future to accelerate machine code execution, unless new cooling technology emerges.

The next step in the evolution of CPU designs were the multi-core architectures. Instead of just one big and powerful core, silicon vendors started to pack multiple cores onto a single die. Initial designs featured two processor cores, whereas modern Intel or AMD processors feature from quad-core up to 32 or 64 core designs for high-end server class CPUs. The cores are not completely independent of each other, as they share caches and other processor pipeline integrated circuitry. We will talk about the memory hierarchy in a later section in this chapter. The multi-core designs do not solve the problem posed by the frequency limitation though. Traditionally, serial programming was the norm. A program had a single entry point, a serial control flow and either a defined exit point or a loop it would run. Thus, ramping up the frequency would always make the program run faster, disregarding any input/output (I/O) induced latencies and memory throughput limitations. With multi-core CPUs, running serial code did not see any runtime improvements, sometimes even an increase in compute time. Scheduling work on multiple cores requires a complex task scheduler in the operating system. An example for this is the completely fair scheduler (CFS) in the Linux kernel. In the worst case, a program is scheduled on a different core in each cycle, which leads to cache trashing as the closest cache lines are specific to each core, leading to decreased performance. There are techniques such as *task pinning* which forces a process to be run on a specific core each time (thus bypassing the placement by the OS scheduler). But even with this workaround, a CPU with n cores could at best perform on par with a single-core CPU. To get better performance out of a single program on multi-core CPUs, *multiprocessing* and *multithreading* are required. The former means spawning multiple processes that communicate with each other and may even share portions of main memory. This of course requires careful consideration and synchronization techniques by programmers. The latter, *multithreading* describes the spawning of multiple threads within a single process and executing these threads in parallel. Since these threads often share data in RAM as well, synchronization is required again. There are methods to circumvent this such as *lockfree* programming or *atomics*, but they are often limited to special cases or hard to apply. In the end, parallel, multithreaded code is always harder to write than serial code, because more potential issues like race conditions and such warrant the programmers' attention.

Advanced technologies such as *hyperthreading* allow running multiple processes or threads on a single CPU core. Since they do not help with the problem at hand any more than multi-core CPU designs do, we do not elaborate further on them.

There is another technology which is widely implemented by processor manufacturers to help overcome the frequency limitation. In simple terms, additional circuitry is added to the CPU so it can execute multiple instructions in a given cycle. One common extension is called fused multiply-add (FMA). For example, imagine a simple instruction set where the assembly instruction *add* is used to add a number to another number. *mov* is used to copy

bytes from a source to a destination memory location. Finally, *mul* is used to multiply a number with another number.

```
1 mov a, r7
2 mul a, 0x0F
3 add a, 0x0A
4 mov r7, a
```

Listing 3.1: Pseudocode for multiply-add

Listing 3.1 contains pseudocode for a simple multiply-add program. In line 1, the content of register $r7$ is copied into the special accumulation register a, where all the calculations take place. In line 2, the value currently stored in the accumulation register is multiplied by 0x0F. The consecutive *add* operation adds the value 0x0A onto the result of the previous multiplication. Finally, the result of the computation is copied back to the register $r7$ in line 4. Without FMA, these four lines of program code would boil down to four instructions being executed serially by the CPU within four cycles. With FMA however, the *mul* and *add* instructions can be executed in a single cycle, so we only need to spend three cycles running the code.

Techniques such as FMA or other modern CPU extensions such as Intel's advanced vector extensions (AVX) drive up the complexity and cost of the hardware. This is a major drawback in modern CPUs. Consumers may be able to pay a few dollars more for advances in CPU hardware. But in the embedded world, every cent counts since so many of these chips are manufactured. Devices need to be cheap to produce and cheap to replace, unlike high-end server class hardware for example. Thus we can see why people began to look for alternative processors for generic computations.

3.1.2 GPU

One of the first alternative hardware designs that emerged were GPUs. Once GPUs started to become popular, they were used for gaming and specifically designed for this sole purpose. The hardware was laid out as a fixed-function pipeline to execute tasks such as viewport mapping, depth testing and such as fast as possible. Perhaps the most important issue with this hardware design was its inflexibility. Each time game developers wanted to add new effects to the graphics application programming interface (API), e.g. Microsoft DirectX or OpenGL, new hardware was required to implement the functionality. Thus, it was quickly understood that this kind of design was too inflexible going forward and the landscape of GPU hardware design drastically changed toward general purpose programming on GPUs (GPGPU). Highly specialized GPU function blocks were replaced with less specialized ones, for example generic vector addition or multiplication blocks. McClanahan states the first GPGPU card was the Nvidia Fermi GPU released in 2010 [30]. According to him, "Generation VII" of GPUs continued the trend from fixed-function cores towards CPU-like cores. In this case, the compute cores are called CUDA cores by Nvidia, just like the programming language they invented to program those cores. Modern GPU hardware features complex schedulers and advanced memory hierarchies, which we will talk about in Section 3.3.3. Many algorithms of today, including machine learning and artificial intelligence ones, lend themselves well to GPGPU hardware. However, it is important to understand what kind of work a program is really doing to choose the best fitting and cost-effective hardware to run it on. For example, Lee et al. state that the performance gap between a Nvidia GTX280 GPU and an Intel Core i7-960 is about 2.5x on average [26] instead of 100x or even 1000x as indicated by similar research.

In the next section, we explore common heterogenous architectures of today and talk about memory hierarchies, design choices and reasons why companies spend billions of dollars every year to try and come up with even better accelerators to run their workload on.

3.2 Accelerators

Since this work is about performance analysis of neural networks, we focus on accelerators exclusively and do not touch upon security coprocessors in detail. In this section, we present the accelerators used in systems that were part of our work. We briefly introduce each accelerator kind and compare them against each other in terms of deep learning and neural network compute capabilities.

3.2.1 GPU

Graphics processing units (GPUs) have been around since decades. Over time, they evolved from simple video output compositors into complex general purpose computation devices. The old kinds of GPUs, like 2D or 3D fixed-function accelerators are not of interest with regards to this work. We solely focus on GPGPU capable devices, since they support freely-programmable compute shaders necessary to execute arbitrary algorithms ("kernels"). This flexibility allowed GPUs to be used in a wide range of applications including machine learning [38], linear algebra [25], statistics [27] and others.

In contrast to a CPU, a GPU features less complex core logic. Features like branch prediction or out-of-order execution may be entirely missing, while no modern CPU would be competitive in performance without these. Instead, a GPU sports a vastly higher core count leading to a much higher possible degree of parallelism [15]. Thus, GPUs are used for work that can be parallelized without many serial dependencies. Highly serial tasks are best left to the CPU, where the powerful cores will usually achieve better performance than the more simple GPU cores. Figure 3.1 illustrates the difference between CPU and GPU hardware architectures. The difference in core count and auxiliary logic is shown in Figure 3.1.

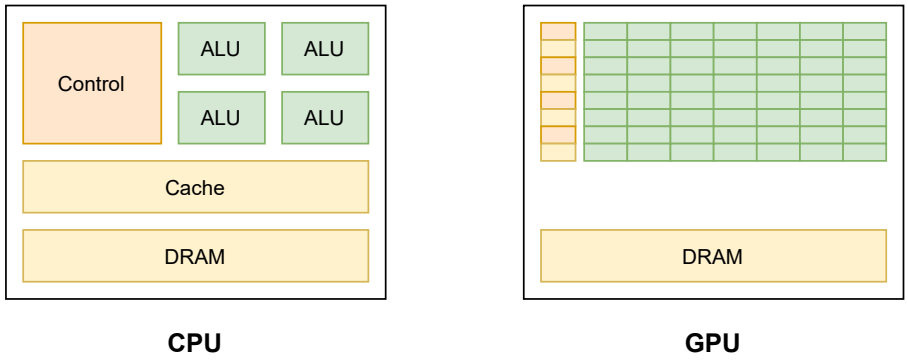


Figure 3.1: CPU vs. GPU architecture based on [37]

GPU hardware is prevalent in most consumer hardware today. As such, it is the most suitable target for compute offload for demanding applications. Most dedicated GPU hardware uses the PCI Express connector standard, which makes it possible to easily add, remove and upgrade the accelerator hardware. Enterprise class GPUs are mostly built on top of their consumer counterparts with various features added on top. Thus application programmers can target both consumer and enterprise users with the same code, as it will be able to run on almost all available hardware - given that the software stacks are compatible.

3.2.2 TPU

A tensor processing unit (TPU) is a dedicated accelerator for neural networks. It was first developed by Google as custom application-specific integrated circuit (ASIC) with the goal of accelerating machine learning [1]. According to Google, the TPU was designed to specifically accelerate TensorFlow workloads.

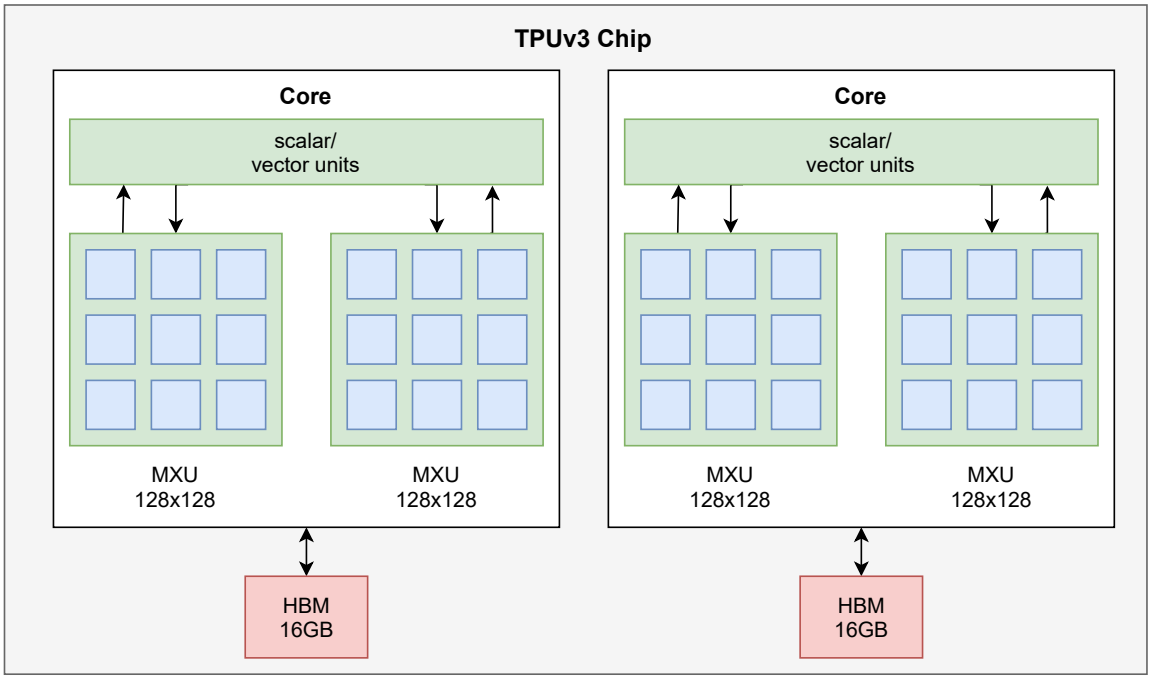


Figure 3.2: TPUv3 architecture based on the Google TPU docs

Figure 3.2 shows the TPUv3 architecture. Google Cloud TPUv3 nodes are equipped with four of these chips, all interconnected on a single board. Each chip features two cores, as illustrated in the figure. A core is made up of an array of scalar, vector and matrix units (MXU). As can be seen from the figure, the MXU array provides the bulk of the TPU compute power. According to Google, each MXU can perform 16.000 multiply-add operations per cycle at bfloat16 precision, which is a 16-bit floating point representation designed for neural networks.

The TPU is quite similar to the early GPU designs in terms of how it came to be. Where GPUs emerged to handle the very specific task of graphics processing, including lighting, shading and triangulation, TPUs are built to compute on n dimensional matrices (also called "tensors") as fast as possible. Just like the GPU, TPU hardware features its own dedicated high-bandwidth memory to load and store compute operands. Analogous to multi-GPU configurations like Nvidia SLI or AMD CrossFire, Google allows multiple TPU devices to be interconnected via high-speed network interfaces.

The Google Cloud TPU architecture has arrived in the world of embedded technology with the Coral Dev Board [6]. According to Google, it is meant to move neural network inference out of centralized datacenters on to embedded devices. The vastly more resource intensive training process of neural networks remains with the datacenters for now, though.

Unfortunately, accessing TPU hardware for custom projects is hard, as Google does not sell them. Instead, Cloud TPU hardware is available through a rent-on-demand model [2].

[1] `https://cloud.google.com/tpu/docs/system-architecture`
[2] `https://cloud.google.com/tpu/pricing`

The only hardware that can be bought is the Coral Dev Board with the embedded variant of the TPU chip. Another restriction is imposed by the Google Colab Terms of Service (ToS), which does not allow you to pay for the service outside of a selection of countries, including the United States and Canada as of 2020 [3] [4]. Colab is the interface through which one can run TensorFlow code on the rentable TPU hardware.

3.2.3 FPGA

Field programmable gate array (FPGA) describes integrated circuitry that can be designed to fit a certain task after manufacturing. The circuit layout can be re-programmed using specific hardware description languages (HDLs). A FPGA chip is made up of an array of logic blocks, which can carry out complex custom operations or simple logic gates like AND or OR. According to Wikipedia [5], most FPGAs usually feature local memory for each logic block. This allows the FPGA to carry out stateful operations at each block. Advanced FPGA designs may feature large on-chip shared memory blocks, bidirectional buses and more complex circuitry. Depending on the requirements, FPGA logic blocks can be designed to handle a variety of data types, including integer and floating-point data.

There has been huge research interest in employing FPGA hardware for machine learning. GPUs are very expensive, consume large amounts of power and are highly complex chips, yet they are the most used hardware for machine learning and deep learning workload acceleration. Naturally, researchers and industry professionals are looking for alternatives to accelerate their algorithms. Custom ASIC hardware is expensive to design and manufacture, but FPGAs have been around for decades and can be bought or built for less money.

Xilinx research compared a Nvidia Tesla P40 GPU (40 TOP/s INT8) with their Ultrascale+TM XCVU13P FPGA (38 TOP/s INT8) [32]. They argue that GPU ALUs typically only operate on FP32 and sometimes FP64 data. Since machine learning applications can usually operate on integer data with a negligible loss in precision, this potential speedup cannot be leveraged using GPUs. The silicon changes made by GPU vendors only resulted in a 3X speedup for INT8 computation vs FP32 on the Nvidia Tesla P40 according to Xilinx. Furthermore, Xilinx argues that the much larger on-chip memory size (500MB) compared to the Nvidia Tesla P4 (80MB) allows it to hide memory latencies better. They also tout a much lower power usage compared to GPUs often requiring several hundreds of watts at peak load. Compared to a Nvidia Tesla V100, they say their Virtex Ultrascale+ offers up to four times better compute efficiency.

The Xilinx claims appear to be backed up by Microsoft researchers [36], who report that an Intel Arria 10 FPGA performs ten times better in terms of power consumption on an image classification task. Nvidia's recently introduced Tensor cores for tensor operations are said to be able to narrow the gap and bring their GPUs to the same level of efficiency as FPGAs though.

Intel conducted research on deep neural network (DNN) performance using their Stratix 10 FPGA and a Nvidia Titan X Pascal GPU [34]. They developed a ResNet implementation with marginally reduced precision for the weights. They say their version of RedNet is within 1% of the full-precision ResNet implementation. Intel researchers report a 60% better performance of the Stratix 10 FPGA, while achieving a 2.3X better performance

[3] `https://colab.research.google.com/pro/terms/v1`
[4] `https://colab.research.google.com/signup`
[5] `https://en.wikipedia.org/wiki/Field-programmable_gate_array`

per watt rating. While the original ResNet benchmark had to be specifically adjusted for the FPGA, the results certainly weigh in favor of it against the current GPU generations.

Independent researches such as Cong et al. [10] show that a Xilinx Virtex 7 FPGA is able to achieve almost equal performance compared to a Nvidia K40c GPU for some kernels of the popular Rodinia GPU benchmark suite. At the same time, the FPGA consumed only about 28% of the power according to the authors. They ported the GPU kernel code to the FPGA, including additional optimization like cache tiling etc. Cong et al. conclude that the FPGA performance was held back by the lower clock frequency and lower off-chip memory bandwidth.

3.3 Heterogenous Architectures

As mentioned in the beginning of the chapter, heterogenous computing is ubiquitous for performant, efficient and safe systems. In this section, we present various heterogenous computing architectures, some common and some rare. We outline the vastly different purposes served by these architectures and categorize them accordingly.

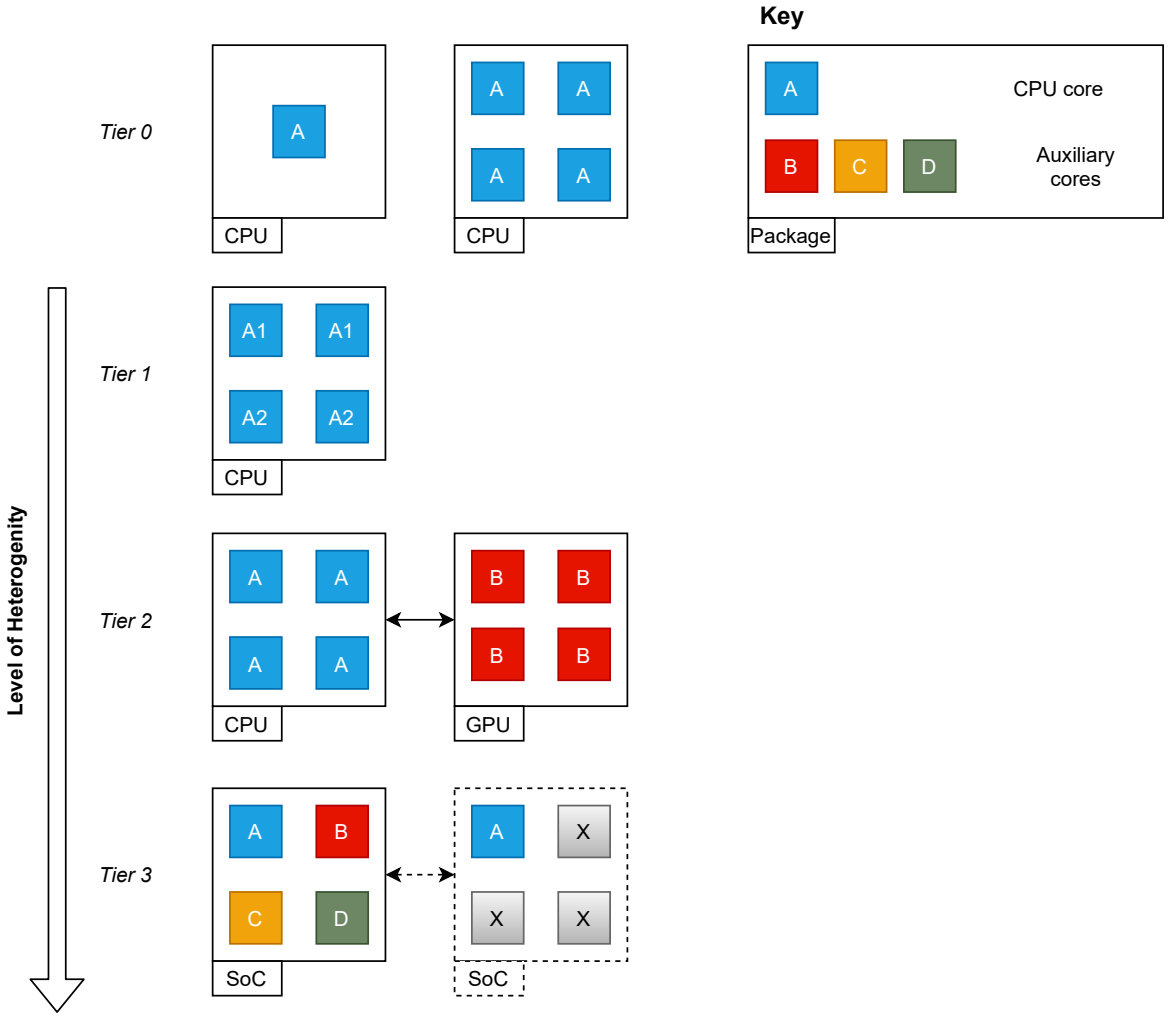


Figure 3.3: Heterogenous Architecture Tiers

In Figure 3.3, we present an arbitrarily chosen ranking of architectures, rated by their level of heterogenity. Tier 0 encompasses fully heterogenous designs, such as traditional CPUs using only one kind of core. The first multicore designs fit into this category as

well, as do recent AMD Ryzen chip designs. Tier 1 describes systems that use different cores, albeit still belonging to the same overall chip family. For example, ARM Cortex CPU designs fit into this category: they often feature a number of high-performance cores along with some low-power ones, which helps mobile use-cases. Tier 2 describes package designs with a CPU and one accelerator chip, each with an arbitrary number of cores. Finally, Tier 3 describes current system-on-chips (SoCs) like the Apple M1 or Intel Tiger Lake series, where a single package contains a CPU and multiple auxiliary cores. Most of the time, auxiliary cores such as neural processing units (NPUs), GPUs or other chips will even share memory with the CPU. Vendors sometimes combine multiple SoCs into a single package, for example cellphone chips with one SoC containing the CPU, a GPU, a security processor and more and a second SoC with an FPGA and additional CPU cores for data processing logic.

In the following sections, we talk about approaches that can be used to exploit heterogenous systems for optimal performance, efficiency or other goals. We discuss the software requirements and various tools used to achieve heterogenous applications.

3.3.1 Offloading

Heterogenous computing is all about leveraging alternative processing core designs to achieve greater efficiency and performance than full blown CPU cores could. At the same time, it is important to understand that the goal is not to replace CPUs. Instead, coprocessors are designed so portions of a task can be executed on them, while other parts are still carried out by the CPU. For simple computations such as summation of all values in an array or vector maths a GPU may be sufficient to perform all tasks on its own. When moving to more complex applications however, we can see why that is not working out.

One example of a sufficiently complex real-world application where offloading can be used is real-time image processing. To a computer, images are just arrays of numbers, one dimensional to be specific. To align more with the human way of thinking about images, they are usually represented as two-dimensional arrays, also called (2D) *matrices*. Such a matrix has one cell per image pixel in the case of grayscale images or multiple cells for color images, e.g. three for RGB. This is also called a *packed* image representation. There are of course other image memory representations such as planar ones, but we will discard them here for sake of brevity. In the most simple case, a program may allocate main memory (RAM) to hold the image pixel values, fill them with some values and then pass it to the coprocessor so it can execute a *kernel* on the image. Such a kernel can perform numerous operations on the image. Simple kernels operate on a single cell of the image, while more complex ones exploit the spatial information of the image, i.e. operating on three pixels at once. The coprocessor is usually programmed in a way that its own scheduler takes care of iterating over the input array and running the kernel when appropriate. Once the processing is done, the memory content needs to be read back into main memory to continue processing.

Figure 3.4 depicts an offloading example application. First, memory is allocated by the CPU in RAM and zeroes out the contents. The CPU then fills the memory with values, in this case grayscale pixel intensities. If we assume eight bit per pixel, that leaves us with a range of [0, 255] for the intensity values. The figure shows the image as a tile of four pixels, where the top left hand tile has an intensity of 255 (white) and the bottom right hand tile has an intensity of 0 (black). The other two pixels have intensities somewhere in between, with the top right hand one being closer to 255 and the bottom left hand one being closer to 0. Our example application now copies the array to the coprocessor, in this case a GPU. This copying operation traditionally involved the CPU copying the bits

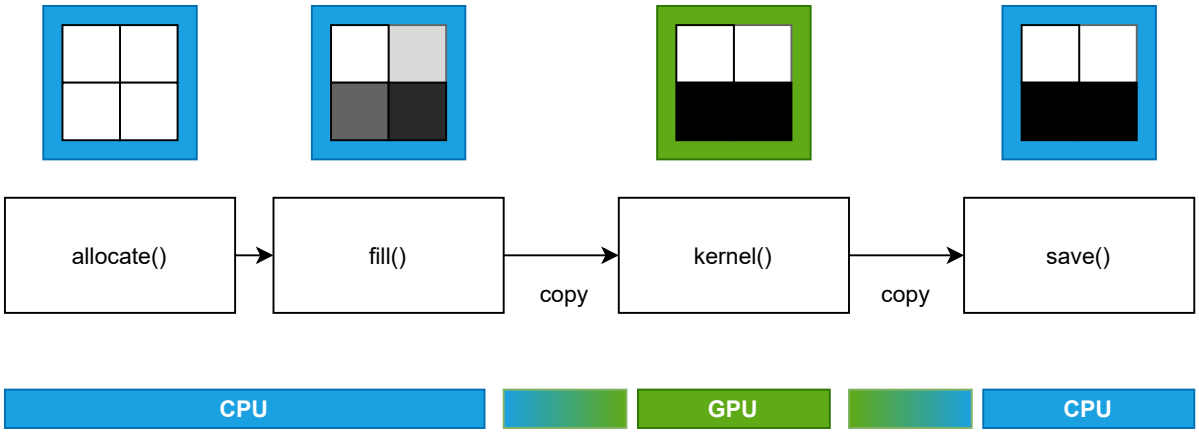


Figure 3.4: Offloading example

from main memory and transferring them to the GPU memory. Modern GPUs support techniques such as direct memory access (DMA), allowing the device to directly access the RAM and copy the contents on its own. This is useful because the CPU does not need to waste any cycles for copying the memory, a phase where the GPU needs to wait for the transfer to finish and thus being blocked anyway. The kernel being executed on the GPU runs code akin to the pseudocode shown in Listing 3.2. One kernel iteration operates on a single matrix cell, which is a single pixel intensity. Both input and output type are eight bit intensities (*u8*). The function described by the kernel is a two-way thresholding: intensities higher than half of the maximum intensity are clamped to the maximum value, while others are set to the minimum value. The grayscale input image becomes a purely black and white image in the process. Once the kernel is run over all input matrix cells, the array contents reside in GPU memory. We now need to transfer them back to the main memory which is CPU accessible to actually use the array contents or share it with other applications. The only thing we can do on the GPU at this point is to run other kernels on the array contents, e.g. blurring the image (which would make it a grayscale image again).

```
1 fn kernel(input: u8) -> u8 {
2     if input > 128 {
3         return 255
4     } else {
5         return 0
6     }
7 }
```

Listing 3.2: Pseudocode for thresholding kernel

So far, we designed a simple image processing pipeline involving a coprocessor. Since only some parts of the program are run on the GPU, this is called *offloading*. The benefits of such a seemingly complex architecture are not immediately obvious. For example, most GPUs support the creation of buffers in their own memory directly. The pixel intensities could also be filled by the GPU itself, so the first two steps *allocate()* and *fill()* do not need to be executed by the CPU anymore. Instead, the CPU can simply enqueue these commands for buffer creation and value assignment directly to the GPU, where a task scheduler will run them. It is easy to come up with a real-world example where the architecture proposed in Figure 3.4 is required though. Processing arbitrary image data generated by the computer is usually not interesting in production applications. Instead, image pixel data may be

acquired by a digital camera, connected to the PC via universal serial bus (USB) or other means. In most popular computing systems, peripherals such as cameras are connected to a bus such as USB, which in turn is accessible only by the CPU. Reading pixel data from a camera involves complex drivers which are run by the operating system. If one wants to perform direct camera access on e.g. a GPU, that GPU would need to run a full-blown operating system with drivers for the peripherals just to capture images. Such a system would be too complex and would drive up the cost of the product considerably. Since the CPU needs to handle peripherals anyway and already runs a complex operating system to begin with, it is the right processor to grab the image data from the camera. The *fill()* step of the example shown in Figure 3.4 is now replaced with a *capture()* step. This is of course discarding details about the image format negotiation, buffer allocation and more that needs to happen before capturing image frames from the camera device.

Offloading is also applied in other contexts. For example, modern smartphones feature dedicated processor cores and memory for running secure applications. They usually run an entire operating system simultaneously to the main OS (e.g. Android) which runs on the CPU. One example for such a security OS is MobiCore by Giesecke & Devrient as presented by Spitz [46]. It is found in most smartphone SoCs of today such as Qualcomm Snapdragon or Samsung Exynos. Use-cases include mobile payment authentication or video content decryption of digital rights management (DRM) protected content. Most content distributors such as Netflix, Amazon Prime Video or even Spotify Music only provide DRM protected content through their subscriptions, which explains why these coprocessors are so ubiquitous. Even the most simple music players require dedicated silicon to offload audio decoding tasks to, because their main CPU is not powerful nor efficient enough. Instead, digital audio processors (DSPs) are highly tuned to perform the floating point math operations as fast as possible, while maintaining minimal energy consumption.

Another important benefit of *offloading* is the domain separation. Looking back at our camera image processing application, systems engineers are usually trained to work with peripheral hardware such as cameras. They know how to "talk" to the hardware by writing drivers to get the image buffer transferred into RAM. However, they usually are not trained specialists in the image processing domain. Granted, a simple thresholding operation may not pose a real challenge to any software engineer. But more complex compute kernels such as image convolution algorithms are usually better implemented by domain experts. These experts need not to worry about the camera hardware. Their work involves writing the code for the accelerator (in this case the GPU), ensuring optimal task placement on the many GPU cores, efficient scheduling and algorithm soundness. The tooling for writing GPU kernels is often complex to setup and even more complex to update (e.g. Nvidia CUDA or Khronos OpenCL). "Offloading" this work to the domain experts frees the system engineers from the burden of GPU specific debuggers to make the code work. Of course, this domain separation is more work than just running all code on the CPU. However, that is usually not an option because of the vast benefits one can exploit when running code on accelerators. The efficiency of offloaded computation often depends on the amount of parallelism of the work package, which we will explore in the next section.

Summarizing, we can conclude that *offloading* is commonly used for the following (non-exhaustive) reasons:

- Performance
- Efficiency

- Security
- Domain separation

3.3.2 Parallelism

In the previous section, we outlined the benefits of offloading work to coprocessors. We touted them to be more efficient, more performant or more secure. In this section, we want to explore the reasons behind the first two aspects - efficiency and performance - of the accelerator chips.

Modern CPUs are made of more than just their cores, some caches and registers. They feature highly complex prefetching pipelines, branch predictors and more clever innovations, all aimed at making serial code execution as fast as possible. Accelerators on the other hand are usually made of cheap, small and specialized cores in great numbers. A modern Nvidia consumer grade GPU, the RTX 3080, features more than 8700 CUDA cores [6]. As we already elaborated in the previous section, this poses a software engineering challenge, because serial control flow programs do not benefit from many-core hardware designs. To exploit the massive parallel execution that accelerator chips are able to perform, the workload has to be structured accordingly. This is paramount to good efficiency and performance with accelerators. Figure 3.5 illustrates Flynn's taxonomy [14] of parallelism in hardware.

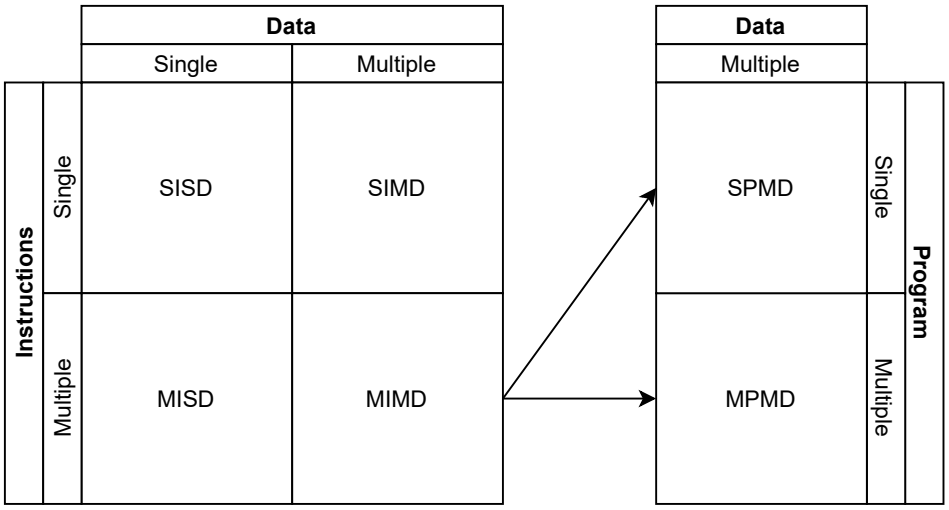


Figure 3.5: Taxonomy of parallelism in hardware

Single core CPU designs operate in SISD fashion: a single instruction operates on a single piece of data at a time. For example, we can refer back to listing 3.1: a value is copied from RAM into a specific register. Now the ADD assembly instruction operates on that single register. Adding a value to another variable stored in a register mandates a second ADD instruction.

Multithreading using multiple hardware threads (possibly backed by multiple CPU cores) belongs into the SIMD category. Apart from SISD operation, modern CPU cores often feature instruction set extensions such as Intel SSE/AVX or ARM NEON to enable SIMD operations. A single instruction can be applied to multiple data pieces in lockstep. Most accelerators work in the same way: Nvidia CUDA cores are consolidated into so-called

[6] `https://www.Nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3080/`

Streaming Multiprocessors (SMs), which share cache lines between the cores they control and operate in SIMD fashion. The same generally applies to Intel and AMD GPUs, although the terminology differs.

The MISD modus operandi, where multiple instructions are applied to a single piece of input data, is highly uncommon. One example where it is applied is avionics computers where multiple machines perform the same computation and must agree on the result, otherwise the result is faulty. This approach was used in the NASA Space Shuttle flight controller [45].

MIMD architectures include distributed systems or multi-core superscalar systems. A superscalar processor can execute multiple instructions during each cycle [20], akin to the fused multiply-add (FMA) instructions found in consumer CPUs, only for generic instructions. They are commonly found in supercomputers such as Cray systems. Seymour Cray's CDC 6600 is regarded one of the first superscalar processor designs. MIMD can be divided into single program, multiple data (SPMD) and multiple program, multiple data (MPMD) as shown in Figure 3.5.

3.3.3 Memory Hierarchy

Modern computer systems have access to an array of different memory stores. To an application developer, only random access memory (RAM) may be known, since it stores the program code and any variables created during the lifetime of the program. But the memory hierarchy in modern CPUs is actually more complex than that. High performance applications mandate good CPU resource saturation and minimizing stall conditions. When thinking about CPU stall conditions, many software engineers only have input/output (I/O) work in mind, where the CPU has to waste cycles idling. This is often caused by data transfers from external storage such as SATA disks into main memory, where the CPU can access it. However, many of the precious CPU cycles are wasted by cache trashing. That kind of trashing occurs e.g. when a CPU core discards a cache line to make place for other content, but needs the old (and discarded) cache contents again in another iteration in a hot loop.

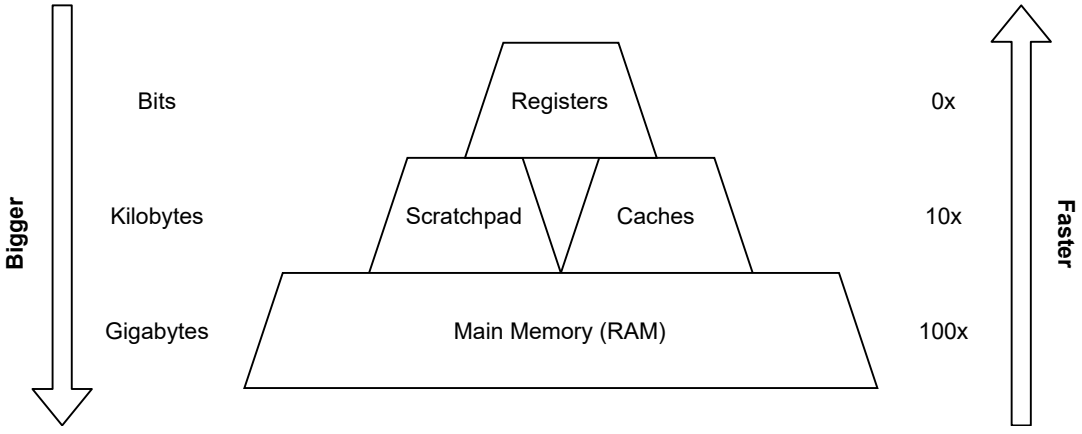


Figure 3.6: CPU memory hierarchy

Figure 3.6 illustrates the memory hierarchy based on the description by Brodtkorb et al. [5]. According to them, the smallest memory regions are the CPU registers, which are usually 64 bits large nowadays. They operate at the same speed as the CPU itself, meaning there is a zero clock cycle latency when accessing them.

Next in the hierarchy are scratchpad memories, also called local store memory, and caches. While caches provide the hardware logic to move data in and out of them, scratchpad memory has to be managed externally. Steinke et al. mention scratchpad memory to be more energy efficient, saving area and energy on a chip usually occupied by cache control logic [47]. Modern mainstream CPUs do not usually come with scratchpad memory. It is more commonly found in the embedded world, where chips need to be as small and cheap as possible. Both scratchpad memory and caches are usually sized in the kilobyte region, although high level caches such as L3 caches can be sized in megabytes, too. There is a notable latency penalty of tens of cycles when accessing this kind of memory.

In terms of size, the biggest memory available to the CPU is the main memory, also called RAM. This is where user-space programs are stored on execution along with any variables and other program data. Modern systems often come with gigabytes or even terabytes of RAM. The access latency is much higher than with any other kind of memory, being in the hundreds of cycles.

So far, we discussed faster, but smaller and slower, but bigger types of memory. Earlier in this chapter we highlighted the importance of knowing about caches for software engineers. We believe the issue of cache trashing and its performance implications are best illustrated by an example: matrix transposition. Imagine a matrix A, with dimensions mxn. The transpose operation turns A into a new matrix B with dimensions nxm as shown in equation 3.2.

$$A_{ij} => B_{ji} \forall i \in \{1, .., m\}, j \in \{1, .., n\} \tag{3.2}$$

Tsifakis et al. [49] compare a *cache-ignorant* against a *cache-blocking* and a *cache-oblivious* transpose algorithm. *Cache-blocking* means that the matrix is split up into smaller "blocks" which fit into the cache. This kind of approach requires the optimal block size to be determined a-priori. The term *cache-oblivious* refers to a class of algorithms which do not specifically target a hardware architecture, but tune themselves to achieve optimal cache exploitation at runtime. This is usually achieved by recursion and probing whether the current data block wholly fits into the cache. The authors restricted their analysis to square matrices which are transposed "in-place", meaning there is only one memory allocation to hold the source matrix A and the destination matrix B. Thus, the same restrictions apply here.

Figure 3.7 illustrates the *cache-ignorant* matrix transpose algorithm and shows the first iteration. The cache exploitation of this algorithm is poor because cache lines are being trashed and reloaded during the swap operation. The inner loop that increases j causes discontiguous jumps between memory addresses during the *swap* operation. For example, imagine a primitive memory model where the first matrix element is stored at address $0x00$. Each cell of the matrix can hold a 32 bit integer value, so a $3x3$ matrix requires $3 * 3 * 32 = 288$ bits or 36 bytes of memory. At some point, the algorithm will swap the element in the top right hand corner of the matrix, A(1,3) with the bottom left hand one, A(3,1). A(1,3) resides at memory address $0x08$ (because the first is at $0x00$, the second at $0x04$, etc), but A(3,1) resides at $0x18$. To perform the swap operation in cache, the entire range from $0x08$ to $0x18$ has to be loaded into the cache, which is five matrix cells or 20 bytes. But for larger matrices of $100x100$ or even $1000x1000$, the fastest caches of modern CPUs are not large enough, leading to a huge number of cache misses.

Tsifakis et al. [49] found that the *cache-blocking* and *cache-oblivious* matrix transpose algorithms could more than halve the cache misses of the naive algorithm, sometimes

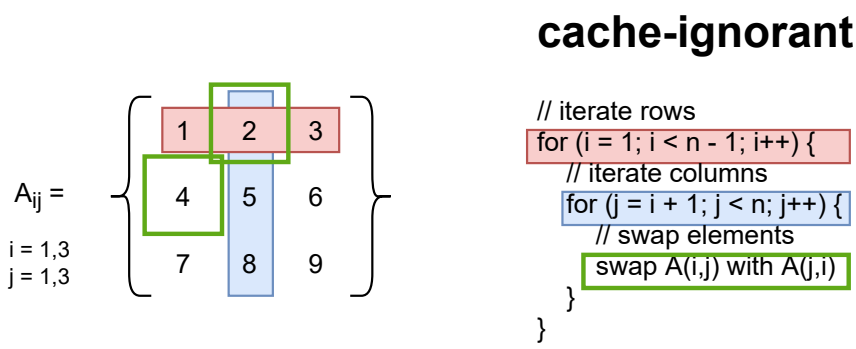


Figure 3.7: Naive matrix transposition

yielding improvements of orders of magnitude. Curiously, the oblivious algorithm actually yielded more cache misses than the other two for very large input matrix sizes. According to the authors, this is caused by an interplay between matrix dimension, cache size and others. They state knowing a-priori which algorithm performs best for a matrix of unknown size is non-trivial. Ultimately, the authors confirmed the previous work of Chatterjee and Sen [7], who arrived at similar conclusions.

Since modern CPUs often feature multiple cores integrated on a chip and sharing at least some larger caches, *cache-aware* programming is becoming more important. Advances in processor technology may hide cache trashing issues by other speedups, but especially constrained environments such as embedded devices require careful algorithm design. As more and more cores are integrated on a single die, the issue only becomes more apparent: one core may trash a cache line, requiring the next core to load it again. This can lead to inter-core cache trashing, which is especially harmful in multithreaded applications, where multiple cores work on the same piece of data.

3.4 Programming Stacks

In the previous section, we touched on heterogenous architectures in hardware. We explained how parallelism, special memory hierarchies and offloading can result in huge performance gains when using accelerators compared to just CPUs. Perhaps the most apparent issue in employing accelerators today is their complex programming procedure. A naive implementation of an algorithm on an accelerator may not perform better than a CPU at all. Achieving the maximum performance and full hardware exploitation usually requires vast domain knowledge and system engineering capabilities. With the vast increase in machine learning and deep learning applications in the recent years, a new breed of frameworks has emerged, allowing for easy programming even by novices. Since domain experts designing the algorithms are usually not expert programmers, this is of utmost importance.

In this section, we present a selection of currently available programming stacks for heterogenous architectures and accelerators. The stacks we chose based on popularity are CUDA, OpenCL and TensorFlow. We add examples illustrating the usage of each stack and compare the cognitive complexity. Finally, we talk about challenges faced when programming accelerators in general, such as tooling environments, debuggers and more.

3.4.1 CUDA

CUDA is a proprietary programming stack and API created by Nvidia for their GPUs, initially released in 2007 according to Wikipedia [7]. The latest stable release is CUDA 11, which was published in 2020. Since its inception, CUDA has gained support for Nvidia consumer GPUs such as the GTX series as well as their enterprise offering, the Tesla series. Apart from the main CUDA stack, an entire ecosystem around it was created with specialized libraries and tools [8]. This includes generic math libraries as well as some spanning specific use-cases such as cuDNN for convolutional neural networks or cuFFT for accelerated fast-fourier transforms. The tool collection encompasses debuggers, performance profilers and more.

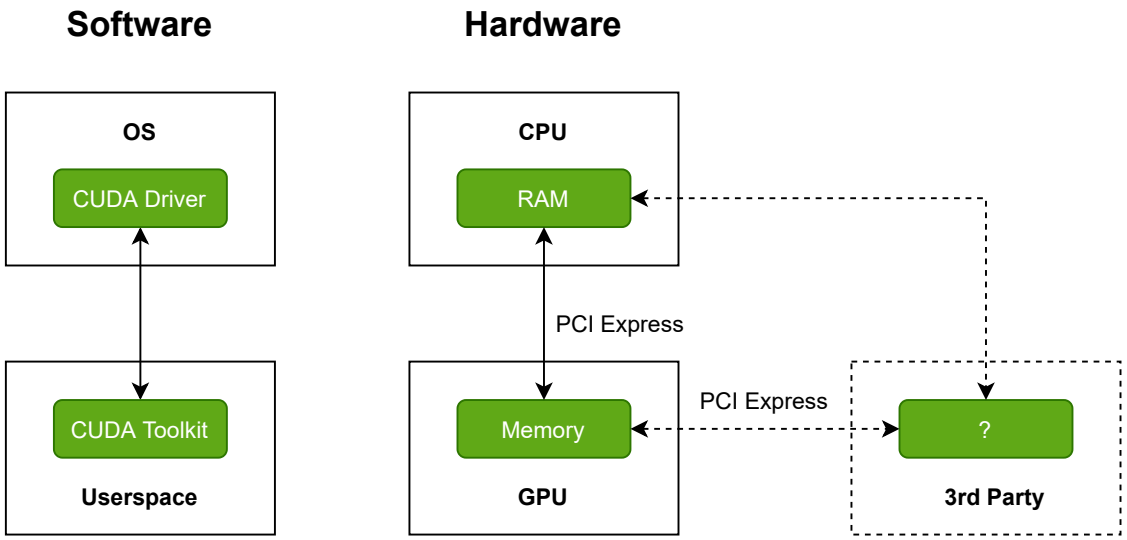


Figure 3.8: CUDA architecture

Figure 3.8 shows the rough CUDA architecture. On the left side, the software stack is divided into the OS specific part and the user-space part. The former consists of the GPU driver while the latter encompasses the ecosystem mentioned earlier in this section. On Linux for example, Nvidia provides a proprietary kernel driver which is not only necessary for hardware accelerated 2D and 3D operations, but CUDA as well. On the right side, the hardware architecture enables one or more GPUs connected to the CPU via PCI Express. The CPU can access the GPU memory or share RAM areas with the GPU for shared memory access. On enterprise class cards such as Nvidia Tesla series models, advanced features such as *RDMA* can be used. RDMA means *remote direct memory access* and describes a mechanism for direct device-to-device communication between the GPU and other third-party PCI Express devices.

CUDA API bindings are available for many general purpose programming languages such as C, C++ or Fortran. Most developers who get into GPU programming will already know one of these languages, easing the transition and flattening the learning curve. CUDA also incorporates various features from the respective languages, e.g. templates from C++.

A CUDA program usually consists of at least the following basic steps:

1. Allocate host and device memory

[7] `https://en.wikipedia.org/wiki/CUDA`
[8] `https://developer.Nvidia.com/cuda-zone`

2. Fill host memory, copy to device memory
3. Run kernel
4. Copy device memory contents into host memory
5. **Process**
6. Release device and host memory

First, memory is allocated to hold the data structures the GPU will operate on. Next, the memory is initialized and filled with data. The input and output memory regions are now copied into GPU memory. Finally, the GPU runs a *kernel*, which is the algorithm that we wanted to run on the input data in the first place. The resulting output memory area is copied back into host memory, where it can now be processed. At the same time, the memory areas on the GPU device can be freed, since the results have already been transferred. Once the processing is done, the host memory regions can be released as well.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// CUDA kernel. Each thread takes care of one element of c
___global___ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}

int main( int argc, char* argv[] )
{
    // Size of vectors
    int n = 100000;

    // Host input vectors
    double *h_a;
    double *h_b;
    //Host output vector
    double *h_c;

    // Device input vectors
    double *d_a;
    double *d_b;
    //Device output vector
    double *d_c;

    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);

    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);

    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
```

```
44      cudaMalloc(&d_c, bytes);
45
46      int i;
47      // Initialize vectors on host
48      for( i = 0; i < n; i++ ) {
49          h_a[i] = sin(i)*sin(i);
50          h_b[i] = cos(i)*cos(i);
51      }
52
53      // Copy host vectors to device
54      cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
55      cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);
56
57      int blockSize, gridSize;
58
59      // Number of threads in each thread block
60      blockSize = 1024;
61
62      // Number of thread blocks in grid
63      gridSize = (int) ceil ((float)n/blockSize);
64
65      // Execute the kernel
66      vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
67
68      // Copy array back to host
69      cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );
70
71      // Sum up vector c and print result divided by n, this should equal 1 within error
72      double sum = 0;
73      for(i=0; i<n; i++)
74          sum += h_c[i];
75      printf("final result: %f\n", sum/n);
76
77      // Release device memory
78      cudaFree(d_a);
79      cudaFree(d_b);
80      cudaFree(d_c);
81
82      // Release host memory
83      free(h_a);
84      free(h_b);
85      free(h_c);
86
87      return 0;
88  }
```

Listing 3.3: CUDA vector addition

Listing 3.3 implements a simple vector addition operation in CUDA, taken from the Oak Ridge National Laboratory CUDA tutorial [9]. The first thing to note is that the heterogenous application consists of a single source file. It is compiled by the Nvidia compiler (nvcc), which extracts the CUDA parts and compiles them to *PTX* (GPU machine code instructions). The rest of the program code is then forwarded to a C/C++ compiler that compiles the host part of the program.

In the Nvidia GPU architecture, each processor core runs a so-called *thread*. Multiple threads form a block, where the threads share block-level memory. The block size is set in line 60 of the example code. It is usually empirically tuned to achieve optimal *hardware occupancy* to hide the GPU memory transfer latencies, but has been the topic of ongoing

[9] `https://www.olcf.ornl.gov/tutorials/cuda-vector-addition/`

research such as the one by Wong et al. [50]. The most interesting part of the example is the CUDA kernel itself in lines 5-14. While the goal is to add one vector to another, CUDA kernels usually require safe guards to ensure that threads (CUDA cores) do not work on invalid data. To achieve this, we compute the global thread ID in line 9 and check that it is smaller than the total number of vector elements, n. If it was not, the thread would be operating on data beyond the input vectors. This kind of safe guarding is paramount to CUDA kernels because we usually do not spawn exactly as many threads or thread blocks as the task requires. Instead, we attempt to estimate the optimal thread count and have some threads running idle.

As can be seen from the CUDA architecture and the accompanying code example, writing heterogenous applications for Nvidia accelerators is a complex task even after years of ecosystem evolution. It still requires expert domain knowledge in systems programming along with empirical tuning to achieve the best resource exploitation. In the next section, we take a look at a competing heterogenous programming API, OpenCL.

3.4.2 OpenCL

In the previous section, we introduced CUDA as the de-facto programming API for Nvidia GPUs. OpenCL is a competing standard initially developed by Apple and later submitted to the Khronos Group in cooperation with industry partners such as IBM, Intel, Nvidia and others [10]. In contrast to proprietary GPGPU solutions, OpenCL was designed to run on other kind of hardware as well, for example FPGAs and DSPs. This marks an important distinction from Nvidia's CUDA, which solely targets GPUs.

The initial release of the OpenCL 1.0 API specification happened in 2009 and the latest version, 3.0, was released in 2020. From the active development of the OpenCL specification and vendor implementations, it is obvious that industry interest in a common compute API is still high. While Nvidia used to support OpenCL in parallel to CUDA with their drivers, they still only provide OpenCL 1.2 implementations on their latest GPUs. AMD, another big vendor of GPUs for consumer and enterprise class hardware, officially supports OpenCL 2.2 as part of their ROCm compute framework.

As stated earlier in this section, OpenCL does not focus on GPUs exclusively, but is meant to run on a wide range of accelerators including FPGAs and DSPs. Naturally, one might expect the API to be more convoluted and cumbersome than CUDA. The basic program scheme is similar to CUDA as listed in the CUDA steps: allocate and initialize the memory, run a kernel, read back the results and process them. Remember to release the host and device memory afterwards.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <CL/opencl.h>
5
6  // OpenCL kernel. Each work item takes care of one element of c
7  const char *kernelSource =                                      "\n" \
8  "#pragma OPENCL EXTENSION cl_khr_fp64 : enable                  \n" \
9  "__kernel void vecAdd( __global double *a,                      \n" \
10 "                    __global double *b,                      \n" \
11 "                    __global double *c,                      \n" \
12 "                    const unsigned int n)                    \n" \
13 "{                                                            \n" \
14 "    //Get our global thread ID                                \n" \
15 "    int id = get_global_id(0);                               \n" \
```

[10] https://en.wikipedia.org/wiki/OpenCL

```
16 "                                                                    \n" \
17 "     //Make sure we do not go out of bounds                         \n" \
18 "     if (id < n)                                                    \n" \
19 "         c[id] = a[id] + b[id];                                     \n" \
20 "}                                                                   \n" \
21                                                                     "\n" ;
22
23 int main( int argc, char* argv[] )
24 {
25     // Length of vectors
26     unsigned int n = 100000;
27
28     // Host input vectors
29     double *h_a;
30     double *h_b;
31     // Host output vector
32     double *h_c;
33
34     // Device input buffers
35     cl_mem d_a;
36     cl_mem d_b;
37     // Device output buffer
38     cl_mem d_c;
39
40     cl_platform_id cpPlatform;        // OpenCL platform
41     cl_device_id device_id;           // device ID
42     cl_context context;               // context
43     cl_command_queue queue;           // command queue
44     cl_program program;               // program
45     cl_kernel kernel;                 // kernel
46
47     // Size, in bytes, of each vector
48     size_t bytes = n*sizeof(double);
49
50     // Allocate memory for each vector on host
51     h_a = (double*)malloc(bytes);
52     h_b = (double*)malloc(bytes);
53     h_c = (double*)malloc(bytes);
54
55     // Initialize vectors on host
56     int i;
57     for( i = 0; i < n; i++ )
58     {
59         h_a[i] = sinf(i)*sinf(i);
60         h_b[i] = cosf(i)*cosf(i);
61     }
62
63     size_t globalSize, localSize;
64     cl_int err;
65
66     // Number of work items in each local work group
67     localSize = 64;
68
69     // Number of total work items - localSize must be devisor
70     globalSize = ceil(n/(float)localSize)*localSize;
71
72     // Bind to platform
73     err = clGetPlatformIDs(1, &cpPlatform, NULL);
74
75     // Get ID for the device
76     err = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
77
```

```
78      // Create a context
79      context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
80
81      // Create a command queue
82      queue = clCreateCommandQueue(context, device_id, 0, &err);
83
84      // Create the compute program from the source buffer
85      program = clCreateProgramWithSource(context, 1,
86                                  (const char **) & kernelSource, NULL, &err);
87
88      // Build the program executable
89      clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
90
91      // Create the compute kernel in the program we wish to run
92      kernel = clCreateKernel(program, "vecAdd", &err);
93
94      // Create the input and output arrays in device memory for our calculation
95      d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
96      d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
97      d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bytes, NULL, NULL);
98
99      // Write our data set into the input array in device memory
100     err = clEnqueueWriteBuffer(queue, d_a, CL_TRUE, 0,
101                                       bytes, h_a, 0, NULL, NULL);
102     err |= clEnqueueWriteBuffer(queue, d_b, CL_TRUE, 0,
103                                       bytes, h_b, 0, NULL, NULL);
104
105     // Set the arguments to our compute kernel
106     err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
107     err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
108     err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
109     err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &n);
110
111     // Execute the kernel over the entire range of the data set
112     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize, &localSize,
113                                                              0, NULL, NULL);
114
115     // Wait for the command queue to get serviced before reading back results
116     clFinish(queue);
117
118     // Read the results from the device
119     clEnqueueReadBuffer(queue, d_c, CL_TRUE, 0,
120                                    bytes, h_c, 0, NULL, NULL );
121
122     //Sum up vector c and print result divided by n, this should equal 1 within error
123     double sum = 0;
124     for(i=0; i<n; i++)
125         sum += h_c[i];
126     printf(" final result: %f\n", sum/n);
127
128     // release OpenCL resources
129     clReleaseMemObject(d_a);
130     clReleaseMemObject(d_b);
131     clReleaseMemObject(d_c);
132     clReleaseProgram(program);
133     clReleaseKernel(kernel);
134     clReleaseCommandQueue(queue);
135     clReleaseContext(context);
136
137     //release host memory
138     free(h_a);
139     free(h_b);
```

```
140     free(h_c);
141
142     return 0;
143 }
```

Listing 3.4: OpenCL vector addition

Listing 3.4 implements a simple vector addition operation in OpenCL, again taken from the Oak Ridge National Laboratory tutorial [11]. Large parts of the program are similar to the CUDA example (3.3), such as the host buffer initialization. In fact, the kernel code in lines 6-20 looks almost identical to the CUDA kernel code. The most obvious and curious difference is that the CUDA kernel can be written in the embodying programming language (C/C++ in this case), whereas the OpenCL kernel is a mere string. The string is compiled by the OpenCL driver at runtime (line 92). This is the reason why OpenCL enabled programs usually take a longer time to start up - kernel compilation happens on first use. From the example code, we can see that the device initialization is quite a bit more convoluted than the CUDA equivalent: it encompasses explicit context initialization and command queue setup. Just like in CUDA, we need to define a fixed *local* and *global* thread size and pass those parameters to the kernel launch command in line 112.

Compared to CUDA, where the buffer transfers using *cudaMemcpy* look almost like regular C *memcpy* calls, the OpenCL API is even more evolved and requires expert domain knowledge. On the other hand, it defines and implements a uniform memory model for all kinds of accelerator devices, something not many standards have accomplished. An OpenCL application written for GPU hardware should theoretically be portable to FPGA hardware using the same code (barring parameter differences such as local and global work group sizes). Compared to the other programming APIs (VHDL, Maxeler MaxJ etc.) used by DSP and FPGA vendors such as Xilinx or Altera, OpenCL seems to be the preferable approach for reusable and maintainable code.

3.4.3 TensorFlow

We introduced CUDA and OpenCL as *low level* programming APIs for accelerators in the previous sections. In this section, we present TensorFlow as an example for a *high level* heterogenous computing framework. Technically, TensorFlow is not an API for talking to accelerator hardware. Instead, it provides multiple levels of abstraction over the execution backends (such as CUDA, OpenCL and more) it can leverage. Thus, it aims to reduce the complexity of heterogenous programming with a focus on machine learning and neural networks.

TensorFlow is developed by Google and was initially released in 2015. The latest stable release, 2.4.1, was announced in 2021 [12]. The framework is written in Python and C++, as are its API bindings. It should be noted that while C++ bindings are available, the TensorFlow community almost exclusively uses the Python bindings, leading to much better documentation and testing.

```
1 import math
2 import numpy as np
3 import tensorflow as tf
4
5 n = 100000
6
7 # allocate memory for each vector
```

[11] https://www.olcf.ornl.gov/tutorials/opencl-vector-addition/
[12] https://en.wikipedia.org/wiki/TensorFlow

```
8  h_a = np.empty(n, dtype=np.float64)
9  h_b = np.empty(n, dtype=np.float64)
10
11 # initialize the vectors
12 for i in range(0, n):
13     h_a[i] = math.sin(i) * math.sin(i)
14     h_b[i] = math.cos(i) * math.cos(i)
15
16 # compute the elementwise addition
17 h_a_tf = tf.convert_to_tensor(h_a, dtype=np.float64)
18 h_b_tf = tf.convert_to_tensor(h_b, dtype=np.float64)
19 h_c_tf = tf.math.add(h_a_tf, h_b_tf)
20 h_c = h_c_tf.numpy()
21
22 # sum up the result vector
23 c_sum = 0
24 for elem in h_c:
25     c_sum += elem
26
27 print("final result: {}".format(c_sum / n))
```

Listing 3.5: TensorFlow vector addition

Listing 3.5 is the previous vector addition example program from the Oakridge National Laboratories, converted to TensorFlow. As can be seen from the code, implementing the task at hand is much easier than in CUDA or OpenCL. The high-level abstractions of TensorFlow (and the Python language) allow us not to worry about memory allocation and release. At the same time, we lose much of the control over memory placement and movement semantics. On the host side, we allocate the two arrays h_a **and** h_b as *numpy* arrays (lines 8-9), which are just standard Python constructs. The output array, h_c, is allocated implicitly in line 19. Once the actual vector operation happens, we explicitly convert them into so called *Tensors*, the main TensorFlow data type. A Tensor has a datatype (integer, float, etc.) and a shape. The shape determines its dimensionality: a Tensor can be an 1D array like in this example, a 2D matrix, a 3D body or even a n dimensional entity.

Perhaps the most obvious simplification introduced by TensorFlow is the *kernel* execution, which boils down to a single statement in line 19. Programmers do not need to pay attention to bounds checking anymore, as was the case with the CUDA and OpenCL examples. Instead, the TensorFlow execution backend, which is implemented in CUDA or OpenCL (or others in the future) handles this transparently. We can conclude that TensorFlow enables scientists and industry professionals without domain expertise to write heterogenous programs to a certain degree. In this sense, it is no different from traditional software libraries and frameworks that provide high-level bindings to make complex functionality accessible to non-experts. There have been numerous efforts porting these high-level frameworks to FPGAs and other accelerators, such as the work done by DiCecco et al. [12].

3.5 Challenges

Programming heterogenous systems is a challenge even with the frameworks and abstractions of today. Part of that is related to the silicon vendors not agreeing on common APIs. For example, Nvidia still treats OpenCL as a second-class citizen on their hardware, providing only OpenCL 1.2 support even on the latest hardware. Nvidia customers have been

demanding OpenCL 2.0 support for years to no avail [13]. AMD's high performance computing stack, ROCm, along with their OpenCL stack is not as mature to compete with CUDA just yet. Even worse, many recent consumer grade GPUs are not supported by AMD's latest ROCm release [14]. Nvidia has traditionally been more reliable in providing timely CUDA support on their cards. Intel on the other hand is providing good OpenCL support for their integrated GPUs and the upcoming dedicated Xe series GPUs. On top of that, they develop their own high-level abstractions as part of the OneAPI initiative. Ever since Intel bought out Altera, their FPGA products support OpenCL to a certain degree, finally enabling truly portable heterogenous applications.

Apart from the software stack challenges, the hardware itself can be challenging to support as well. For example, many FPGAs and embedded accelerators such as TPUs only support integer operation modes. If one attempts to run algorithms on floating point data, the accelerator will reject the data, resulting in a crash at worst and punting to CPU computation at best. Performance consistency is another pain point, often critical to latency guarantees and service level agreements (SLAs). Since most accelerator designs such as GPUs or TPUs adopted CPU power-saving features like frequency scaling, they are prone to load spike issues. On the other hand, FPGAs usually run with a fixed clock frequency, making runtime performance deterministic.

The industry faces yet another challenge. When designing a product, what kind of accelerator fits the use-case best? Building prototypes for each variant, porting the algorithm and optimizing it so as to be able to run comparative benchmarks is often not an option for monetary and time constraint reasons. Thinking in terms of product families, is the accelerator and the code reusable for the next product? Many accelerators are purpose-designed and not easily integratabtle into a new product. What if the data type that needs to be processed changes, but the existing accelerator cannot cope with it? Uncertainties like these prevent various industries from using accelerators in their hardware systems today. Andrade et al. report additional non-functional challenges such as energy consumption or overheating on top of software and hardware complexity [1]. Furthermore, they report a vast need for domain specific expertise to successfully reach heterogenous platform adoption.

[13] `https://forums.developer.nvidia.com/t/opencl-2-x-support-plans/44215`
[14] `https://github.com/RadeonOpenCompute/ROCm`

4 Neural Networks

In this chapter, we present an overview on neural networks in general. In doing so, we touch on classic architectures as well as more recent state of the art developments. In Section 4.1, we introduce the biological neuron and its artificial representation. Building on that, we present basic concepts of artificial neural networks in Section 4.2. We go on to motivate the use of convolutional neural networks for our heterogenous computing benchmark experiments in Section 4.3. Finally, we present a state-of-the-art convolutional neural network for object detection, YOLO, which is used in our benchmark suite. We chose YOLO for its popularity within the deep learning community and its outstanding performance compared to other object detection network architectures.

4.1 The Neuron

The neuron is the fundamental building block of neural networks. In this section, we present the basic model that can be used to describe a neuron through the course of this thesis. Furthermore, we take a look at the origin of that model, looking at the human brain. Finally, we present a mathematical model of the neuron as described by other authors during the last few decades. In doing so, we also discuss and motivate so called *activation functions* which enable neural networks to solve the complex, non-linear problems of today.

4.1.1 Biological Inspiration

All artificial neural networks are modeled after the neurons in the human brain. Thus, it is vital to understand the way they work to be able to reason about their importance in computer systems.

Figure 4.1 shows a biological neuron. The following description is based on Jain et al [19]. Apart from its cell body (also called *soma*), a neuron sports two types of connectors for communication with other parties: *dendrites* and *axon*. One may describe the neuron using an information-flow schema: signals are received as small electrical impulses from other neurons through the *dendrites*. In the second step, they are processed by the cell body. Finally, the output signal is emitted through the cells' *axon*. That axon will branch out into smaller strands which in the end are connected to synapses, which in turn are again connected to the *dendrites* of other neurons. Thus, a one-to-many connection scheme can be established.

According to Boers and Kuiper [4], the human brain sports around 10^{11} neurons. If all of these neurons were interconnected with each other, that would result in a human head with a diameter of more than ten kilometers. Curiously, only relatively few connections exist between the functional areas of the human brain like the motor area, the auditory system or the visual cortex [4]. Thus, the brain appears to be rather modularized instead. In [19], Jain et al mention the so called *hundred step rule*. Apparently, the duration of message transmission between neurons is in the ballpark of several milliseconds, being modulated on a frequency in the range of a few up to several hundred hertz. However, even complex tasks like face recognition are done within a few hundred milliseconds by humans. Such small end-to-end latencies imply that the computations do not take more than 100 serial stages (*layers* in neural network speech). Looking purely at the computational power of the human brain, Schwartz estimates a capability of 10^{18} arithmetic operations per second, using around 10^{16} bytes of memory [43]. According to Wikipedia, some supercomputers were able to do over 10^{17} floating point operations per second [15] as of 2017. This would

[15] https://en.wikipedia.org/wiki/Supercomputer

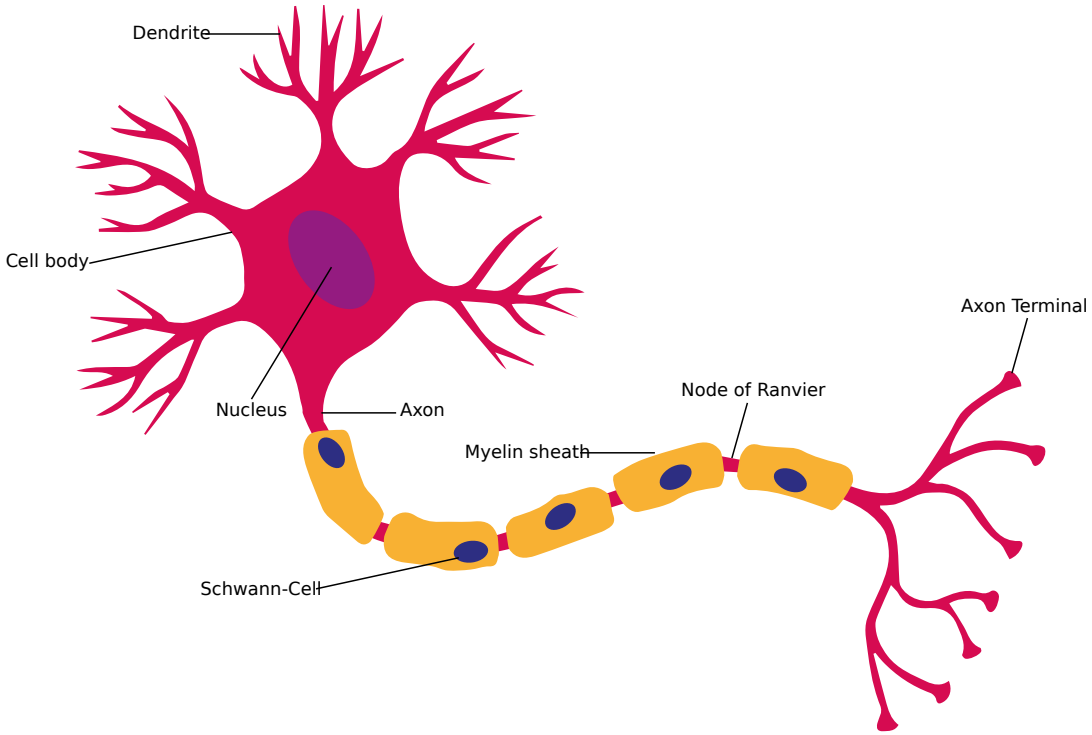


Figure 4.1: Biological Neuron

indicate we are finally able to match or even exceed the human brain in terms of raw compute power.

We know about the immense computing capabilities of the human brain from the previous section. But being able to quickly process input signals cannot make for a decisive point regarding intelligence, be it biological or artificial. Current developments in the semiconductor industry indicate we will soon surpass the human brain in terms of operations per second, as outlined at the end of the previous section. Yet many algorithms still struggle greatly at tasks that seem trivial to humans.

An example where this holds true is face detection (and recognition). There are many methods to perform efficient facial feature detection known to the author of this thesis, such as History of Oriented Gradients (HoG), Local Binary Patterns (LBP), Haar-Cascades and many more. Zafeiriou et al explore various classes of face detection algorithms in [51]. They categorize them into two categories: *rigid templates* and *deformable models*. Rigid templates describe entire objects, e.g. a car that features a chassis, two wheels and windows when viewed from the side. In contrast, part models would scan for each component individually and then conclude whether the image contains a car. Thus, a "car" like object with only one wheel could still be recognized as a car. This is a useful property with regards to object occlusion in pictures. Famous algorithms such as Viola-Jones [21] and all neural network approaches belong to the former category. The latter encompasses e.g. deformable parts models (DPMs) such as the one presented by Felzenszwalb et al [13].

Zafeiriou et al. conclude that the performance of the best competitors of the two categories is roughly equal, but the best performing approach is a hybrid one. The important question is which steps have to be performed and what effort has to be spent to achieve such levels of performance. Approaches such as convolutional neural networks (CNNs) enable us to go from only a set of pixels to full facial features by applying supervised learning (training on face image datasets). They can arrive at detecting facial features by only exploiting the spatial information contained in the image itself. Compared to deformable parts models (DPMs), where the base hypothesis is that there are deformable parts in the first place, this seems to be a much more robust approach at *generic* feature learning.

Perhaps one of the most detrimental feature of the human brain is the ability to *learn*. The way in which they do so was described by Donald Hebb in 1949 as *Hebbian Learning Law* or *Hebbian Rule* as described by Graupe [16]. It describes the relationship between two neurons in the following way: imagine three neurons A, B and C. The neurons A and B are connected to C. If neuron A is able to take part in exciting and thus firing ("activating") neuron B enough times, a metabolical process will strengthen the efficiency of neuron A. If previously neurons A and B were both needed to fire neuron B, neuron A might now be able to fire it by itself. We can think of the increase in efficiency as increase in *weight* of an edge when thinking of directed acyclic graphs (DAGs). Graupe goes on to apply the *Hebbian Learning Law* to the Pavlovian dog (experiments carried out by Pavlov in 1927): assume a dog which is excited (as seen by its salivation) at the sight of food. Now each time the dog is fed, one also rings a bell. We can think of cell C as being responsible for causing salivation. Cells A and B are excited at the the sound of the bell and sight of food respectively. The *Hebbian Rule* states that once the dog hears the bell enough times right before seeing the food, the efficiency of cell A will increase so much that eventually, the dog will salivate from the sound of the bell alone, even if there is no food.

In the next section, we will outline how machines "learn" from previous experience. The biological process of neuron activation needs to be converted into a mathematical model, so we can model stronger and weaker neuron interconnections. Just like with the Pavlovian dog, we want to *train* an artificial brain to figure out the link between one or more stimuli and an action, event or output. Researchers figured out how to translate the biological model into machine code already decades ago. However the compute power at that time was not sufficient. Nowadays, we see even low end devices such as smartphones or internet of things (IoT) hardware being equipped with accelerator chips to run neural networks.

4.1.2 Mathematical Model

After looking at the biological neuron in the human brain in Section 4.1.1, we will now take a look at the mathematical model of the same. The model presented in this section has not changed for more than two decades now, further reinforcing the close relationship to its biological counterpart.

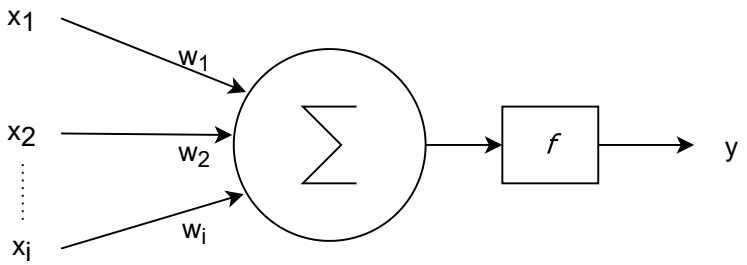


Figure 4.2: The Neuron

Given n input signals, a mathematical neuron computes the weighted sum of those signals as depicted in Figure 4.2. Here, $x = x_1, x_2, .., x_i$ is the input vector, $w = w_1, w_2, .., w_i$ is the weights vector, f is the activation function and y is the output value computed by the neuron. The diagram already shows most the actual parts of an artificial neuron. In the next step, we show a mathematical equation to describe the process in a functional way. The notation of the mathematical term tends to vary throughout the literature. For example, Jain et al. use θ as the outer activation function being applied after the sum and $-u$ as negative bias term [19].

$$y = f\left(\sum_{i=0}^{n} w_i x_i + \theta\right) \tag{4.1}$$

In our case, y is the output signal emitted by the neuron. w_i are weights used to represent the state of the neuron, for example memory gathered through past data. x_i are the input signals. $+\theta$ denotes a positive bias term being applied. Finally, f is an activation function commonly used to introduce non-linearity for complex problem solving strategies.

A more intuitive notion of the activation function is a dimmer switch as found on modern lighting solutions. The neurons' output can be increased, decreased or even turned off completely using a value of zero. Initially, artificial neurons would not gate their output using an activation function, but a simple threshold instead. These primitive neurons are also referred to as *perceptrons*. If the output of the weighted sum falls below that threshold, the output value would just be set to zero. Otherwise, it would be set to one, making it a binary decision. As Nielsen points out, this makes it very hard for networks composed of such perceptrons to actually *learn* [33]. A small change in one of the weights could flip the entire network output, which is a non-desirable property. Instead, training a neural network is usually done by showing it samples of data and associated ground truth so it can slowly adjust its weights to solve the task.

Some example activation functions as presented by Sharma et al [44] include:

- Linear activation

  $$f(x) = c * x \tag{4.2}$$

  where c is a scaling factor.

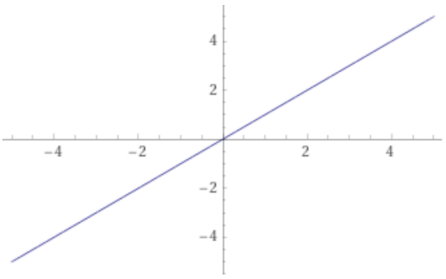


Figure 4.3: Linear Activation Function (plot by WolframAlpha)

- Sigmoid function

  $$f(x) = \frac{1}{(1 + e^{-x})} \tag{4.3}$$

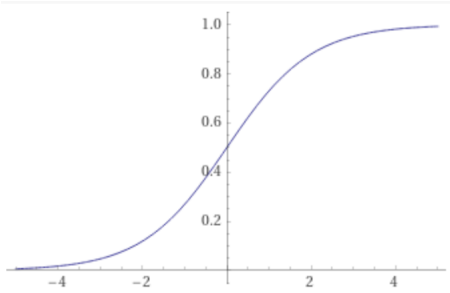


Figure 4.4: Sigmoid Activation Function (plot by WolframAlpha)

- Rectified Linear Unit (ReLU)

$$f(x) = max(0, x) \tag{4.4}$$

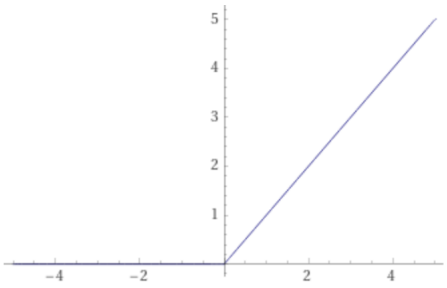


Figure 4.5: ReLU Activation Function (plot by WolframAlpha)

- Leaky ReLU

$$f(x) = \begin{cases} c * x, & \text{if } x < 0 \\ x, & \text{otherwise} \end{cases} \tag{4.5}$$

  where c is a small factor, usually 0.1 or 0.01.

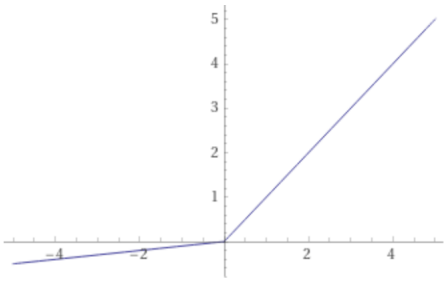


Figure 4.6: Leaky ReLU Activation Function (plot by WolframAlpha)

The list of activation functions presented in this chapter is non-exhaustive. As interest in the field of artificial neural networks from both the scientific community and the industry continues to rise, more and more activation functions are bound to surface in the future. We talk about the benefits and drawbacks of select functions for modern network architectures later in this section. We can already make various observations from the

mathematical terms alone though. The most simple function shown here is the *linear activation* function (4.2). Its sole purpose is to apply a linear scaling factor c to its input value x. One interesting property of this function is its first order derivative, since it is always a constant value: $f'(x) = c$. Next on the list is the *sigmoid* function (4.3), which has some rather interesting properties. For one, it normalizes all input values to be in the range of $[0, 1]$. Its gradient is smooth, so there are no sudden jumps in output value. For very large input values e.g. above three or or below minus three, the value of the function does not change that much anymore. Thus, the intrinsic impact of such extreme values is decreased. Compared to the *linear activation* function (4.2), the sigmoid function is more computationally expensive since it involves calculating a fraction and e^x. The third function shown here is the *rectified linear unit* (ReLU) (4.4). While it may look similar to the *linear activation* function, there are some distinct differences. Its first order derivative is not constant, which is a desirable trait for neural networks as we will see later on in this chapter. In contrast to the *sigmoid* function, it is far less computationally expensive. Since a modern neural network architecture generally involves thousands to millions of neurons, the performance of the activation function is crucial to the end-to-end network performance in terms of latency. A modified version of the *ReLU* activation function is the leaky ReLU function (4.5). While its sibling has a non-constant first order derivative, it becomes zero for any input values below zero. The *leaky ReLU* function improves on this by not capping the output values at zero. Thus its first order derivative is not constant even for inputs below zero.

Apart from decreasing or increasing the influence of the neuron with regards to the next layer in the network, the activation function also serves other purposes. First, non-linear activation functions are required to solve non-linear problems presented by the data. To illustrate this, Figure 4.7 shows a dataset containing values of two classes (red and green). The neural network is now tasked with separating the points by assigning a class label to each of them. We can easily imagine doing the task by hand: how can we best separate the data into two disjoint sets?

Figure 4.7 shows two decision boundaries side-by-side. On the left, one can see a (naive) linear separation of the data. The task of separation into two classes is fulfilled as far as basic logic is concerned: most of the red dots are on the left side of the decision boundary while most of the green dots are on the right side. Once we stop restraining ourselves to linear boundaries, we can come up with a better decision boundary though, as displayed on the right hand side of the figure. Now there is a clean separation with each data point being assigned to the set it belongs to with no outliers.

Another useful aspect of non-linear activation functions is related to the way most modern neural network models are trained. The term *backpropagation* describes a training method where the network output is compared to some ground truth and an error is computed. Based on that error value, feedback is propagated backwards through the network to readjust each neurons' weights to decrease the error metric. During this process, the derivative of the activation function needs to be computed. For each linear activation function (and some others as well) the first derivative is a constant value, rendering them unfit for this training method.

With the basic understanding of the biological and the artificial neuron, we now move on to entire neural networks in the next section.

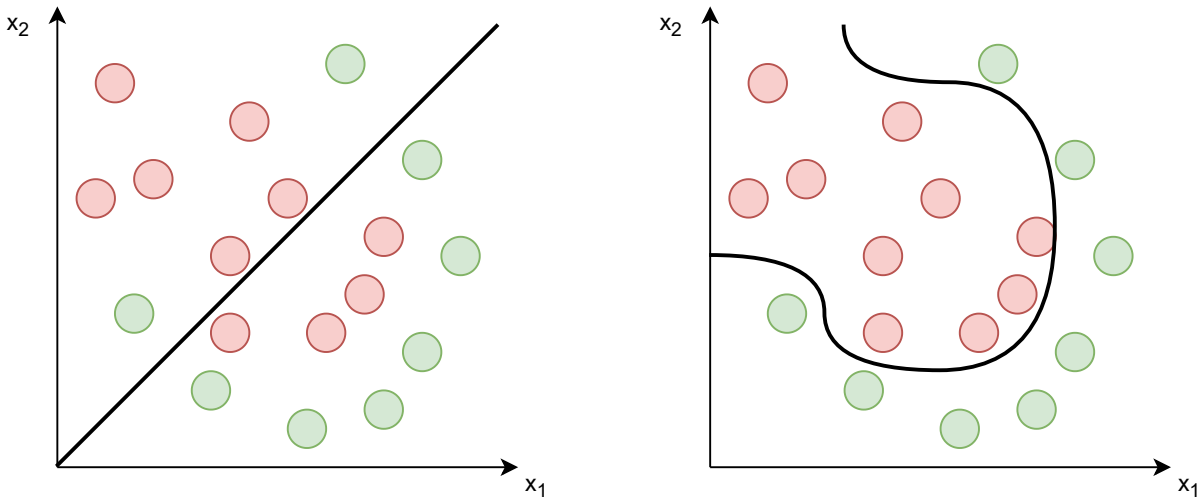


Figure 4.7: Decision boundaries: linear (left) and arbitrary (right)

4.2 Artificial Neural Networks

Jain et al. describe artificial neural networks (ANNs) as weighted digraphs where the vertices are the individual neurons and the edges are their connections [19]. One can further split up ANNs by their architecture, where the connection patterns are usually the distinguishing feature. The two main categories of ANNs as depicted in Figure 4.8 are thus:

1. feed-forward networks (graph becomes a DAG)
2. recurrent networks (with feedback loops)

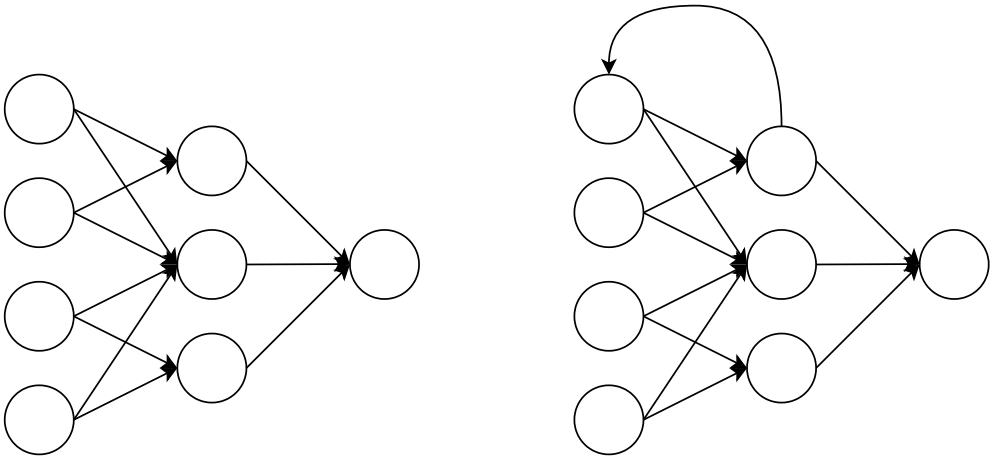


Figure 4.8: Feed-Forward (left) and Recurrent (right) networks

In the case of feed-forward neural networks, the graph is acyclic because information can only "flow" in a single direction. Thus, we get a directed acyclic graph (DAG). Since a neural network is usually made up of several layers (and possibly several hidden ones), an input stimuli will only pass each layer once and eventually end up in the output layer. This is not the case with recurrent networks. An input value x^1 may traverse a layer n with a set of neurons $n_1, n_2, .., n_i$. The computed output value y^1 is then fed to the next layer. After passing through a number of neurons in that layer, the output of that layer,

y^2 may then be passed back to the previous layer and fed to the same neurons $n_1, n_2, .., n_i$ again as input x^2. Thus, recurrent neural networks (RNNs) are dynamic systems whereas feed-forward networks are static with regards to their response [19].

Another distinction commonly made is that between *shallow* and *deep* neural networks. Nielsen describes deep neural networks as having at least two or more hidden layers. The term hidden does not really describe anything here and basically just means "neither input nor output". Figure 4.9 shows a deep neural network with input and output layers as well as three hidden layers between them. The network depicted in the figure is also a *fully connected* network, which means that each neuron in a layer is connected to all neurons in the next layer.

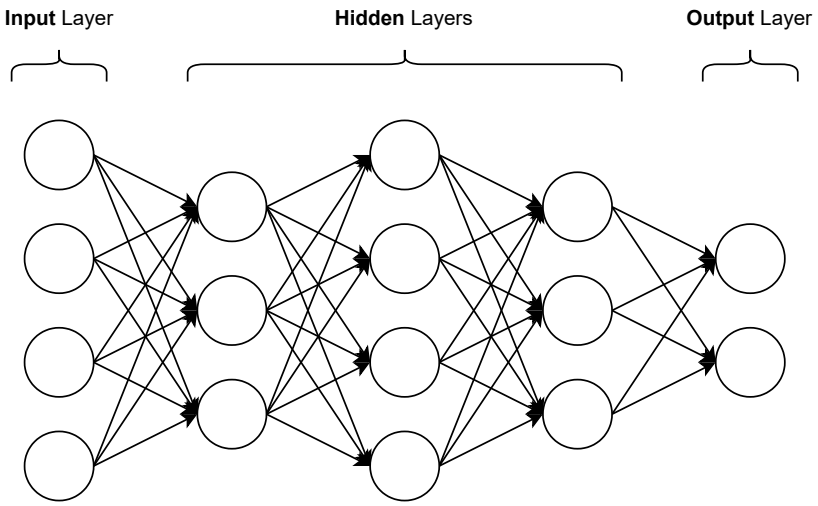


Figure 4.9: Deep Neural Network

4.3 Convolutional Neural Networks

Convolutional neural networks (CNN) are a special kind of artificial neural networks (ANN). While ANNs can be used to learn features on arbitrary data, CNNs are specifically designed for image input. Before talking about the architectural differences made to account for the input data, we want to motivate the necessity of CNNs.

O´Shea and Nash [35] show that traditional ANNs quickly struggle with computational complexity. They refer to the MNIST database, which contains images of handwritten digits saved as black and white raster images. If one was to train a standard feed-forward network (FNN), a single neuron in the first layer would contain 784 (28 x 28 x 1) weights, since each image is 28 x 28 pixels with a depth of 1 (black or white). Since 28 x 28 is a rather low resolution for a picture, we now redo the calculation for bigger images. To preserve detail, more advanced datasets may choose to store the images in color. Given e.g. 128 x 128 images with a depth of 3 (RGB channels), we arrive at 49,152 (128 x 128 x 3) weights for *each* neuron in the first layer. As can be seen from this example, the parameter count quickly explodes when using traditional ANNs for image classification or related tasks. A second reason for avoiding regular ANNs given by O´Shea and Nash is *overfitting*. Due to the high parameter count and the inherited complexity, the model is prone to to overfit to training data instead of generalizing to the underlying features contained in the data.

To overcome the issues mentioned above, CNNs do not link each part of the input data to a neuron. Instead, the spatial dimensionality of image data is exploited. The convolutional

layer, a basic building block in each CNN, connects a *patch* of the input data to a neuron. The size of the patch, which is just a local image region, can be defined as part of the network layer architecture. For example, we can define a 3 x 3 kernel size, which is then moved over the input data in a sliding window fashion. Given color input data of size 128 x 128, we arrive at 27 (3 x 3 x 3) weights for each neuron in the convolutional layer, compared to 49,152 for regular ANNs.

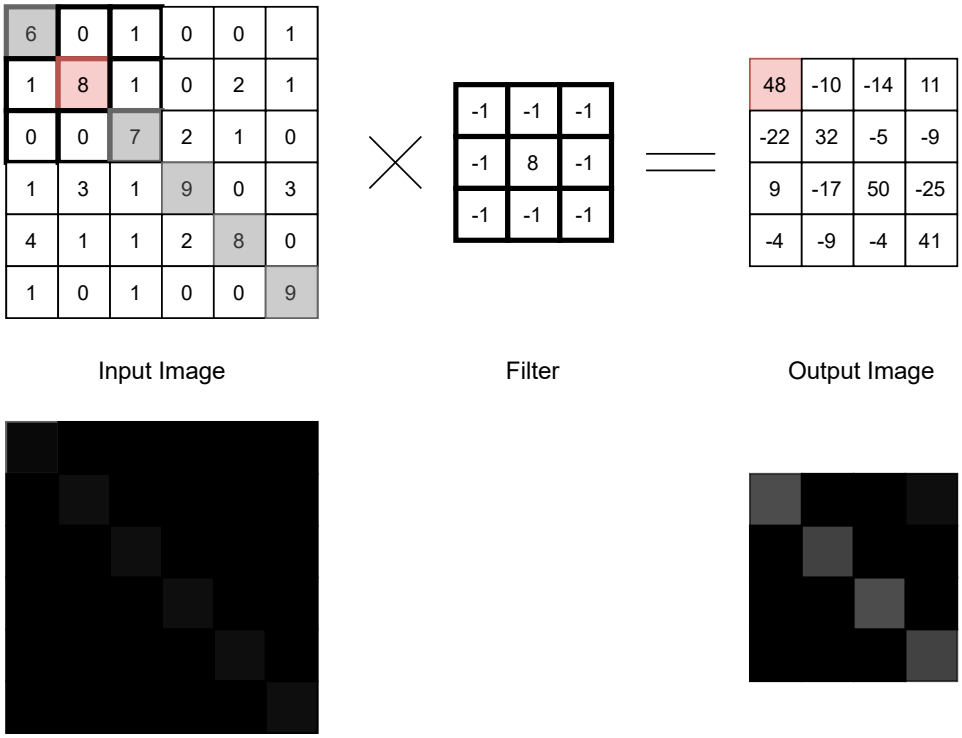


Figure 4.10: Image convolution

Figure 4.10 illustrates the generic image convolution operation. In the example, an 8-bit grayscale image is given with pixel values ranging from 0 (black) to 255 (white). Despite the small difference in pixel values, we can already recognize a pattern forming a diagonal edge. When looking at the color pixels, all we can see is a black image though. The filter is an edge detection filter of size 3 x 3. It enhances the image by increasing pixel values when there are huge disparities for neighboring pixels. To understand what the convolution operation is doing, we focus on the cell in the input image shaded in red. We lay the 3 x 3 filter on top of the input image, so that the nine cells of the filter cover nine of the input image. The center of the filter (the value 8) lies right on top of the value 8 in the input image (shaded in red). Now we compute the sum of the input values multiplied by the filter values, also called *weights*. The resulting sum is the first pixel value of the output image. For the second iteration we move the filter on to the next column of the input image, so that the center of the filter lies on top of the 1 in the input image (right of the 8 shaded in red). Moving the filter by one column or row at a time means we are using a *stride* of one.

The convolution operation is usually followed by an *activation* function that introduces non-linearity into the data. For image data especially, it will also cap the output range at a zero boundary, so all negative values are eliminated. This is what we did in the example as well, since negative pixel values are not valid for basic grayscale images. Common types of activation functions are discussed in Section 4.1.2.

As can be seen from the output image pixel colors, we can now recognize the diagonal pattern much better compared to the input image. A side-effect of the convolution operation is that the input image is shrunk in size from 6 x 6 to just 4 x 4. This happens because the filter requires a number of input data cells equal to its own cells to operate. To get around this limitation and maintain the input image size, we can use a *padding* technique. Common variants used for neural networks are *same* or *zero* padding. The former repeats each value at the edge of the input data, while the latter treats non-existing values beyond the image border as a zero.

With CNNs, the filter has no values at the start of the training process. The filter values ("weights") are either initialized with zero or assigned in a random fashion. During the training, the weights are initialized and the network learns to recognize certain patterns, for example edges like the one shown in Figure 4.10. There is another parameter which can be tweaked for convolutional layers in neural networks: the filter *depth*. It affects the number of filter outputs generated by a convolution operation. In our example, there is only one depth level for both filter and output, which detects edges. If we wanted to detect corners as well, we could use a filter of depth two to learn two features.

The next important layer in CNNs is the pooling layer. It is used to reduce the spatial dimensions ("subsampling") of filters computed by a preceding convolutional layer. The pooling operation allows us to preserve much of the information in a filter while reducing the parameter count and thus computational complexity by a huge amount.

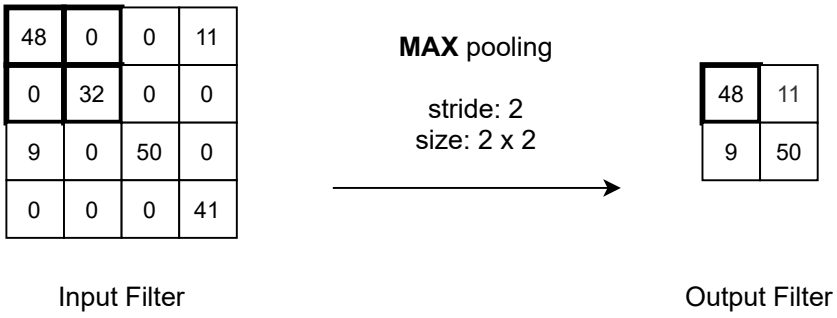


Figure 4.11: Max pooling

Figure 4.11 illustrates the max pooling operation. First, we define a pooling window by providing the *stride* and *size* parameters. The parameters have the same meaning as the ones in the convolution operation: the stride defines how many rows/columns we move the window during each iteration and the size defines how many input cells are affected during an iteration. In our example, the pooling operation is applied four times to the input image, since each iteration covers two rows/columns. Out of the four input values given to the pooling operator, it always picks the largest value and adds it to the output filter at the correct position. A window size of 2 x 2 and a stride of 2 effectively halves the input dimensions, as can be seen from the figure, leading to a 50% reduction in parameter count. The stride and size parameters usually match each other to avoid overlapping during the sliding window operation. Apart from max pooling, there are other operators such as average pooling or regularization methods.

The convolution and pooling layers are the two main defining characteristics of convolutional neural networks. Other layers such as fully-connected or dense layers are basic building blocks of ANNs in general. In the following section, we present a real-world state-of-the-art CNN: YOLO. It features several unique ideas that allow it to outperform most CNN architectures in terms of performance with regards to its (low) complexity.

4.3.1 YOLO

In 2016, Redmon et al. presented a novel approach to object detection using neuronal networks: YOLO aka "You Only Look Once" [39]. At that time, it was the fastest object detection system in terms of inference time and accuracy as measured in mean average precision (mAP) according to its authors. Furthermore, it was one of the only systems that could handle realtime video at 30 frames per second (FPS) or more. To fully understand the impact of YOLO on the computer vision community and object detection in general, we first need to take a look at previous object detection approaches.

As of 2016, most object detection approaches worked by employing two main steps [39]:

1. Region selection
2. Image Classification

Since there were many image classification systems readily available, people tried to reuse them by first searching for regions in the image which ideally contain only a single object. Once such a region has been identified, the existing image classification algorithm would then compute a class label for the selected region. This step is repeated for all regions of interest in the input image, yielding a set of bounding boxes with their corresponding class label. In case region proposal-based techniques are deemed too expensive or otherwise unfit for the task, one may resort to simple sliding window approaches instead. In this case, a hyper-parameter determines the number of windows where the classification task is run in the input image.

The issue with such kind of systems (both region proposal-based and sliding window techniques) is their complexity: one ends up splitting an input image into n sub-images and running the image classification system on each of those sub-images. On sufficiently large input images, this can easily lead to tenths or hundreds of classifications per image, which is the reason why most existing solutions could not run in real time (at least 30 FPS).

Enter YOLO. Given an input image, YOLO predicts bounding boxes and class probabilities in a single evaluation pass. This is done by dividing the image into a SxS grid, where each cell predicts B bounding boxes along with the corresponding confidence scores [39]. The process is visualized in Figure 4.12.

Apart from the bounding boxes, each cell also predicts a conditional class probability $Pr(Class_i|Object)$. Pr encodes the probability of that cell containing the specified object class, given that it contains any object at all. Thus, we can summarize the YOLO object detection pipeline in four main steps:

1. Divide image into SxS grid
2. For each grid cell, compute bounding boxes and probabilities
3. For each grid cell, compute a conditional class probability
4. If the bounding box probability exceeds a certain threshold, assign a final bounding box and the corresponding class label

In late 2016, Redmon and Farhadi presented YOLO9000 ("better, faster, stronger"), which brought numerous improvements over the previous YOLO object detection pipeline. The name derives from the fact that the authors claim to be able to detect more than 9000 object categories [40]. Some of the main improvements over YOLO are summarized below.

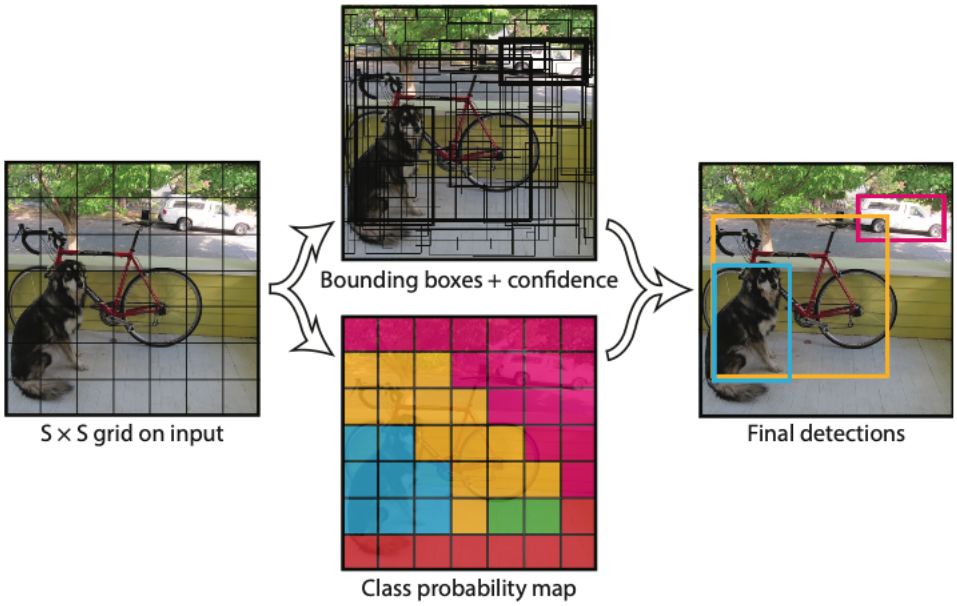


Figure 4.12: YOLO object detection taken from [39]

To avoid confusion about the naming scheme, we will refer to YOLO9000 as YOLOv2 throughout this chapter.

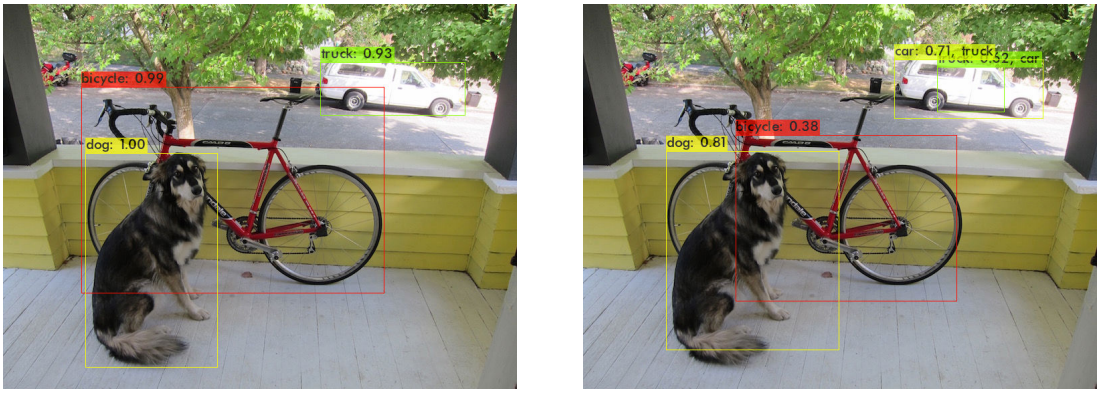
- Batch Normalization
  Added to all convolutional layers: **2% mAP** increase
- High Resolution Classifier
  Training resolution changed from 224x224 to 448x448: **4% mAP** increase
- Anchor Boxes
  Used for bounding box predictions (hand-picked initial sizes)
- Dimension Clusters
  Instead of hand-picking initial anchor boxes, use k-means clustering to find good initial dimensions based on the ground truth bounding boxes
- Direct Local Prediction
  Improved anchor box center position (x,y) prediction
- Multi-Scale Training
  Move from fixed input size (416x416) to random dimension size batch training: every 10 batches a new size is chosen, leading to an easy speed vs. accuracy tradeoff depending on input size

With all the improvements mentioned above (and some additional ones), YOLOv2 achieves a 78.6 mAP score on the PASCAL VOC2007 dataset. This marks a significant improvement over original YOLO, which achieved a score of 63.4 [40]. At the same time, YOLOv2 continues to be less complex than other architectures. For example, the base feature extractor used by most state-of-the-art frameworks is VGG-16, coming in at 30.69 billion floating point operations for a single forward pass run over a 224x224 input image. The custom, GoogleNet-based feature extractor used by YOLO only requires 8.52 billion floating point operations for a single forward pass run on an image the same size.

In early 2018, Redmon and Farhadi presented YOLOv3 [41] as an incremental improvement over YOLOv2. One of the main novelties is the base feature extractor. Where YOLOv2 used a network called *Darknet-19* consisting of 19 convolutional layers, YOLOv3 is much bigger coming in at 53 convolutional layers and thus being called *Darknet-53*. Apart from this change, Redmon and Farhadi made improvements to the bounding box prediction and scale invariancy using techniques akin to feature pyramids.

The YOLOv3 authors go on to compare the performance of the new model to its predecessor, YOLO9000 by evaluating it on the Microsoft COCO dataset. According to them, YOLOv2 scores 44.0 in the mAP-50 benchmark, while YOLOv3 achieves 55.3 on the same input size of 416x416. At this size, they report a latency of 29 milliseconds for a single forward pass on a NVIDIA Titan X GPU, making it one of the fastest object detectors as of 2018. At such low latencies, real-time image processing suddenly becomes possible, something which other famous object detectors such as RetinaNet [28] do not achieve.

The YOLO object detection frameworks are very versatile. This is in part due to their *tiny* variants, which are basically "lite" versions of the full networks. For example, the "full" YOLOv3 network uses Darknet-53 as a feature extractor, with 53 convolutional layers. YOLOv3-tiny uses just 13 layers according to the configuration file present in the code repository [16]. YOLO author Joseph Redmon presents a table on his website which compares various YOLO variants [17]. In terms of computational complexity, he full YOLOv3 network comes in at 65.86 billion floating point operations, while YOLOv3-tiny cuts that down to just 5.56 billion operations, which amounts to 8% of the full model. This of course does not come without a hit in accuracy: YOLOv3 achieves a mAP-50 score of 55.3 at an input size of 416x416, while YOLOv3-tiny achieves 33.1 for the same input size. Still, that is a 92% decrease in complexity for a 40% decrease in accuracy.



(a) YOLOv3 (b) YOLOv3-tiny

Figure 4.13: YOLOv3 *full* vs. *tiny* at 416x416

Figure 4.13 shows the difference between the *full* and the *tiny* YOLOv3 networks applied to an example image. We ran the YOLOv3 network variants in the Darknet framework developed by the YOLO authors. The input image was resized to 416x416 before running it through the network. Humans would naturally detect a dog in front of a bike in the foreground and a truck in the background. As can be seen from the left image, YOLOv3 is able to match that expectation. The right image shows that YOLOv3-tiny is pretty close as well. In the foreground, the bounding boxes for the dog and the bike are a bit off, which would contribute to a smaller intersection over union (IOU) match, which basically encodes

[16] `https://github.com/pjreddie/darknet/blob/master/cfg/yolov3-tiny.cfg`
[17] `https://pjreddie.com/darknet/yolo/`

the match between ground truth and the predicted bounding box. Thus, we can understand how the mAP score of the *tiny* network is lower than the *full* variant, although this might not actually matter much depending on the real work application. In the background, the network got confused about the truck and predicted two objects, both car and truck for the same object. Interestingly, the bounding box predicted for the truck (green) is wholly contained inside the one for the car (yellow). This means a non-maximum suppression (NMS) post-processing step with appropriate parameters would be able to suppress the inner prediction for the truck and leave us with the car.

The YOLO family of object detection networks is widely popular and delivers state-of-the-art performance while being able to retain real-time latencies on powerful accelerator hardware. Multiple authors took YOLO as a base and modified or added parts of the network to tune it for a specific purpose. For example Mao et al reduced the Darknet-53 backbone to decrease the latency in the feature extractor step [29]. They state that their "Mini-YOLOv3" network reduces the parameter size of the original YOLOv3 network by 84%. They further claim to achieve a comparable mAP-50 score of 52.1 while halving the time for a forward pass. The fact that they aim at using the network in embedded scenarios is exciting as it paves the way for efficient, yet accurate object detection on those power-constrained devices. Unfortunately they did not report the hardware used in their experiments, which is a problem we often face with scientific work in this area.

5 Approach

In this chapter, we outline our approach to answering the research questions as introduced in Section 1.2. First, we introduce various metrics to compare the system performance and power usage in Section 5.1. This includes metrics to measure benchmark performance as well as system resource usage ones. Next, we present the benchmarks we developed and show which benchmarks were run on what platform in Section 5.2. In Section 5.3, we show the tools used to get the record the metrics.

5.1 Metrics

As outlined in the introduction chapter 1, we are not only interested in raw benchmark numbers, but want to reason about the systems' resource usage as well. While classic benchmarks tend to record the CPU usage and memory pressure at most (if at all), our machine learning workloads often require external device monitoring when offloaded to accelerators. These devices often have specialized circuits such as arithmetic logic units (ALUs) or tensor cores to carry out machine learning tasks in an efficient manner. As a consequence, CPU resource usage cannot be compared to e.g. GPU usage directly. Since many device monitoring tools only report coarse metrics such as relative usage in percent anyway, this becomes an issue we are forced to ignore.

5.1.1 Latency

In benchmarking, it is common to refer to the *performance* of a system when talking about its compute capabilities. The machine learning (and deep learning) community uses the term to refer to the accuracy of an algorithm or a network though. When talking about the time it takes to complete one iteration of an algorithm (or one forward-pass in a neural network), the term *latency* is used instead.

It is common practice to batch several input values and feed them to a neural network at once during training or inference time. For the sake of comparable benchmark results, we measure the latency for a batch size of one unless stated otherwise. This ensures any specific optimizations of a framework or implementation do not come into play and skew the results.

In our experiments, we use the term latency to refer to the time it takes the system to complete one benchmark iteration. Depending on the benchmark, this can mean computing a single convolution for a whole image or running a forward pass for a neural network.

5.1.2 CPU

Since all of the test systems run on Linux, this is the only platform we need to gather monitoring data from. The Linux kernel exposes cpu usage statistics in a virtual file system in /proc [18]. Overall system metrics are contained in the virtual /proc/stat file.

```
1  cpu  167721 4918 143452 6244336 25565 20690 16441 0 0 0
2  cpu0 40612 1122 43533 1542237 10250 9179 6091 0 0 0
3  cpu1 40090 1311 38484 1560875 5598 3811 3722 0 0 0
4  cpu2 43889 778 32276 1568395 4729 3938 3486 0 0 0
5  cpu3 43128 1706 29158 1572827 4985 3761 3141 0 0 0
6  intr  ...
7  ctxt 19412198
8  btime 1610710978
9  processes 27364
10 procs_running 1
```

[18] https://linux.die.net/man/5/proc

```
11 procs_blocked 0
12 softirq  4578563 2 1548028 147174 672680 0 0 151876 1103145 0 955658
```

Listing 5.1: /proc/stat contents

Listing 5.1 shows the /proc/stat contents grabbed from the Nvidia Jetson Nano board. As can be seen from the listing, there are several other metrics reported in that file apart from CPU ones. For example, there is information on interrupts (*intr, softirq*), context switches (*ctx*) and more. Our system CPU usage monitoring task requires only the very first line, where overall system CPU statistics are provided. The per-core statistics in lines two to four are not relevant for our analysis.

Table 5.1: Linux CPU System metrics

ID	Name	Description	Unit
CS1	user	Time spent in user mode.	Hz
CS2	nice	Time spent in user mode with low priority.	Hz
CS3	sys	Time spent in system mode.	Hz
CS4	idle	Time spent in the idle task.	Hz
CS5	iowait	Time waiting for I/O to complete.	Hz
CS6	irq	Time servicing interrupts.	Hz
CS7	softirq	Time servicing softirqs.	Hz
CS8	steal	Time spent in other operating systems.	Hz
CS9	guest	Time spent running a virtual CPU for guest operating systems.	Hz
CS10	guest_nice	Time spent running a niced guest.	Hz

We list the values available from the first line of the file contents in table 5.1. In our case, we are mainly interested in the CPU resource usage caused by the benchmark task. To do so, we first need to calculate the total time the CPU was busy given a fixed time period and then calculate the fraction of that time occupied by the benchmark process. We now define some aliases in order to refer to them throughout this thesis.

$$T := T_{user} + T_{nice} + T_{sys} + T_{idle} + T_{iowait} + T_{irq} + T_{softirq} + T_{steal} + T_{guest} + T_{guest\_nice} \quad (5.1)$$

$$T_b := T - T_{idle} \quad (5.2)$$

First, we define T as the cumulated CPU time as sum of all the individual values in equation 5.1. Next, we introduce T_b as the cumulated amount of time doing work in 5.2. Throughout this work, we also refer to T_b as *CPU usage* or *busy time*.

Now that the overall cpu statistics are covered, we move on to per-process monitoring in Linux. Such process statistics are exposed by Linux again as virtual file in /proc/<PID>/stat, where <PID> is the process identifier. Several key metrics are contained in that file, but the important ones for CPU usage reporting are *utime* and *stime*.

```
1 1 (systemd) S 0 1 1 0 -1 1077936384 15724 603410 873 4247 74 262 5634 2178 20 0 1 0 2
      165072896 1088 18446744073709551615 1 1 0 0 0 0 671173123 4096 1260 0 0 0 17 1 0 0 117 0 0
       0 0 0 0 0 0 0 0
```

Listing 5.2: /proc/1/stat contents

Listing 5.2 shows the file contents for the *systemd* process with PID 1 captured on the Nvidia Jetson Nano board. We extract the numbers we need based on the indices given by Linux manuals, e.g. linux.die.net [19].

Table 5.2: Linux CPU Process metrics

ID	Name	Description	Unit
CP1	utime	Time scheduled in user mode.	Clock ticks
CP2	stime	Time scheduled in kernel mode.	Clock ticks
CP3	cutime	Time of children scheduled in user mode.	Clock ticks
CP4	cstime	Time of children scheduled in kernel mode.	Clock ticks

Table 5.2 shows the two metrics we use to monitor the CPU usage of a Linux process. We can now define additional metrics such as relative CPU usage. First, we define the sum of time spent in user mode and kernel mode in equation 5.3.

$$T_b := T_{utime} + T_{stime} + T_{cutime} + T_{cstime} \tag{5.3}$$

The key takeaways from this section are the system CPU usage T_b (as T_b^{SYS}) from equation 5.2 and the per-process usage T_b (as T_b^{PID}) from equation 5.3. The monitoring tools used to conduct the research as part of this thesis read the Linux performance counters and calculate these metrics. Again, T_b is also referred to as (per-process) *CPU usage* by us in this work.

5.1.3 RAM

Analogous to the CPU statistics, Linux exposes random access memory (RAM) stats in the virtual file system under /proc/meminfo. The values in this file describe the system state. Per-process values have to be obtained through another file, which we will describe later in this section.

```
1 MemTotal:       929988 kB
2 MemFree:         34952 kB
3 MemAvailable:   528864 kB
4 ...
```

Listing 5.3: /proc/meminfo contents

Listing 5.3 shows the first three lines of the /proc/meminfo file contents captured on a Raspberry Pi 3b+ system. We can observe that the system has a total of about 1GB of memory installed. From that 1GB, the Linux kernel reserves some bits and places the kernel image itself, which is how we end up with the number of 929988 kB (or 908 MB) for *MemTotal*. Of that total memory, only roughly 34MB are free as in *unused*, reflected by the *MemFree* entry in the file. Finally, the *MemAvailable* entry describes the amount of memory available for processes before swapping starts [20]. Swapping is the process of moving memory pages out of RAM and into (potentially) slower storage, such as persistent ones (HDD, SSD). It becomes necessary when a system runs out of RAM. Once the available RAM becomes close to zero, all processes allocating memory would just have to crash or handle the allocation failure in another way. Swapping is used to allow processes to continue allocating memory to avoid such crashing.

[19] `https://linux.die.net/man/5/proc`

[20] `https://man7.org/linux/man-pages/man5/proc.5.html`

Table 5.3: Linux RAM System metrics

ID	Name	Description	Unit
MS1	total	Total amount of useable RAM	kB
MS2	free	Amount of unused RAM	kB
MS3	available	Amount of RAM available before swapping starts	kB

We summarize the RAM stats used as part of this work in table 5.3. Based on these values, we define our RAM system usage metric M_u as shown in equation 5.4.

$$M_u := M_{total} - M_{available} \tag{5.4}$$

It is important to note that the metric M_u describes a very coarse-grained memory usage notion. We do not care about how the memory is used exactly, e.g. for user-space buffers, kernel-space drivers, or other auxiliary devices. Our aim is to provide a notion of *memory pressure*. If the M_u value is very high and starts to get close to the total amount of memory installed in the system, we are in danger of swapping and potential slowdowns. Thus, we can reason about the validity of benchmark results with the help of this metric.

Analogous to the CPU metrics, we also monitor per-process memory usage on Linux systems. This is useful so we can compare the memory usage of the benchmark process between the heterogenous platforms. Memory leaks and other implementation bugs can be detected by looking at the minimum, maximum and average memory consumption of a process with regards to the overall time series. If the memory usage of a benchmark increases monotonically, we can reason about a bug somewhere, since all our benchmarks perform the same operation thousands of times in a loop.

Table 5.4: Linux RAM Process metrics

ID	Name	Description	Unit
MP1	rss	Resident set size (physical RAM usage)	kB
MP2	vmsize	Virtual memory size (includes mapped memory regions)	kB

5.1.4 Accelerator

In contrast to CPU and RAM metrics defined in the previous sections, common metrics for accelerators are hard to think of. For one, the heterogenous coprocessor landscape makes it difficult to track down common performance properties. The hardware blocks for each accelerator chip (or lack thereof) further widen the gap between the devices. For example, graphics processing units (GPUs) usually pack their own memory, called *GDDR** or *HBM** (high bandwidth memory). Other accelerators such as FPGAs or TPUs may not feature such on-chip memory and opt for using shared main memory (RAM) instead.

We do not define specific accelerator metrics we want to measure for all systems, as the available tools vary by a great degree. Instead, we choose generic metrics, so that we can compare the accelerators against each other in the end. To facilitate this, we focus on performance and power draw, since that will allow us to reason about efficiency, performance-per-watt and more.

Table 5.5 lists the two metrics used in this work to evaluate accelerators. *Power* refers to the total power draw of the accelerator. While we would have liked to measure this for each iteration, that was not possible with most tools. On top of that, some accelerators do

Table 5.5: Accelerator metrics

Index	Key	Description	Unit
1	power	Power draw	watts
2	usage	Hardware exploitation grade	%

not expose such power usage metrics directly. The *usage* metric is similar in this regard, as it can only be acquired if there exists a tool that can read the value from the hardware. Usage measured in percent is a very generic metric that means something different for each silicon vendor. For example, most Nvidia GPUs expose a "GPU busy" metric that states how many streaming processors (CUDA cores) were busy during the last monitoring cycle. The usage metric here generally carries the same meaning as *CPU usage*, thus we refer to it as T_b as well, analogous to the CPU metric.

5.1.5 Power

Many vendors offer software tools to gather monitoring data for their chips. For example, Nvidia provides the *nvidia-smi* tool, which we describe briefly in Section 5.3.2. Unfortunately, documentation with these tools is often sparse and it is not clear what exactly the metrics refer to. Sometimes the total board power is measured, but in other cases it is only the power used by the compute cores. There is also no cross-platform interface to get power draw data among operation systems.

We decided to use external power measurement utilities in our work. After a brief research period, we decided on the Voltcraft Energy Logger 4000. The resolution of the recordings is one data point per minute. The device records the current, amperage, apparent power and true power usage. The *apparent* power metric represents the power not doing real work. It is still necessary to operate the system and will be drawn from the grid. We use the *true* power metric in our analysis, which refers to the power doing actual work. The difference between them is caused by inductors and capacitors in circuits. These two are power storage elements and the net power consumed by them *should* be zero. In reality though, it is a positive number more often than not.

5.2 Benchmarks

Benchmarking heterogenous systems requires generic benchmarks that perform the same task, preferably in a loop. It is important to perform multiple measurements to catch outliers and reason about performance consistency. Benchmark implementations should be comparable, yet a single, shared implementation for all platforms is usually not possible.

In our benchmark setup, we chose to focus on convolutional neural networks. First, these kind of networks are reasonably understood and implementations are mature. Second, convolutions are intrinsically parallel, so they lend themselves well to being run on accelerators that are made of many simple cores instead of a few complex ones like a CPU. We split the benchmarks into two categories: microbenchmarks encompass synthetic benchmarks exploring the boundaries of the hardware. Macrobenchmarks contain real-world neural networks, which have been used by other authors before. Since those networks solve real problems like object detection, they usually feature layers not present in synthetic networks and thus cover a wider range of applications.

All our benchmarks are implemented using the following scheme:

1. Setup

2. Run
3. Teardown

Step 1 includes the starting of monitoring tools to acquire CPU, RAM and accelerator metrics. These tools run for the entire duration of step 2, the main benchmark execution. In step 3, we stop the monitoring tools and save the recorded metrics. Using this scheme, we ensure each benchmarks can run stateless and monitoring tools only record data for the specific benchmark.

We chose to implement all benchmarks using the Google TensorFlow framework where possible. Some accelerators, such as the Google Cloud TPU required specific edits to the code so it would run. Others, like the FPGA systems, could not run TensorFlow code. Luckily, the Intel OpenVino library allowed us to convert TensorFlow models into FPGA compatible networks and run the network on that engine. In the remainder of this section, we list the various micro- and macrobenchmarks developed as part of this work and the tools used to run them and gather the metrics introduced in previous sections.

5.2.1 Microbenchmarks

We designed four main microbenchmarks, each being part of the convolutional neural network (CNN) architecture. The main components of a CNN are the convolutional layer and the pooling layer. The convolutional layer often incorporates an *activation* post-processing step, which introduces non-linearity into the network as explained in chapter 4. The pooling layer reduces the resolution of feature detection outputs. This way, the following layers loose some of the spatial information, but computational complexity is reduced by a huge amount. Furthermore, only those cells with low activation scores, as determined during the activation step in the convolution layer, are dropped. Thus, the important image features are preserved.

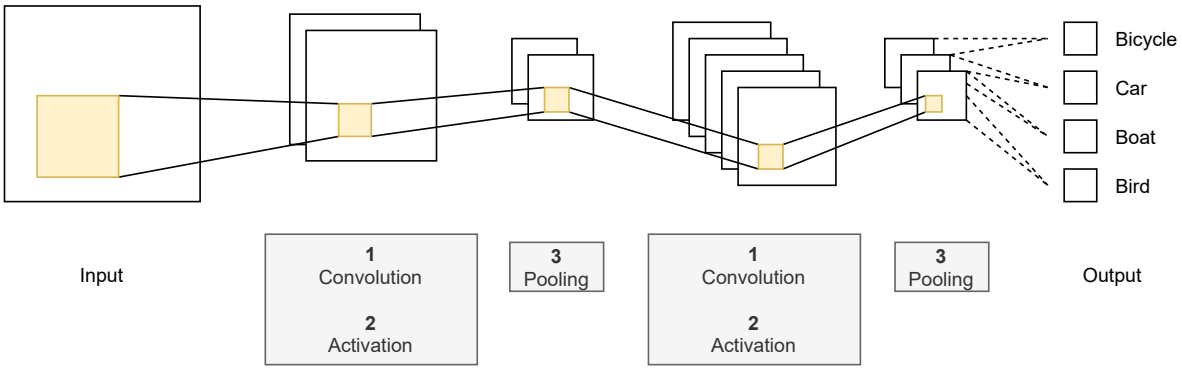


Figure 5.1: CNN microbenchmarks

Figure 5.1 illustrates the basic CNN pipeline. The areas covered by our microbenchmarks are shaded in gray. The last step, usually a mix of fully connected and flatten layers, is not covered, as no computation happens here. It is included in our macrobenchmarks though.

Table 5.6 lists the microbenchmark variants we developed. The parameters W, H, C refer to the width, height and channel count respectively. For each benchmark, we test a multitude of varying input sizes. For the *conv2d* operation, we test single-channel (grayscale) and multi-channel (RGB) input data. For the *maxpool2d* and *relu* benchmarks, we only tested multi-channel input since single-channel input exhibited low accelerator exploitation and business in our testing. The same thinking applies to the chosen parameter sizes. We tested

Table 5.6: Microbenchmarks

ID	Name	Description	Parameters (W x H x C)	
MIB1	conv2d	2D convolution	Input size	Kernel sizes
			100x100x1	3x3x1, 5x5x1
			1000x1000x1	3x3x1, 5x5x1
			3000x3000x1	3x3x1, 5x5x1
			100x100x3	3x3x3, 5x5x3
			1000x1000x3	3x3x3, 5x5x3
			3000x3000x3	3x3x3, 5x5x3
MIB2	conv2ddep	2D depthwise convolution	Input size	Kernel sizes
			100x100x3	3x3x3, 5x5x3
			1000x1000x3	3x3x3, 5x5x3
			3000x3000x3	3x3x3, 5x5x3
MIB3	maxpool2d	2D max pooling	Input size	Kernel sizes
			100x100x3	3x3x1, 5x5x1
			1000x1000x3	3x3x1, 5x5x1
			3000x3000x3	3x3x1, 5x5x1
MIB4	relu	rectified linear unit activation	Input size	
			100x100x3	
			1000x1000x3	
			3000x3000x3	

the most commonly used filter sizes of 3x3 and 5x5, since these filters cover a symmetric neighborhood around a single pixel. We vary the filter channel dimension according to the input data it operates on. In case of the *relu* benchmark, there are no kernel size parameters since the activation function operates on single values, not filter patches. For example, a grayscale input image of size 100x100 requires 100 activation function passes, one for each pixel value. A three-channel input image thus requires 300 activation function runs, and so on.

The *conv2ddep* benchmark performs a 2D depthwise convolution. Each channel of the image is convoluted individually and the output filters have the same number of channels. Since the operation is more complex than the "regular" convolution, it is more computationally expensive. Figure 5.2 illustrates the difference between the two operations. We take a RGB image as input and apply the convolutions to it. For regular convolutions, the filter depth (3 in this case) has to match the channel count of the input data (RGB, thus 3). The three dimensional multiplication done as part of each patchwise convolution yields a scalar value in the output filter. Thus, the output filter is one dimensional. In the case of depthwise convolution though, the input data and the kernel are split up into n parts of depth 1. Afterwards, each part of the input is convolved with the corresponding part of the kernel, yielding three output filters of depth 1 each in our case. The filters are then stacked on top of each other to form the output filter, which has the same depth as the input data again.

All of the benchmarks used by us allow for a high degree parallelism in theory. The convolution and pooling operations are essentially just sliding window operations, where the result of one pass does not affect the result of the pass over the next patch of pixels. Similarly, the activation function could be run on all image cells (pixels) in parallel. Since these functions are very simple, the added overhead of parallelism may not be worth

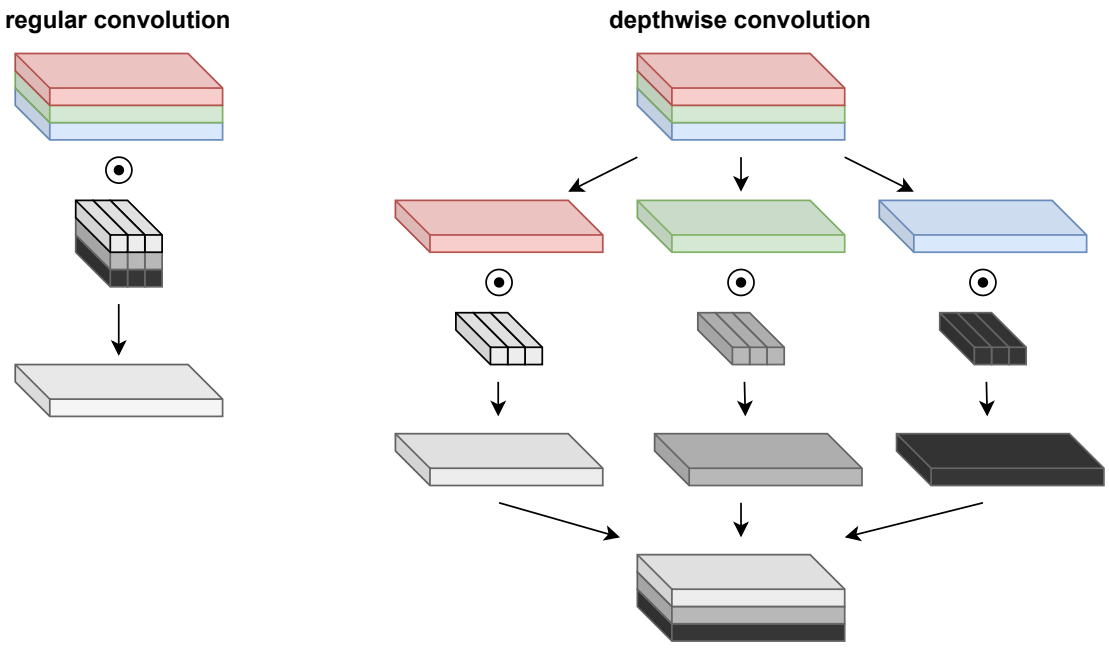


Figure 5.2: Regular vs. depthwise convolution

it in this case. We will come back to looking at the degree of parallelism exploited by TensorFlow when looking at the results in chapter 7.

5.2.2 Macrobenchmarks

We used two kinds of macrobenchmarks as part of this work. The first kind is a synthetic convolutional neural network (CNN) we developed in TensorFlow directly. We used that network for inference and training benchmarks. The second kind is YOLOv3 benchmarks, a state-of-the-art network developed by Redmon et al. [41]. We create two benchmark variants, one for the main YOLOv3 network and one for the reduced variant, YOLOv3-tiny. The latter has fewer layers to reduce the computational demands while maintaining a high performance (in terms of precision).

Figure 5.3 shows the architecture we chose for the synthetic CNN benchmarks. We designed the network to train and infer on the CIFAR10 dataset, because it is easily accessible through TensorFlow and widely used in other research works [17] [48] [24] [2]. The network layers are derived from the TensorFlow example [21]. We originally intended to test various input sizes other than the default 32x32 pixel images contained in the CIFAR10 dataset. Unfortunately, resizing the images on the fly would have skewed the results, as each platform would spend a different amount of time on the resizing operation. The alternative was to resize all images beforehand as a pre-processing step and keep them in RAM. This resulted in out-of-memory errors on most platforms except for the most powerful servers. In the end, we decided to explore the alternative input size of 224x224 pixels only for the inference benchmark, but not for the training benchmark.

Table 5.7 lists the macrobenchmarks we used to evaluate the performance of the heterogenous systems as part of this work. We also state the input sizes we use for testing. W x H x C refers to width x height x channels in this table. We mostly keep the recommended default values of 32x32 for CIFAR10 and 416x416 for YOLOv3. While YOLOv3 can work

[21] `https://www.tensorflow.org/tutorials/images/cnn?hl=en`

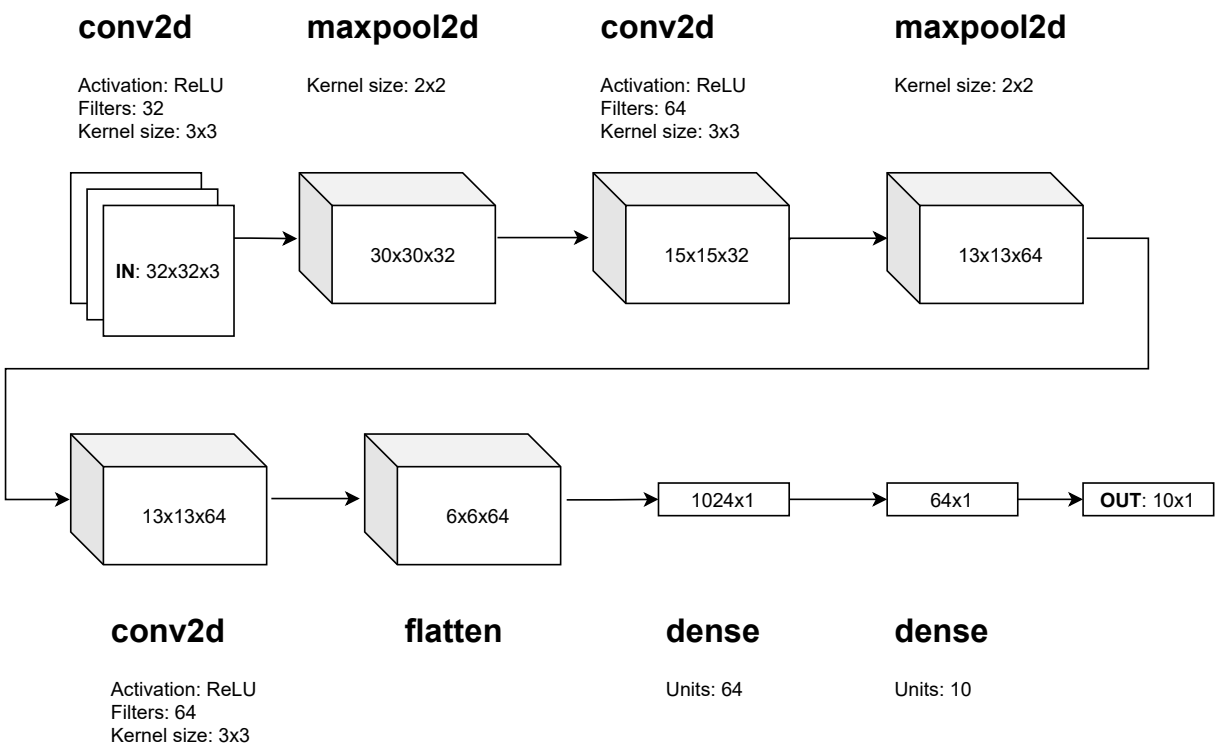


Figure 5.3: Synthetic CNN layers

with other input sizes such as 224x224 or 608x608, the 416x416 size is used in most real-world scenarios based on our findings. This is also a case of dimishing returns: according to Redmon, one of the main YOLOv3 authors, the mean average precision (mAP) scores listed in table 5.8 were recorded for the varying input sizes [41].

It should be noted that we used the TensorFlow implementation of YOLOv3 provided by Zhiao Zhang [22]. We did not use the "original" reference implementation in Darknet as developed by the YOLO authors, because the TensorFlow code allowed for better portability of the benchmark across systems. To ensure the TensorFlow implementation would not underperform the original code, we tested both variants on the Nvidia Jetson Nano board. We tested both the full and the tiny network variant for the 416x416 input size. We ran each implementation in a loop with 100 iterations, then computed the average FPS. There were close to no outliers during this short test. Furthermore, it is important to note that we exclude the last layer of YOLO from the benchmark time measurements, since it could not be quantized to run on FPGA hardware. The last layer excluded by us is used to compute the bounding boxes and perform non-maximum suppression (NMS) in YOLO.

As can be seen from the data in table 5.9, the TensorFlow implementation outperforms the Darknet one by a factor of roughly two.

5.3 Tools

We used various tools to carry out the benchmarking task on the various platforms. Apart from using Bash and Python to implement our own lightweight, modular benchmarking

[22] `https://github.com/zzh8829/yolov3-tf2`

Table 5.7: Macrobenchmarks

ID	Name	Description	Parameters (W x H x C)
MAB1	cnn-train	CNN training on CIFAR10	Input size 32x32x3
MAB2	cnn-infer	CNN inference	Input size 32x32x3 224x224x3
MAB3	YOLOv3	YOLOv3 object detection	Input size 416x416x3
MAB4	YOLOv3-tiny	YOLOv3-tiny object detection	Input size 416x416x3

Table 5.8: YOLOv3 performance

Input	mAP_{50}	FPS
320x320	51.5	45
416x416	55.3 (+ 7%)	35 (- 22%)
608x608	57.9 (+ 12%)	20 (- 55%)

framework, we developed and used additional tools to gather the metrics defined in Section 5.1.

5.3.1 perfstat

We developed a new tool to monitor CPU and RAM usage on Linux called *perfstat*. The tool is implemented in the safe systems programming language Rust. The motivation for developing a new tool instead of using existing ones such as "top" are twofold. First, top always scans all processes and computes statistics. This is unnecessary, as we already know which process we are interested in - the benchmark itself. Second, top tracks each PID individually. Our tool, *perfstat*, allows us to search for a command and track all PIDs spawned by that command. On each refresh iteration, the tool will scan for new processes or remove vanished ones from the tracked set. This allows for easy command monitoring without convoluted post-processing.

```
// example command name
cmd = "python3.7 benchmark.sh --timer 600 --engine CPU"

loop {
    // get the system CPU and RAM load
    cpu_stats = read_cpu_stats()
    ram_stats = read_ram_stats()

    // get the command CPU and RAM load
    cmd_cpu_stats = {}
    cmd_ram_stats = {}

    // search for processes spawned by a command
    pids = search_pids_for_command(cmd)
    // now compute per-process statistics for the command
    for pid in pids {
        cmd_cpu_stats += read_pid_cpu_stats(pid)
        cmd_ram_stats += read_pid_ram_stats(pid)
    }
    cmd_cpu_stats /= pids.len()
    cmd_ram_stats /= pids.len()
```

Table 5.9: YOLOv3 TensorFlow vs Darknet

Framework	Variant	FPS (avg)
Darknet	Full	1.7
	Tiny	7.0
TensorFlow	Full	3.3
	Tiny	16.7

```
22
23      // output stats to stdout or write to a CSV file
24      dump(..)
25  }
```

Listing 5.4: perfstat monitoring loop pseudocode

The basic *perfstat* monitoring loop is described in Listing 5.4. The tool tracks both the system statistics (lines 6 and 7) as well as the command stats (lines 17 and 18). A *command* in this case describes a set of processes spawned by a user command. Usually, this will be the python program that runs the TensorFlow benchmark.

5.3.2 Nvidia-smi

Nvidia provides a user-space tool with their driver stack on Linux called *Nvidia-smi*. It allows users to capture GPU device statistics, such as streaming processor clocks, usage, memory clocks and usage and more. Unfortunately, it does not work on their embedded boards, such as the Jetson Nano board we used. For that reason, we were not able to record GPU device usage metrics on the board.

Table 5.10: Nvidia-smi metrics

Key	Description	Unit	Example
timestamp	UTC Timestamp	-	"2021/01/17 02:54:37.791"
index	GPU index	-	"0"
utilization.gpu	GPU core utilization	%	"55"
utilization.memory	GPU memory utilization	%	"34"
memory.free	Amount of free GPU memory	MiB	"32480"
memory.used	Amount of used GPU memory	MiB	"30"
temperature.gpu	GPU board temperature	°C	"37"
power.draw	GPU board power draw	W	"46.67"

Table 5.10 lists the values we captured using the Nvidia-smi tool. We used external power monitoring devices to measure the power draw for the entire system, but it is certainly helpful to be able to compare this to just the accelerator power draw. Since the tool directly accesses the performance counters implemented in hardware on the GPU, the statistics should be the most accurate. On the other hand, these closed-source vendor-specific tools were known to contain bugs and report bogus values in the past, so they need to be observed carefully.

6 Setup

In this chapter, we present the various setups on which the experiments were conducted. The hardware can be categorized into three main classes:

1. Server
2. Desktop
3. Edge

We chose these hardware classes to cover a range of systems which are used to run neural network applications in real-world environments. With regards to our initial research goals, we believe this selection will allow us to make meaningful comparisons with regards to raw performance, performance per watt and performance per Euro.

This work by no means pretends to provide perfect comparison data on an exhaustive set of platforms. Instead, we seek to provide guidance in choosing the right platform for projects requiring neural network capabilities.

Most of the systems listed in this chapter was connected to a Voltcraft Energy Logger EL4000 to facilitate independent power usage monitoring. Additionally, vendor provided tools were used to measure the power draw of accelerator chips, e.g. the Nvidia GPU or Intel Arria FPGA.

6.1 Server

We used two very different servers to run our experiments on. The first is a Nvidia DGX-1 Volta GPU server equipped with eight Nvidia Tesla V100 GPUs. The GPU server also features vast amounts of RAM and the proprietary NVLink technology, which enables direct GPU-to-GPU communications without involving the CPU [23]. According to Nvidia [24], a single Tesla V100 GPU is able to provide 15.7 TeraFLOPS for single-precision and 7.8 TeraFLOPS for double-precision calculations in the NVLink setup. The available bandwidth amounts to 32 GB/s for CPU-GPU communication using PCI-Express and 300 GB/s for GPU-GPU communication using NVLink. The exact specifications are listed in table 6.1.

Table 6.1: Nvidia DGX-1 specifications

	S1: DGX-1
CPUs	2x Intel Xeon E5-2698
GPUs	8x Nvidia Tesla V100
RAM	512 GB RDIMM DDR4
Storage	4x 2TB SATA SSD RAID 0
Performance	1 Peta FLOPS
Power Usage	max. 3500 W
OS	Ubuntu 18.04 LTS
Cost	84,000 €

The second server is a Dell PowerEdge R740 which is equipped with two Maxeler Maia dataflow engines (DFE). Each of those DFEs features an Altera Stratix V FPGA and some RAM. Again, the exact specifications are contained in table 6.2.

[23] https://www.Nvidia.com/de-de/data-center/nvlink/
[24] https://www.Nvidia.com/de-de/data-center/tesla-v100/

Table 6.2: FPGA server specifications

	S2: PowerEdge R740
CPUs	1x Intel Xeon Gold 5217
DFEs	2x Maxeler Maia (Altera Stratix V 5SGSD8 FPGA with 48 GB DDR3)
RAM	96 GB RDIMM DDR4
Storage	SATA SSD
OS	Ubuntu 18.04 LTS
Cost	8,000 € (Dell Server) + 11,900 € (DFEs)

Looking at the two servers, one could assume the Nvidia GPU server to perform better for neural network tasks, given that it was specifically designed for machine learning purposes. Then again, it is the more powerful and costly choice just by going with the specifications on paper. However, as explained in earlier chapters, the hardware is only one part of accelerator technology. Software is the other, arguably more important one. With this in mind, the servers could not be more different since one is specifically designed for select purposes like image processing, rasterization and tensor operations. The FPGA server on the other hand is a general-purpose computation engine.

Apart from the two servers listed above, we also had access to hardware from the Institute of Microelectronic Systems (IMS) of the Leibniz University Hannover. The systems were accessible as part of the HAISEM Lab [25], a project funded by the German Federal Ministry of Education and Research (BMBF).

Table 6.3: IMS GPU servers

	S3: IMS-P100-16	S4: IMS-P100-32
CPUs	2x Intel Xeon E5-2697 v4	
GPUs	2x Nvidia Tesla P100-PCIe 16 GB	2x Nvidia Tesla V100-PCIe 16 GB
RAM	256 GB RDIMM DDR4	
OS	Ubuntu 18.04 LTS	
Year	2016	

Table 6.4: IMS GPU servers (cont.)

	S5: IMS-V100-32	S6: IMS-A100-40
CPUs	2x Intel Xeon Gold 6148 v4	2x AMD Epyc 7662
GPUs	2x Nvidia Tesla V100-SXM2 32 GB	4x Nvidia Tesla A100-SXM4 40 GB
RAM	384 GB RDIMM DDR4	1000 GB RDIMM DDR4
OS	Ubuntu 18.04 LTS	Ubuntu 20.04 LTS
Year	2017	2020

Tables 6.3, 6.4, 6.5 list the specifications of the IMS servers used in our experiments. The FPGA server is an outlier since its hardware was virtualized for the HAISEM project. The virtual machine resources are listed in Table 6.5.

Unfortunately, the FPGA server described in Table 6.2 was not useable for our experiments. The data-flow engines (DFE) could only be programmed using the proprietary Maxeler tools. They offer a C based API and a high-level MaxJ language to program the DFEs. While a deep-learning library was provided by Maxeler, we could not use it

[25] `http://haisem-lab.de/`

Table 6.5: IMS FPGA server

	S7: IMS-FPGA	VM
CPUs	2x Intel Xeon Gold 6248 v4	5 cores
FPGAs	4x Intel Arria 10 GX PAC	4x FPGA
RAM	384 GB RDIMM DDR4	62 GB
OS	Ubuntu 20.04 LTS	

due to software stack issues. The library did not detect a version of OpenMP, a required dependency.

We originally planned to conduct our benchmarks on a Google Cloud TPU equipped machine as well. Since the free tier of the Google Colab service providing TPU accelerated computing works by sharing a virtual machine between users, we tried to get exclusive access to a node. As mentioned in Section sec:tpu, we were not legally allowed to buy exclusive access to a Google Colab instance due to geographical restrictions. We then contacted Google, asking for access to the TensorFlow Research Cloud (TFRC) [26] where such access is granted for research purposes. Unfortunately, Google did not respond after initial contact in December 2020.

6.2 Desktop

Moving on to desktop class hardware, we evaluated an HP Omen 15 gaming laptop from 2019. These models were bought by the University of Hildesheim to be used by students in various computer science courses. At the same time, it also represents the kind of mobile hardware used by many machine learning engineers and data scientists. Thus, we think it makes for a valuable addition to the set of test hardware as part of this work.

Table 6.6: HP Omen 15-dh0007ng specifications

	D1: Omen Laptop
CPUs	1x Intel Core i7 9750H
GPUs	1x Nvidia GeForce RTX 2070 Max-Q
RAM	16 GB SDIMM DDR4
Storage	512 GB NVMe M.2 SSD
OS	Fedora Workstation 33
Cost	1,800 €

At the end of 2020, the author acquired an Apple Mac Mini machine featuring the company's newly developed M1 ARM SoC. The Apple M1 chip is unique in terms of its architecture compared to other consumer hardware platforms. Traditional x86 SoCs have been using the same layout since decades: a CPU is connected to off-chip RAM and talks to accelerators via PCI-Express. The Apple M1 SoC on the other hand features a *unified memory architecture*, where CPU, GPU, Neural Engine are all connected to one cache and one RAM module via *fabric*, Apple's new silicon interconnect mechanism.

Table 6.7 lists the specifications for the Mac Mini 2020 model. According to the Apple website on power draw [27], the Mac Mini M1 2020 has a maximum power draw of 39W for the entire system. This puts it below the HP Omen Laptop's Intel CPU, which is rated for 45W TDP. Since the Apple M1 SoC is also built into the 2020 MacBook Air and Pro

[26] `https://www.tensorflow.org/tfrc`

[27] `https://support.apple.com/de-de/HT201897`

Table 6.7: Apple Mac Mini M1 2020 specifications

	D2: Mac Mini M1
CPUs	1x Apple M1 (4 high power, 4 high efficiency cores)
GPUs	1x Apple M1 (8-core)
RAM	16 GB
Storage	1 TB SSD
OS	macOS 11.2
Cost	1,500 €

models, it will be interesting to compare the benchmark results to the HP Omen Laptop in terms of power and efficiency.

6.3 Edge

Edge computing is different from the previous two system classes in many ways. For one, neural networks are usually not trained on edge devices because they run on a tight thermal and power budget. Furthermore, they are not built for debugging and profiling or any development tasks due to their lack of I/O ports and CPU power. Edge devices usually feature select coprocessors which carry out a particular task in isolation. The CPU, which can often be a lot weaker in terms of raw compute power, will offload the computation to that coprocessor at runtime.

For our experiments, we chose to use a Nvidia Jetson Nano Developer Kit [28]. It is a single board computer (SBC) akin to a Raspberry Pi for example. The printed circuit board (PCB) packs several I/O pins, an ARM CPU, the Nvidia Maxwell GPU chip, some RAM and various connectivity options. Table 6.8 lists the specifications according to [29].

Table 6.8: Nvidia Jetson Nano Developer kit specifications

	E1: Nvidia Jetson Nano
CPUs	1x ARM A57 SoC (4-core)
GPUs	1x Nvidia Maxwell GPU (128-core)
RAM	4GB LPDDR4
Storage	microSD
Performance	472 GFLOPs
Power Usage	5/10 W
Cost	100 €

Nvidia markets the Jetson Nano as a platform for makers, learners and embedded developers. They especially tout its image classification, object detection and speech processing capabilities. Thus, we believe it is a good candidate to add to the array of test systems.

[28] `https://developer.nvidia.com/embedded/jetson-nano-developer-kit`
[29] https://coral.ai/docs/dev-board/datasheet

7 Results

In this chapter, we present the results of our benchmark experiments. The data we gathered is too large to be fully represented here. Thus, we show only the most interesting and meaningful results. More detailed benchmark results including data for all benchmark variants is provided in Appendix A.1.

Table 7.1: System matrix

ID	Description	Variants
S1	DGX-1	CPU, GPU
S3	IMS P100 16 GB	GPU
S4	IMS P100 32 GB	GPU
S5	IMS V100	GPU
S6	IMS A100	GPU
S7	IMS Arria 10	FPGA
D1	Omen Laptop	CPU, GPU
D2	Mac Mini M1	CPU, GPU
E1	Jetson Nano	CPU, GPU

Table 7.1 lists the systems used in our experiments and assigns an ID to each. This was done so we can easily refer to the systems throughout this chapter, without writing out their names. The IDs are taken from the tables in Chapter 6. The last column of Table 7.1 lists the variants we tested. For example, system S1 features a powerful Intel Xeon CPU that we tested in isolation from the GPU it has equipped. Throughout this chapter, we refer to the CPU variant of system S1 like this: $S1_{CPU}$. The same notation is used for all the other systems and their variants.

In the remainder of this chapter, we first show a full example of all data we recorded during each benchmark. Second, we list and analyze the microbenchmark results for the various heterogenous systems in Section 7.2. Afterwards, we present the macrobenchmark results in Section 7.3. Finally, we discuss the results and compare the system performance in terms of latency and power usage in Section 7.4.

7.1 Full data example

As stated in the introduction of this chapter, it is not possible for us to show all data gathered through benchmarking and monitoring here. The total data we collected (including autogenerated plots and statistics) amounts to roughly 50 GB for all systems combined. In this section, we show a full example of all data for a single benchmark run on one system variant.

We show a "Busy (%)" metric for the benchmark runs where available. This value describes the usage of the processors running the test. For CPU variants, this means the CPU usage across all cores. For GPU variants, the metric refers to the usage of the GPU cores. The values refer to the T_b metrics as defined in Section 5.1.

We chose to show the 2D max pooling benchmark ("maxpool2d") data from system variant $S1_{GPU}$. As a reminder, system S1 features eight Nvidia Tesla V100 GPUs, each rated for a total board power of 250W according to the specifications [30].

[30] `https://images.nvidia.com/content/tesla/pdf/Tesla-V100-PCIe-Product-Brief.pdf`

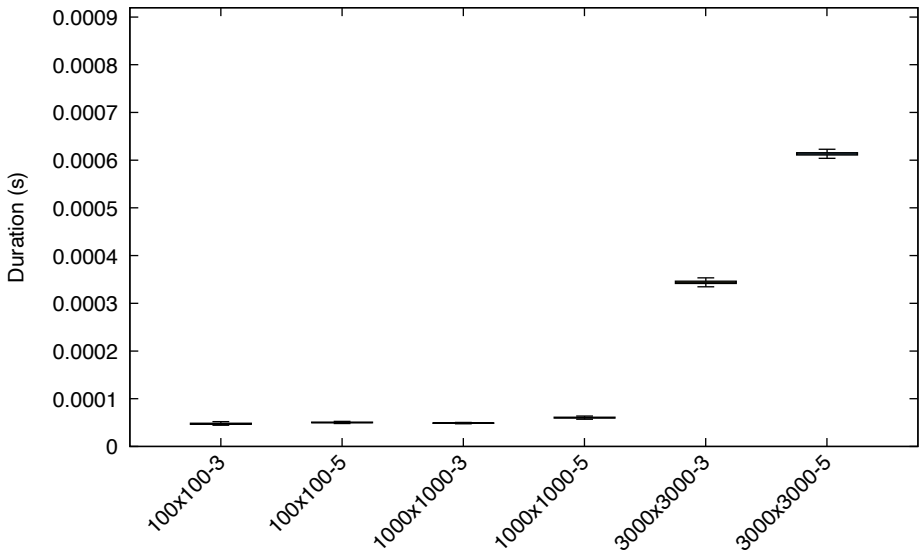


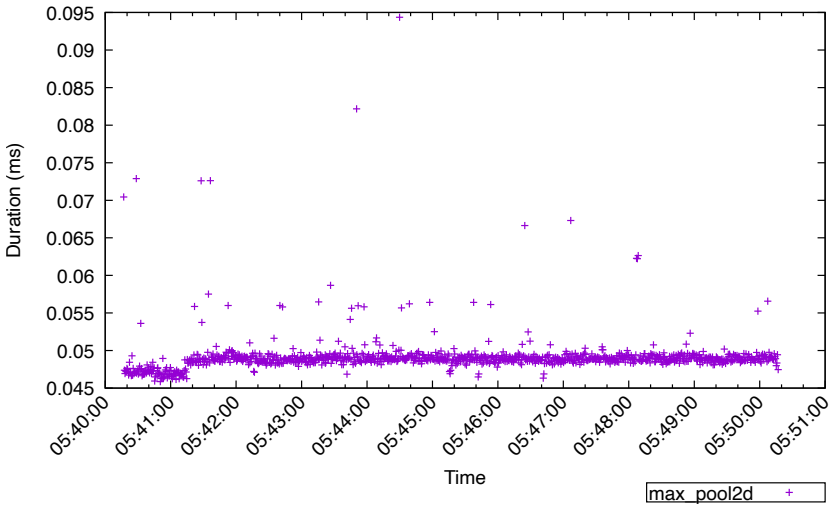
Figure 7.1: $S1_{GPU}$ maxpool2d benchmark results

Figure 7.1 shows a box plot for the mean benchmark iteration durations ("latencies") of the parametrized benchmarks. There are several conclusions that can be drawn from the plot. First, the standard deviation of the latencies is very small. For some tests, e.g. 100x100-5 (which means input size of 100x100 elements and window size of 5x5) it seems virtually nonexistent. The autogenerated statistics confirm this assumption: for 9073646 benchmark iterations completed in 600 seconds, the mean value is approx. 5.1 x 10^{-5} seconds while the standard deviation is approx. 7.1 x 10^{-6} seconds. We observed very small standard deviations for almost all our benchmark results across all systems, meaning we expect only few outliers in the data.

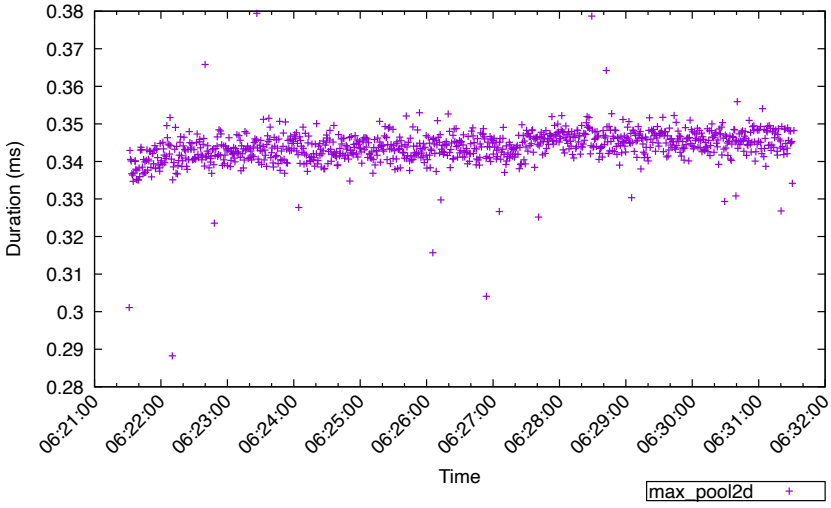
We now take a closer look at two benchmark variants, 1000x1000-3 and 3000x3000-3. This way, we can reason about the effect of the input size on latency, GPU business and power draw. From Figure 7.1 we can see that e.g. the change in input size from 100x100 to 1000x1000 does not appear to noticeably affect the latency. The first jump comes with the 3000x3000 input size.

Figure 7.2 plots the benchmark iterations completed in 600 seconds for both variants. For benchmark (a), there were 9392384 iterations performed, while microbenchmark (b) was run 1668963 times (approx. 82% less). The data in (a) was subject to a standard deviation of 1.1 x 10^{-5}, while (b)'s was 1.4 x 10^{-5}. One interesting difference between the two result sets is the position of outliers. In (a), most outliers were caused by higher-than-average latencies. In (b) however, half of the outliers are below the mean value observed. Overall, the increase in input size from $1000x1000 = 1,000,000$ to $3000x3000 = 9,000,000$ (900% increase) values led to a mean latency change from 0.049 to 0.344 milliseconds (702% increase).

Figure 7.3 shows the business of the streaming multiprocessing (SM) cores which carry out GPGPU operations. We can see that GPU 0 is busy about 70% throughout benchmark (a). Benchmark (b) shows 100% GPU usage, meaning we are using the GPU to its full capabilities (assuming optimal exploitation by the TensorFlow framework and the CUDA backend).



(a) 1000x1000-3



(b) 3000x3000-3

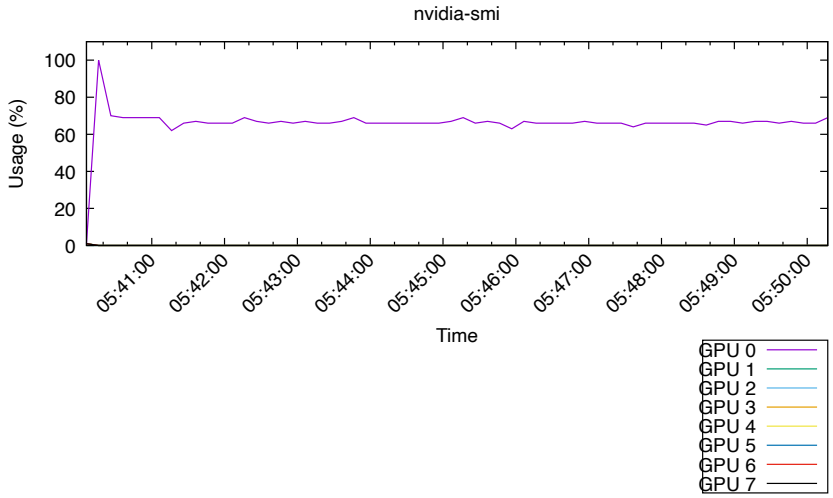
Figure 7.2: $S1_{GPU}$ maxpool2d GPU benchmark

Figure 7.4 plots the power draw of the GPUs installed in system S1 during the benchmark runs. Benchmark (a) leads to a GPU power consumption of approx. 250 watts, while benchmark (b) causes a consistent draw of approx. 300 watts. The idle power draw of each GPU in the system is about 45 watts, as can be seen in the very beginning of the plots. Interestingly, the idle power draw of all GPUs increases by a few watts when one GPU is being used.

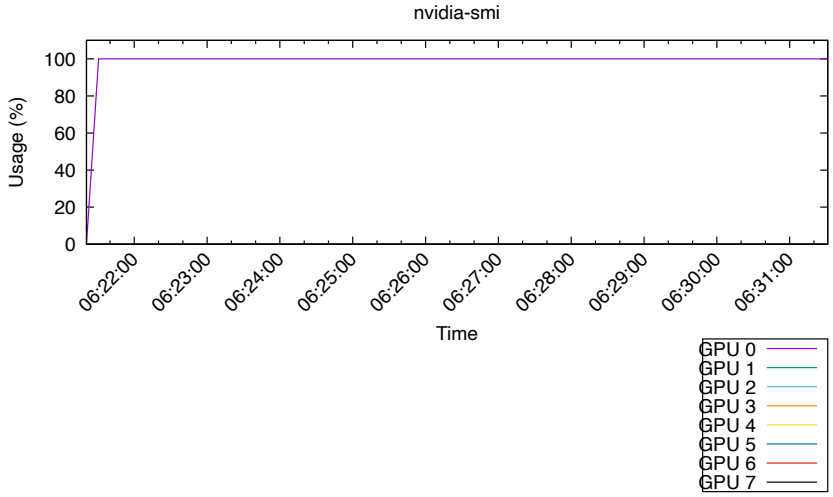
For the remainder of this chapter, we only show the reduced data. We still provide detailed data for system S1 so we can discuss the effect of varying input sizes, kernel sizes etc. for each particular benchmark.

7.2 Microbenchmarks

In this section, we show our microbenchmark results for the systems listed in Table 7.1. All the metrics like latency, power usage, cpu usage etc. are mean values. We calculate the standard deviation for all metrics, but do not display them here for brevity. Excess standard deviations are explicitly mentioned where necessary. We selected benchmark



(a) 1000x1000-3



(b) 3000x3000-3

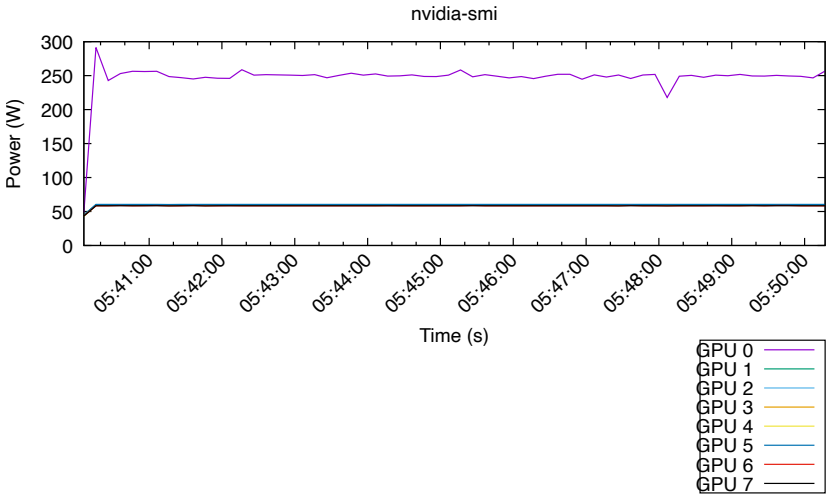
Figure 7.3: $S1_{GPU}$ maxpool2d GPU usage

variants for system S1 in the CPU and GPU configuration. This way, we can reason about the general benchmark task characteristics. For all other systems, we only show the most demanding benchmark variant, since that will exploit the systems' hardware the most.

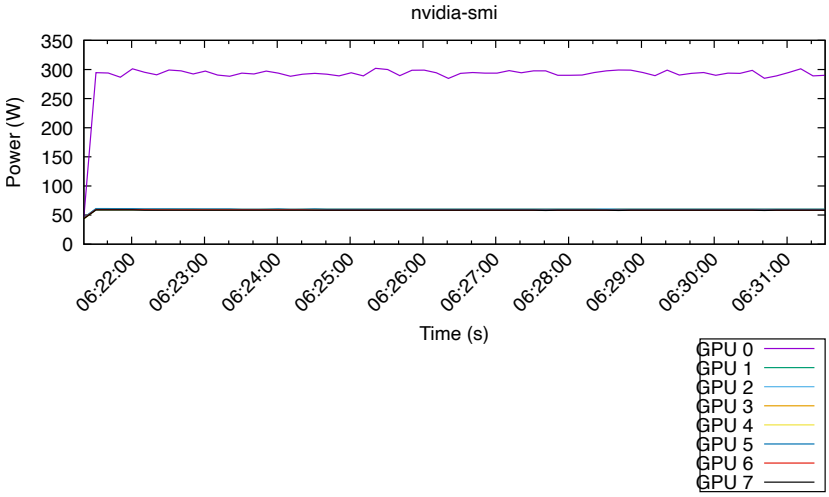
Each benchmark was run for 600 seconds. Most systems were able to complete over a million benchmark iterations during this time. This enables us to argue about the stability of the results, since only a very small number of outliers was observed.

7.2.1 2D convolution

We ran the 2D convolution benchmarks in a number of variants on each platform. In total, we test six input sizes and two kernel sizes per input size, so $6^2 = 36$ combinations. Since we cannot possibly display all of these results here, we decided to only choose three variants for system S1. First, we show the results for the input size 1000x1000x3 and the kernel 3x3x3. We follow up with the 3000x3000x3 input size and the same kernel size to be able to reason about the effect on the results. Finally, we conclude with the 3000x3000x3



(a) 1000x1000-3



(b) 3000x3000-3

Figure 7.4: $S1_{GPU}$ maxpool2d GPU power

input size again, but this time in combination with the 5x5x3 kernel size, to find out how changing kernel size affects the result.

Table 7.2 lists the benchmark results. Looking at the results for system S1, the input size seems to have a larger impact on the latency than the kernel size. The power draw does not change much between the variants. On system S1, the CPU is not fully saturated, while the GPU is. If we compare the 1000x1000x3 input size, 5x5x3 kernel size with between the CPU and GPU runs, we can see that the GPU offers a speedup of roughly 15x, but an increase in power draw of only approx. 23%.

The Mac Mini M1 (system D2) delivers the most impressing results in this benchmark. It achieves a mean latency of only 0.4 ms, which is the lowest among all systems. At the same time, it consumes only 12 watts. Compared to the Nvidia DGX-1 GPU server (system S1), the Apple device offers a 10x speedup at a power usage decrease of 99%. If we compare Nvidia's embedded offering E1 to the server class S1 system, we arrive at a 10x slowdown, but a power usage decrease of more than 99% as well.

Table 7.2: 2D convolution

ID	Input	Kernel	Latency (ms)	Busy (%)	System Power (W)
$S1_{CPU}$	1000x1000x3	3x3x3	7.9	64	895
	3000x3000x3	3x3x3	39.5	84	957
	3000x3000x3	5x5x3	60.0	84	1007
$S1_{GPU}$	1000x1000x3	3x3x3	0.3	100	1178
	3000x3000x3	3x3x3	2.3	100	1211
	3000x3000x3	5x5x3	3.9	100	1235
$S3_{GPU}$	3000x3000x3	5x5x3	9.9	100	—
$S4_{GPU}$			9.5	100	—
$S5_{GPU}$			3.9	100	—
$S6_{GPU}$			1.2	100	—
$D1_{CPU}$	3000x3000x3	5x5x3	240.7	99	73
$D1_{GPU}$			11.6	100	129
$D2_{CPU}$			34.9	—	15
$D2_{GPU}$			**[0.4]**	—	12
$E1_{CPU}$	3000x3000x3	5x5x3	3255.2	99	6
$E1_{GPU}$			404.5	—	7

During our data analysis, we discovered an interesting behavior pattern for system D2 with regards to the benchmark parameters. When comparing the mean iteration times of the benchmark variants (varying input and kernel sizes), the system would spend almost exactly the same amount of time for each iteration.

Figure 7.5 shows a plot of the distribution of all benchmark variants for the 2D convolution benchmark. While we can clearly see the impact of varying input and kernel sizes for other systems, the iteration times for system D2 are almost *evenly* distributed. This was the case for all microbenchmarks run on the system. We believe that this is an implementation bug in the Apple TensorFlow port, which is still in alpha state as of early 2021 [31]. To ensure this was not a one-off error, we ran the benchmark suite two more times, but each time arrived at roughly the same data.

7.2.2 2D depthwise convolution

The depthwise convolution benchmark contains three input sizes and two kernel sizes per input size. In total, there are $3^2 = 9$ benchmark variants. We again show three combinations for system S1 and the most demanding for the other systems. For some reason, the benchmark was not executed successfully on system S3, thus there is no data available.

Table 7.3 lists the benchmark results and Figure 7.6 plots the iteration time distribution. We can immediately recognize a pattern similar to the previous 2D convolution benchmark. The input size appears to have a bigger effect on the latency than the kernel size again. Comparing the CPU against the GPU for the tested systems leads to a huge win for the GPU in terms of latencies. At the largest input size, the GPUs on all systems are fully utilized.

Again, system D2 wins the benchmark. Compared to the powerful GPU server S1, it offers a speedup of roughly 2x and a decrease in power usage of approx. 99%. Since the system

[31] `https://github.com/apple/tensorflow_macos`

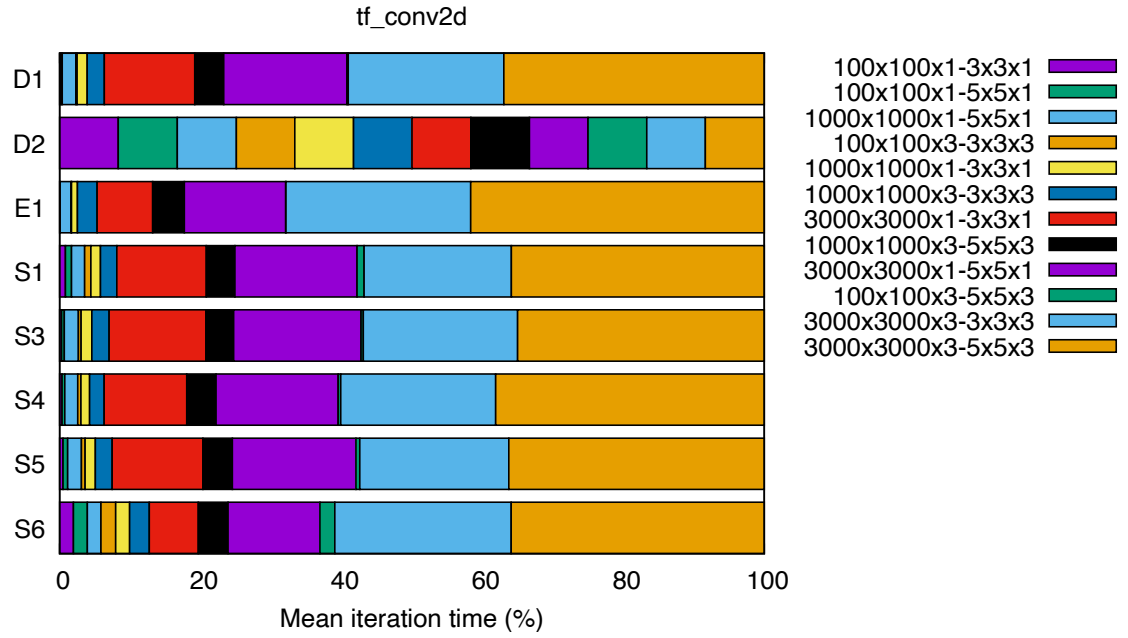


Figure 7.5: 2D convolution iteration time distribution

closest to D2 in power usage is E1, we take a closer look at the two. Comparing E1 against D2 unveils a 50x slowdown at a power usage decline of only 41%.

7.2.3 2D max pooling

The 2D max pooling benchmark features three input sizes and two kernel sizes for each, resulting in $3^2 = 9$ benchmark variants in total. We cover three variants for system S1 and one for all other systems.

Table 7.4 shows the benchmark results. The iteration time distribution is plotted in Figure 7.7. Similar to the previous two benchmarks, the input size has a bigger impact than the window size in terms of latency. GPUs beat the CPUs again by a large margin. The TensorFlow code is able to fully saturate the GPUs, hinting at good parallelization.

The Apple Mac Mini again beats all other systems. It offers a 2x speedup and a 99% decline in power usage compared to system S1. Compared to a system in its own class, system D1, D2 still achieves a 16x speedup at 90% less power usage.

7.2.4 ReLU activation

The ReLU benchmark features three distinct input sizes, thus three variants. We show the two largest for system S1 and the largest for the other systems.

Table 7.5 displays the benchmark results, whereas the iteration time distribution is plotted in Figure 7.8. Again, GPUs offer a huge speedup compared to CPUs. The level of parallelization allows for full GPU usage, while the CPUs cannot be used to their full potential. This may be related to the slower RAM accessible to the CPU versus the faster GPU memory. The results for system E1 support this claim, since the memory access latencies are hidden by the slower CPU cores.

Like in the other microbenchmark results, system D2 achieves the lowest latency at 0.1 ms. Compared to system S1, it offers a 3x speedup and a 99% decline in power usage.

Table 7.3: 2D depthwise convolution

ID	Input	Kernel	Latency (ms)	Busy (%)	System Power (W)
$S1_{CPU}$	1000x1000x3	3x3x3	4.9	70	922
	3000x3000x3	3x3x3	44.6	86	948
	3000x3000x3	5x5x3	106.3	88	970
$S1_{GPU}$	1000x1000x3	3x3x3	0.2	37	1023
	3000x3000x3	3x3x3	0.6	100	1265
	3000x3000x3	5x5x3	1.3	100	1246
$S3_{GPU}$	3000x3000x3	5x5x3	—	—	—
$S4_{GPU}$			3.6	100	—
$S5_{GPU}$			1.3	100	—
$S6_{GPU}$			0.9	100	—
$D1_{CPU}$	3000x3000x3	5x5x3	515.1	100	69
$D1_{GPU}$			4.1	100	130
$D2_{CPU}$			59.7	—	13
$D2_{GPU}$			**[0.7]**	—	12
$E1_{CPU}$	3000x3000x3	5x5x3	2757.1	99	6
$E1_{GPU}$			359.0	—	7

Furthermore, even the $D2_{CPU}$ test variant achieves a 3x speedup compared to $S1_{CPU}$, which is a powerful Intel server CPU.

7.3 Macrobenchmarks

Our macrobenchmarks cover training and inference using a synthetic convolutional neural network (CNN). Additionally, we use the YOLOv3 network to test real-world inference performance across the systems.

Similar to the microbenchmarks, each benchmark was run for 600 seconds. Depending on the test variant, most systems were able to complete between a hundred and several thousand iterations. Given the small standard deviation of the results, this confirms the stability of our benchmark results.

7.3.1 CNN inference

The CNN inference benchmark features two variants with a varying input size. Due to software dependency issues, it was not possible to run this benchmark on the Apple Mac Mini (system D2).

Table 7.6 lists the benchmark results. The iteration time distribution is plotted in Figure 7.9. This time, we can observe low resource utilization on both CPUs and GPUs throughout the range of test systems. With regards to latency, there is another interesting observation to be made. On system S1, the CPU outperforms the GPU, something that did not happen in other benchmarks. On the other systems, e.g. D1 or E1, the contrary is true.

Interestingly, the HP Omen Laptop (system D1) wins this benchmark with a recorded mean latency of 21.6 ms. This amounts to a speedup of approx. 1.5 and a decrease in power usage of 90% compared to the Nvidia GPU server (system S1).

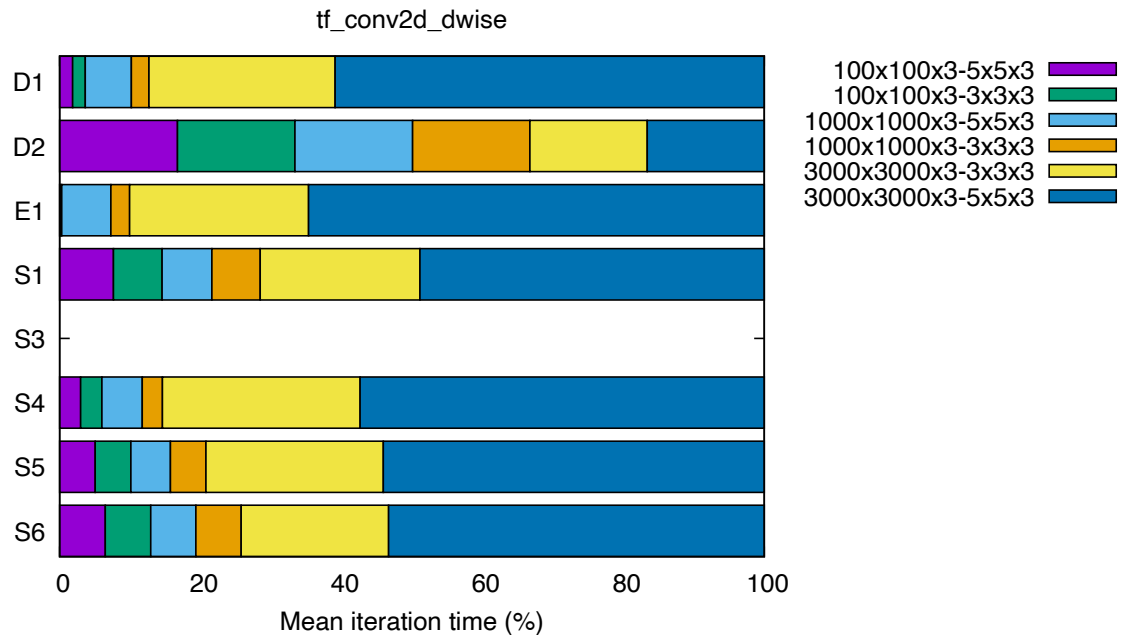


Figure 7.6: 2D depthwise convolution iteration time distribution

7.3.2 CNN training

In our CNN training experiment, we train a model using the architecture presented in Section 5.2.2. Each benchmark iteration performs a whole training iteration using all training images from the CIFAR10 dataset. The dataset contains images composed of 32x32 pixels. While the accuracy of the model improves during training, the iteration times are constant.

The CNN training benchmark exhibited an interesting RAM usage pattern. We show this by looking at system $S1_{CPU}$, but the same pattern was observed on all systems, regardless execution engine (CPU or GPU). Figure 7.10 shows the RAM usage of the benchmark process (PID) and the system during the benchmark. At roughly 15:27:00, there is a sharp pullback in memory usage.

Interestingly, there is no such anomaly in the CPU usage of either the benchmark process or the system in general, as can be seen in Figure 7.11.

Looking at a plot of the benchmark iterations in Figure 7.12, there does appear to be an outlier at roughly 15:27:00. Then again similar outliers are observed around e.g. 15:22:00 or 15:29:00, with no observable RAM usage anomalies.

Table 7.7 shows the benchmark results. Interestingly, the advantage of GPUs over CPUs is not very large in this benchmark. System S3 is an outlier in this benchmark, as its average iteration time was 125.9 seconds. We do not know why this is the case. At the same time, the GPU activity data reveals 24% usage during the benchmark, which only raises more questions. System E1 crashed due to out-of-memory errors at the beginning of the benchmark. This happened for both the CPU and the GPU test variant. Perhaps one could make it work using a different allocation strategy than the one TensorFlow uses by default.

Table 7.4: 2D max pooling

ID	Input	Kernel	Latency (ms)	Busy (%)	System Power (W)
$S1_{CPU}$	1000x1000	3x3	52.0	1	790
	3000x3000	3x3	511.5	1	786
	3000x3000	5x5	1188.1	1	787
$S1_{GPU}$	1000x1000	3x3	0.05	70	1162
	3000x3000	3x3	0.3	100	1234
	3000x3000	5x5	0.6	100	1227
$S3_{GPU}$	3000x3000	5x5	1.9	100	—
$S4_{GPU}$			1.9	100	—
$S5_{GPU}$			0.6	100	—
$S6_{GPU}$			0.4	100	—
$D1_{CPU}$	3000x3000	5x5	846.1	8	50
$D1_{GPU}$			1.9	100	128
$D2_{CPU}$			1141.4	—	18
$D2_{GPU}$			**[0.3]**	—	12
$E1_{CPU}$	3000x3000	5x5	5885.6	25	3
$E1_{GPU}$			167.5	—	7

The Nvidia Tesla V100 GPU server (system S1) achieves the best mean iteration times coming in at 5.3 seconds. Comparing the Apple Mac Mini to this, we find that the server is not able to show even a 2x speedup, all while drawing 84x more power.

7.3.3 YOLOv3 inference

In our YOLOv3 benchmarks, we test both the full and the tiny model. We always use a fixed input size of 416x416, since this is the optimal trade-off between accuracy and speed according to its authors [41].

Table 7.8 contains the results for the YOLOv3 benchmarks. Again, the advantage of GPUs over CPUs is obvious, often only consuming a few watts more. On the other hand, no system is able to fully exploit its hardware capabilities, meaning there is a bottleneck somewhere in the network architecture. Our microbenchmark results point at memory latencies, since small input sizes lead to less GPU accelerator usage in the 2D convolution, pooling and ReLU benchmarks.

Figure 7.13 shows the iteration time distribution. Unlike the previous benchmarks, this one does not exhibit the bug we found earlier on system D2. Since we run exactly the same code on all systems in our benchmark suite, we are not sure why the bug is triggered only for certain benchmarks. We believe it may be related to explicitly choosing the Apple MLCompute GPU engine to execute all TensorFlow code.

While it was not possible to measure the power draw for the FPGA system S7, we were able to measure the total power draw of the FPGA cards. When running the full YOLOv3 model, the FPGA board consumed between 27.5 and 48.5 watts. For the YOLOv3-tiny variant, we measured a power draw between 27.5 and 30.5 watts. At these levels of energy consumption, the FPGA server outperforms all other server class machines by far. For example, a single Nvidia Tesla V100 GPU in system S1 draws about 45 watts in idle state. When fully loaded, this can go up to roughly 300 watts.

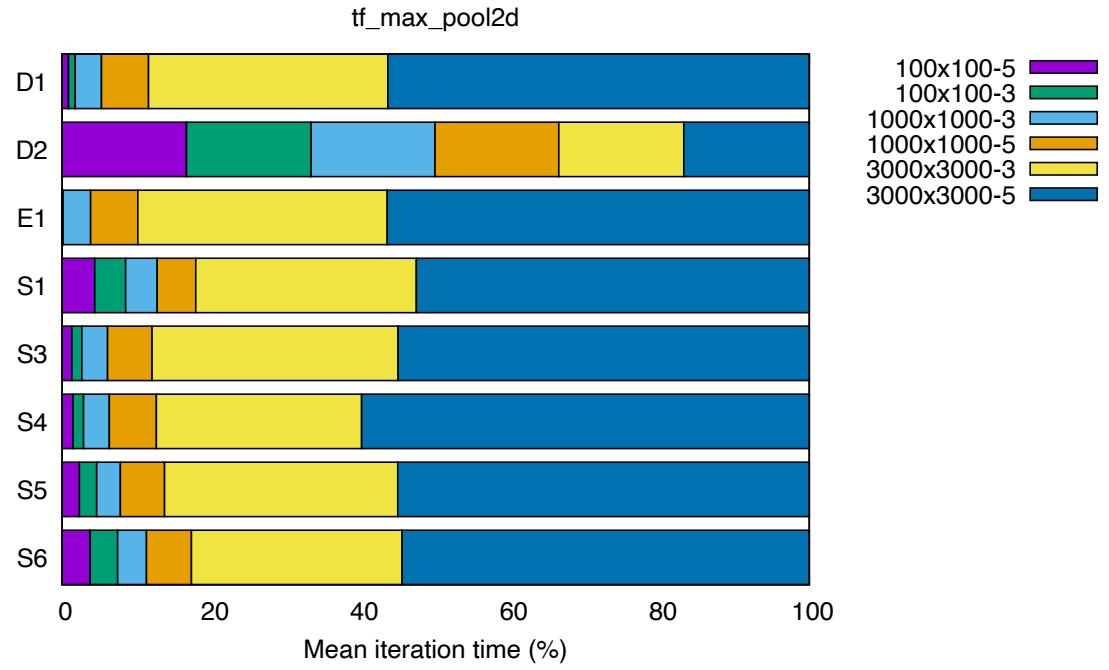


Figure 7.7: 2D max pooling iteration time distribution

We noticed an interesting pattern in the YOLOv3 benchmark latency distributions on system D2 (Apple M1). When running the benchmark on the GPU, we could observe outliers starting to form a parabolic shape.

We plotted the YOLOv3-Full 416x416 benchmark results for system D2 in Figure 7.14. The pink crosses are single iteration results, where each iteration represents a full forward pass in the YOLOv3 network. We also plotted the mean and the standard deviation for convenience. While negative durations are not possible, we added the Y axis offset here for visual clarity of the standard deviation. The parabolic outliers start appearing at roughly 03:21:00. The first data point falling outside of the standard deviation range is the one at approx. 03:22:35. The last outlier recorded is the one around 03:29:00, taking over 100 seconds to finish the iteration.

For comparison purposes, we plot the YOLOv3-Full 416x416 benchmark results for system D1 (HP Omen Laptop) as well in Figure 7.15. As can be seen from the plot, the outliers are distributed as one would expect them to be. There are frequent data points that fall outside of the standard deviation range. Yet even the outliers are rather close to the mean and do not form a shape of any particular kind, contrary to the results from system D2. System D2 was unique in showing this anomaly. We think it may be related to thermal constraints on the small Apple machine. Unfortunately we had no tool at our disposal to record the CPU load of the machine during the benchmarks, so detailed monitoring data is not available.

7.4 Discussion

The data we gathered through benchmarking the heterogenous systems allows us to draw multiple conclusions about convolutional neural networks on modern heterogenous systems. First of all, the microbenchmark results in Section 7.2 demonstrate the advantage of many low-complexity cores over few high-complexity ones (e.g. a CPU). For sufficiently large input sizes, GPU accelerators are operating at almost 100% core usage. Even for

Table 7.5: ReLU

ID	Input	Latency (ms)	Busy (%)	System Power (W)
$S1_{CPU}$	1000x1000	0.2	57	929
	3000x3000	11.6	52	843
$S1_{GPU}$	1000x1000	0.02	95	1089
	3000x3000	0.3	100	1082
$S3_{GPU}$	3000x3000	0.4	100	—
$S4_{GPU}$		0.4	100	—
$S5_{GPU}$		0.3	100	—
$S6_{GPU}$		0.2	100	—
$D1_{CPU}$	3000x3000	13.6	69	71
$D1_{GPU}$		0.6	100	132
$D2_{CPU}$		3.7	—	15
$D2_{GPU}$		[**0.1**]	—	12
$E1_{CPU}$	3000x3000	28.8	95	6
$E1_{GPU}$		18.4	—	7

small input sizes, approx. half of the cores are used as is visible in Appendix A.1. The building blocks of CNNs, convolutional layers, pooling layers and activation functions are highly parallelizable. Thus, they intrinsically lend themselves well to modern accelerator architectures such as GPUs, TPUs, or FPGAs.

That being said, there are some remarkable differences between the systems even within their own accelerator class. For example, the Apple M1 GPU consistently beats the much larger, much costlier Nvidia GPU server in raw performance. This is a very impressive result, since the power budget of the M1 system is smaller by more than an order of magnitude. Although the Apple M1 SoC features dedicated neural network accelerator cores, they were not used in our benchmarks. Another interesting fact is that the Apple M1 device was the only system to draw less power while using the GPU than using the CPU to run the benchmarks.

Looking at systems D1 (the HP Omen Laptop) and E1 (the Nvidia Dev Board), we see that dedicated accelerator hardware is paramount to competitive neural network performance. The most powerful Intel/AMD CPUs used in the server systems S1 through S7 already deliver low performance compared to any kind of modern accelerator chip. But on mobile or embedded devices, the difference is even more critical. These devices often operate on a tight power budget, so achieving CPU levels of performance at a much lower power draw is a desirable trait.

The macrobenchmark results in Section 7.3 are interesting in their own right. System S1 shows better results when running on the CPU instead of the GPU, regardless of input size. At the same time, the power draw of the system is higher when running the benchmark on the GPU. This is not the case on e.g. system E1, which is able to profit from the available GPU accelerator both in terms of benchmark iteration times and power draw. The IMS GPU servers, systems S3 through S6 show curious behavior as well. Systems S3 and S4 are equipped with nominally weaker GPUs than S5 and S6, yet they achieve better benchmark results. Overall, we conclude that the CNN model is not complex enough to exploit the systems' capabilities, as the more powerful accelerators show single-digit usage levels.

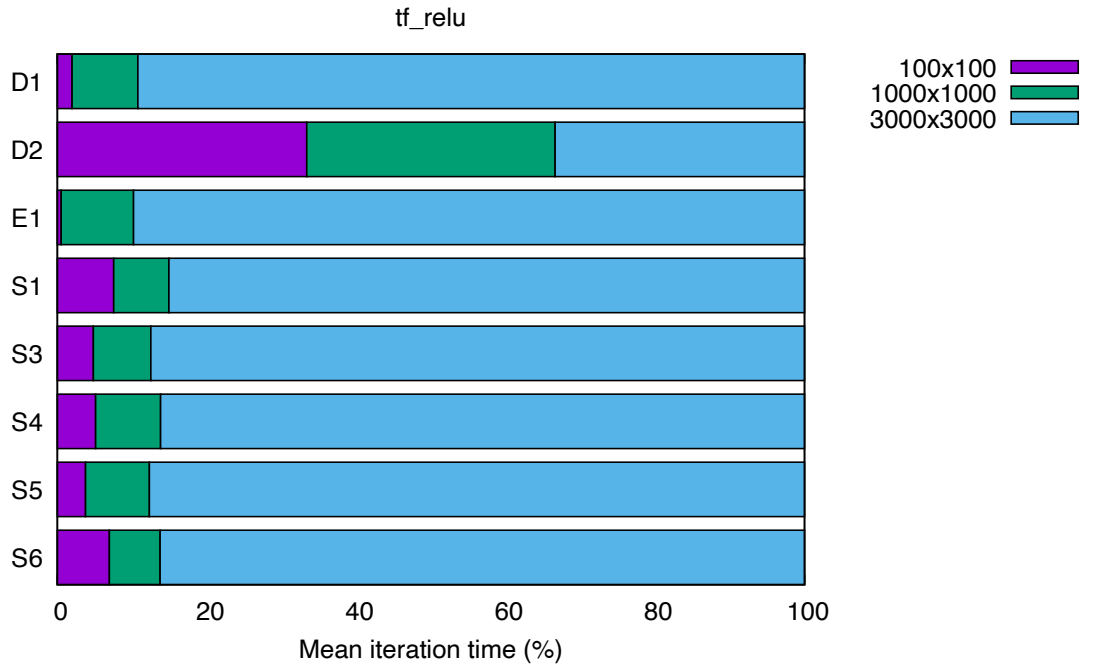


Figure 7.8: ReLU iteration time distribution

The CNN training benchmarks again indicate better performance for the accelerators compared to the system's CPU. The actual hardware usage (and thus the power draw) remains low though. This benchmark is also one of the first cases where the Apple system, D2, is severely outperformed by the most powerful server class systems S1 and S6.

Finally, the YOLOv3 benchmarks are mostly in line with the other results. Accelerator chips are able to reduce the latency by orders of magnitude in most cases compared to the CPU. System D1 is able to achieve surprisingly good results, given that it is consumer grade hardware. At a GPU core usage of only 8%, it beats the more powerful GPU server S1. At the same time, D1 uses only a fraction of the power. The most important data point in this benchmark is system S7. The Intel Arria 10 FPGA accelerator achieves the lowest latencies by a large margin. No other system comes close to achieving its sub millisecond inference latencies for YOLOv3.

Table 7.6: CNN inference

ID	Input	Latency (ms)	Busy (%)	System Power (W)
$S1_{CPU}$	32x32	25.8	1	783
	224x224	34.6	4	776
$S1_{GPU}$	32x32	45.4	1	905
	224x224	45.3	1	894
$S3_{GPU}$	224x224	29.1	1	—
$S4_{GPU}$		29.8	1	—
$S5_{GPU}$		38.7	1	—
$S6_{GPU}$		36.5	1	—
$D1_{CPU}$	224x224	27.1	15	52
$D1_{GPU}$		**[21.6]**	8	73
$D2_{CPU}$		—	—	—
$D2_{GPU}$		—	—	—
$E1_{CPU}$	224x224	229.3	41	4
$E1_{GPU}$		189.5	—	3

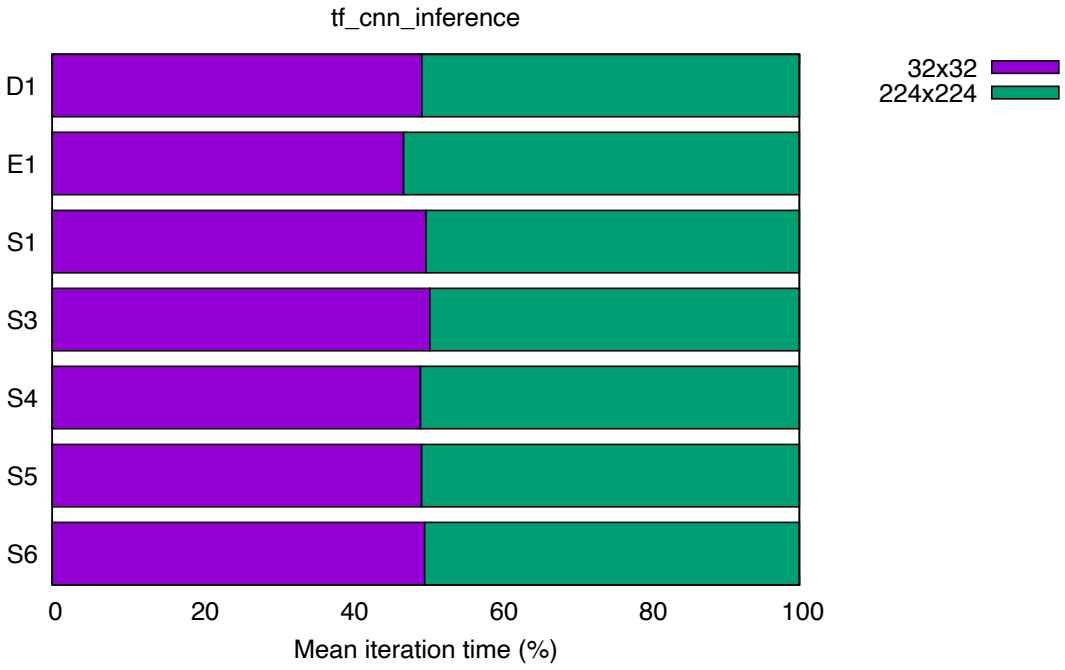


Figure 7.9: CNN inference iteration time distribution

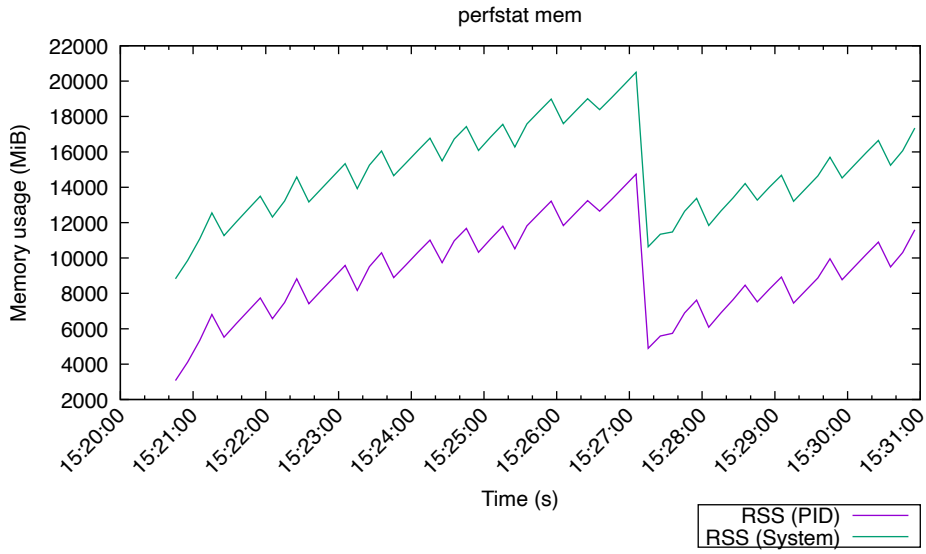


Figure 7.10: S1$_{CPU}$ RAM stats

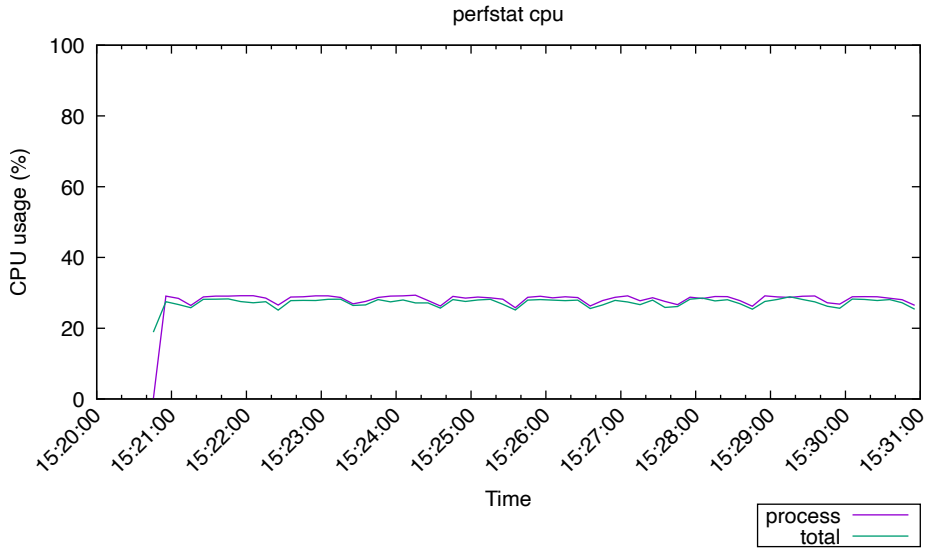


Figure 7.11: S1$_{CPU}$ CPU stats

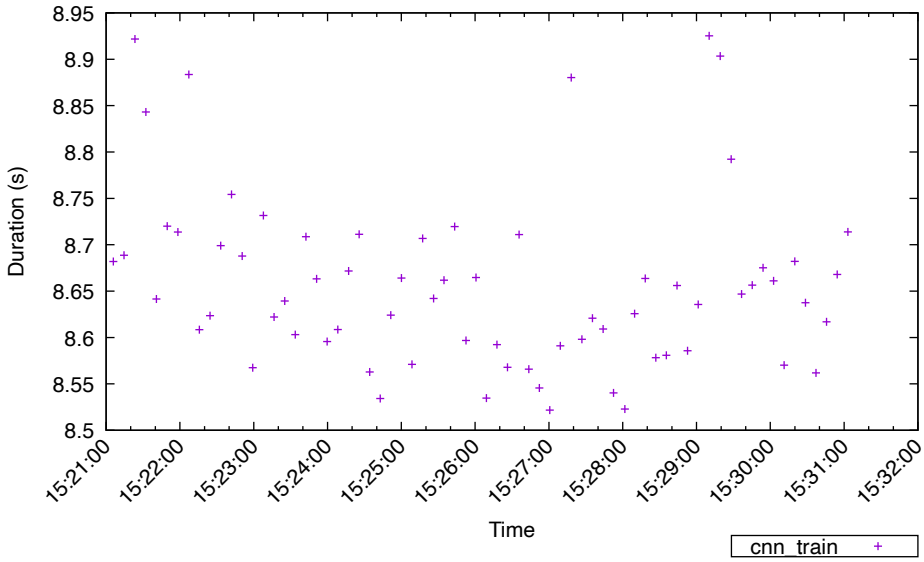


Figure 7.12: $S1_{CPU}$ benchmark iterations

Table 7.7: CNN training

ID	Input	Latency (s)	Busy (%)	System Power (W)
$S1_{CPU}$	32x32	8.7	27	868
$S1_{GPU}$	32x32	[**5.3**]	31	919
$S3_{GPU}$	32x32	125.9	24	—
$S4_{GPU}$		9.2	19	—
$S5_{GPU}$		8.2	20	—
$S6_{GPU}$		6.9	20	—
$D1_{CPU}$	32x32	13.1	100	72
$D1_{GPU}$		8.3	14	59
$D2_{CPU}$		11.4	—	13
$D2_{GPU}$		9.2	—	11
$E1_{CPU}$	32x32	—	—	—
$E1_{GPU}$		—	—	—

Table 7.8: YOLOv3 inference

ID	Variant	Input	Latency (ms)	Busy (%)	System Power (W)
$S1_{CPU}$	Full	416x416	173.8	36	895
	Tiny		36.3	27	838
$S1_{GPU}$	Full		46.6	27	942
	Tiny		10.7	18	923
$S3_{GPU}$	Full	416x416	53.2	36	—
	Tiny		8.9	33	—
$S4_{GPU}$	Full		47.9	25	—
	Tiny		9.4	20	—
$S5_{GPU}$	Full		34.6	37	—
	Tiny		6.8	25	—
$S6_{GPU}$	Full		45.7	18	—
	Tiny		7.9	19	—
$S7_{FPGA}$	Full		[**0.9**]	—	—
	Tiny		[**0.9**]	—	—
$D1_{CPU}$	Full	416x416	265.9	66	70
	Tiny		39.6	51	63
$D1_{GPU}$	Full		32.7	8	131
	Tiny		6.3	8	122
$D2_{CPU}$	Full		401.6	—	13
	Tiny		55.1	—	14
$D2_{GPU}$	Full		1214.9	—	12
	Tiny		8.9	—	12
$E1_{CPU}$	Full	416x416	3877.8	81	6
	Tiny		429.9	78	6
$E1_{GPU}$	Full		1442.7	—	7
	Tiny		185.0	33	7

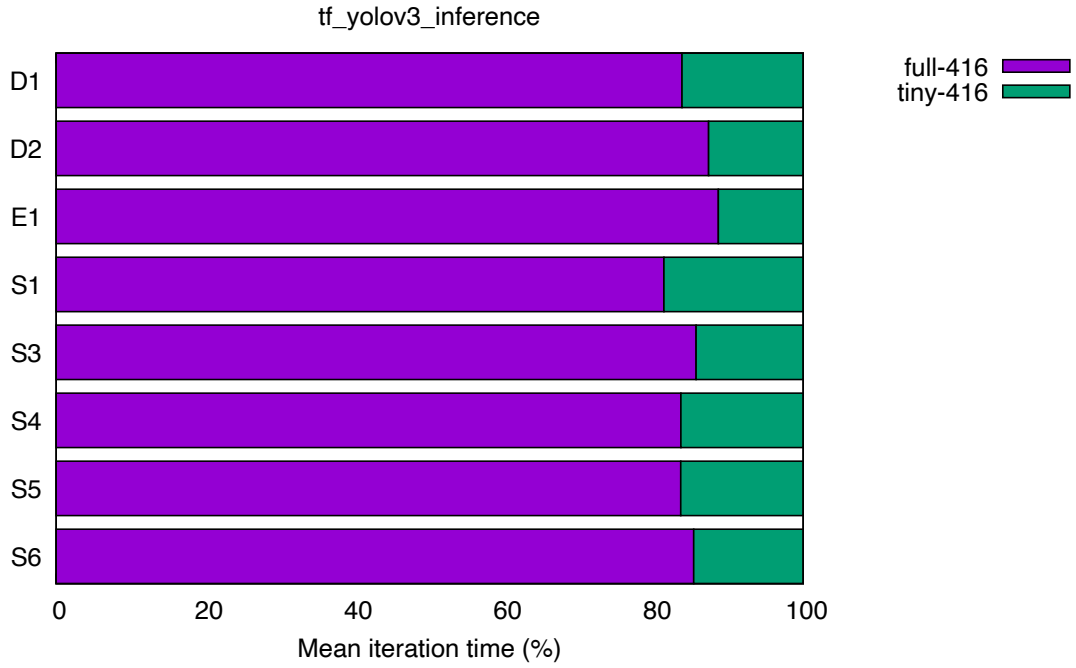


Figure 7.13: YOLOv3 inference iteration time distribution

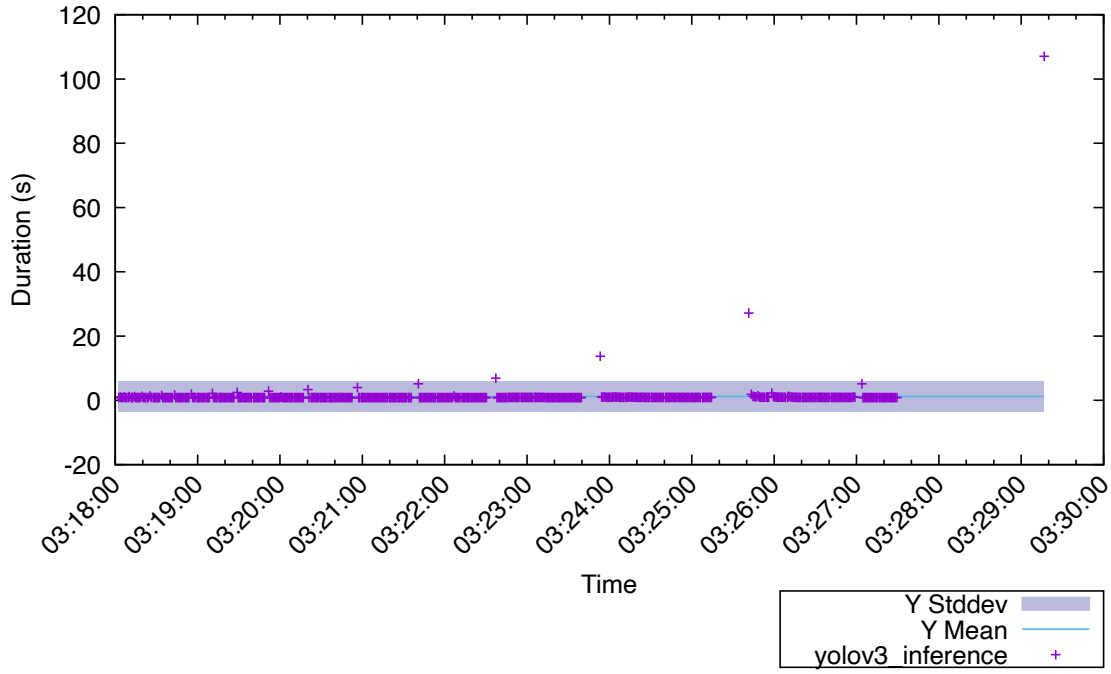


Figure 7.14: $D2_{GPU}$ YOLOv3 benchmark iterations

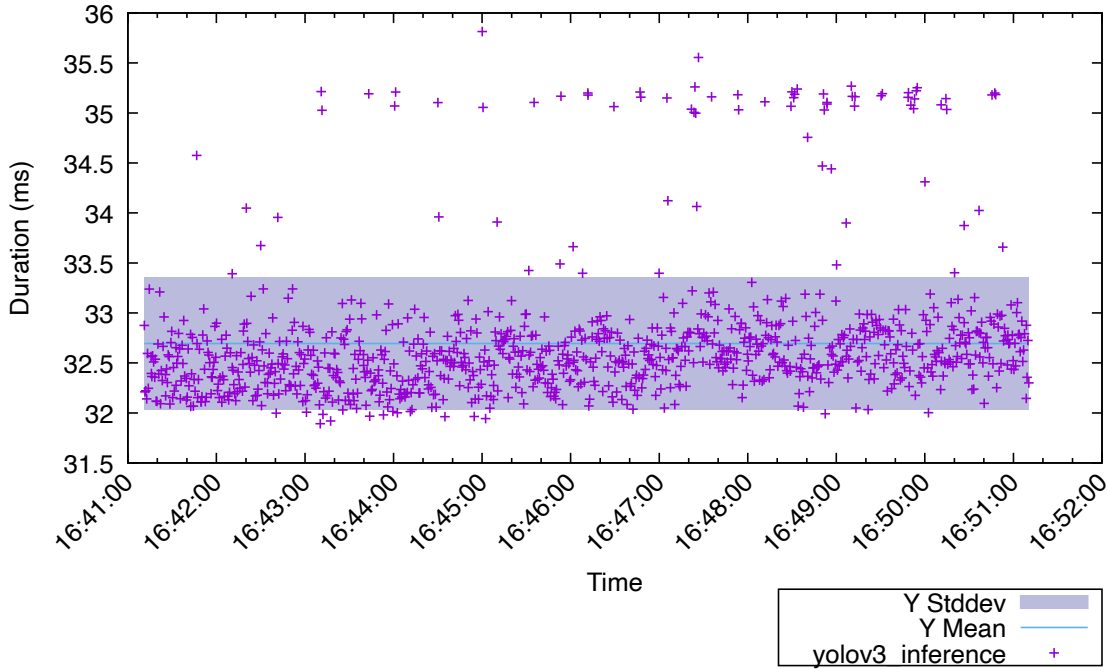


Figure 7.15: $D1_{GPU}$ YOLOv3 benchmark iterations

8 Closing thoughts

In this chapter, we summarize the results of our experiments and strive to answer the initial research questions. We infer on the performance, efficiency and effectiveness of the heterogenous systems for neural network applications based on our results. Afterwards, we touch on lessons learned during this work and state why working with accelerators is still not as easy as running code on the CPU.

8.1 Conclusions

To reiterate, our initial research questions from Section 1.2 were:

1. which platform is the most performant in terms of iterations per time?
2. which platform is the most efficient in terms of iterations per energy?
3. which platform is the most cost-effective in terms of iterations per money?

We now attempt to answer these questions based on our experimentation results. For convolutional neural networks, the Apple M1 SoC tested by us dominates the microbenchmark results, even crushing big and expensive GPU servers. In real-world applications such as YOLOv3 network inference, the Apple system is still able to compete with the more powerful systems for the reduced variant, YOLOv3-tiny. With the full model, the Apple system falls behind. Looking at the neural network training benchmarks, we see the GPU server class systems beat the desktop class systems by a small margin. This may be related to the small input image size though as we alluded to before. Unfortunately, we were only able to test the FPGA system S7 once in the YOLOv3 inference benchmark due to time constraints and FPGA programming complexity. However, the system dominated this benchmark effortlessly for all model sizes. This hints at FPGAs being the most capable accelerators and S7 the best performing heterogenous system. Thus, we answer research **question 1** as follows. System S7 with its FPGA platform is the most performant in our experiments. For microbenchmarks, where FPGA performance data is missing, the Apple M1 SoC achieves the best raw performance.

The next research goal involved finding out which platform performs the most efficient. While we could not monitor the power usage of systems S3 through S6, they are similar in architecture to S1. Thus, we believe system S1 is representative for GPU server class power usage metrics. Unfortunately, it was not possible to monitor system S7 using a dedicated, independent measurement device either. We strongly believe the FPGA platform would outperform any GPU-based platform for neural network applications. This claim is supported by other research as explored in Section 3.2.3. Going by the data we collected in this work, we answer **question 2** as follows. The Apple M1 platform is the most efficient in our benchmarks. For applications like YOLOv3-tiny inference, it even beats the FPGA platform since it performs 10x worse, but draws much less than 10% of the power of the fully assembled FPGA server. Although we do not have the power draw data for system S7, the Intel CPU alone uses more power than the Apple M1 system we tested.

Research **question 3** is easy to answer. The Apple M1 system has no competition at a price point of approx. 1500 € for the Mac Mini 2020 we tested. No other system we tested comes close to its price/performance ratio for convolutional neural network applications, going by our benchmark results. Unfortunately, there is no server class hardware available from Apple, so the answer is a different one if we restrict ourselves to the enterprise area. Here, GPUs win over CPUs and FPGAs should win over GPUs as indicated from other research we cited in Section 3.2.

8.2 Lessons learned

Throughout this work, we faced numerous challenges. For example, making the TensorFlow stack run with hardware acceleration on the various test systems proved to be rather intimidating. On heterogenous systems with Nvidia GPU accelerators, it involves not only matching driver and CUDA stack versions, but additional ecosystem libraries as well. Such libraries include libcudnn, a proprietary library for accelerated neural networks or libcufft, a proprietary library for fast-fourier transforms. These libraries come in a variety of versions, which have to be selected carefully to match the rest of the CUDA stack.

Even for hardware such as the Nvidia Jetson Nano Developer Kit, we ran into software version issues. The software distribution for this board is called "Nvidia Jetpack" and is currently based on Ubuntu 18.04 LTS. With the Jetpack 4.4 *production* release that was available at our time of testing, Nvidia chose to include a *preview* build of their cuDNN library, which is a vital part in TensorFlow neural network acceleration. Nvidia themselves acknowledged performance issues in this preview release of cuDNN [32]. In February 2021, Jetpack 4.5 was released with an updated cuDNN release included. We repeated our benchmarks and compared them to the results we acquired on the Jetpack 4.4 version. Interestingly, there was little to no difference in the results. Still, including *preview* software in a *production* OS release looks like a strange decision.

Running the YOLOv3 benchmarks on the IMS FPGA machine required converting pretrained network weights into another format using the Intel OpenVino tool stack. This is often the case with such accelerators. Instead of directly adding an execution backend to frameworks like TensorFlow, many silicon vendors build their own proprietary formats and offer conversion tools. While this allowed us to easily run the YOLOv3 network variants on the FPGA hardware, it was a hindrance when it came to implementing microbenchmarks or other algorithms. Oftentimes this requires diving into the silicon vendors' specific abstraction stack and directly programming the accelerator. Vendors such as Intel at least provide the heterogenous and open-source OpenCL API for most of their accelerators (CPU, GPU, FPGA, Myriad, etc). This way, portable algorithm implementations are possible, albeit way more complex than using high-level abstractions like TensorFlow primitives. Many vendors (e.g. Maxeler) only offer their vendor specific programming environments. This mandates a large amount of domain specific expertise and can ultimately lead to vendor lock-in situations.

Another problem with accelerators is hardware monitoring. Most vendors offer hardware performance counters, e.g. Nvidia, AMD and Intel. Unfortunately, there is no vendor-independent interface definition for performance counters. The metrics can only be read by vendor specific tools, such as nvidia-smi for Nvidia GPUs or rocm-smi for AMD GPUs. Oftentimes, various vendor specific performance metrics cannot be understood without access to documentation that is available under NDA only. The author has multiple years of experience in working with Intel, Qualcomm and other SoC vendors, so this came as no surprise. It makes comparing hardware monitoring logs hard between accelerators from different manufacturers though. The broad term "GPU usage" can refer to different things for different vendors. The same applies for power usage reporting tools. Oftentimes, it is not clear whether a tool reports the total board power of an accelerator or just the processor core power usage. The former would include the consumption of any memory installed on the chip as well, for example. Thus, the only reliable way to compare power usage between systems is to measure it externally, the way did in this work.

[32] `https://forums.developer.nvidia.com/t/darknet-slower-using-jetpack-4-4-cudnn-8-0-0-cuda-10-2-than-jetpack-4-3-cudnn-7-6-3-cuda-10-0/121579/16`

A Appendix

A.1 Results

The full result data is provided on external storage media along with this work. All of the recorded data together with generated plots and statistics comes in at roughly 50 GB. Thus, adding the data to the thesis itself was not feasible.

Table A.1: tf-conv2d

ID	Variant	Time (s)	CPU (%)	GPU (%)	Power (W)
$D1_{ALL}$	100x100x1-3x3x1	5e-05	8	39	67
$D1_{CPU}$	100x100x1-3x3x1	0.00065	8	—	52
$D1_{CPU}$	100x100x1-5x5x1	0.00049	15	—	62
$D1_{ALL}$	100x100x3-3x3x3	6e-05	8	39	66
$D1_{ALL}$	100x100x1-5x5x1	6e-05	8	38	66
$D1_{CPU}$	100x100x3-3x3x3	0.00046	15	—	60
$D1_{CPU}$	100x100x3-5x5x3	0.00043	31	—	69
$D1_{ALL}$	1000x1000x1-3x3x1	0.00044	8	98	131
$D1_{ALL}$	1000x1000x3-3x3x3	0.00077	8	98	125
$D1_{ALL}$	3000x3000x1-3x3x1	0.00402	8	98	132
$D1_{CPU}$	3000x3000x3-3x3x3	0.16202	98	—	72
$D1_{ALL}$	1000x1000x1-5x5x1	0.00061	8	98	128
$D1_{ALL}$	100x100x3-5x5x3	6e-05	8	39	108
$D1_{ALL}$	1000x1000x3-5x5x3	0.00129	8	98	126
$D1_{CPU}$	1000x1000x1-3x3x1	0.01353	96	—	69
$D1_{CPU}$	1000x1000x3-3x3x3	0.01795	96	—	70
$D1_{CPU}$	1000x1000x1-5x5x1	0.01947	95	—	70
$D1_{CPU}$	1000x1000x3-5x5x3	0.02654	98	—	71
$D1_{ALL}$	3000x3000x3-3x3x3	0.00691	8	98	126
$D1_{ALL}$	3000x3000x1-5x5x1	0.00548	8	97	125
$D1_{ALL}$	3000x3000x3-5x5x3	0.01158	8	97	129
$D1_{CPU}$	3000x3000x1-3x3x1	0.12193	98	—	70
$D1_{CPU}$	3000x3000x1-5x5x1	0.173	99	—	70
$D1_{CPU}$	3000x3000x3-5x5x3	0.24071	99	—	73
$D2_{ALL}$	100x100x1-3x3x1	0.00037	—	—	12
$D2_{ALL}$	100x100x1-5x5x1	0.00037	—	—	13
$D2_{ALL}$	100x100x3-3x3x3	0.00037	—	—	12
$D2_{ALL}$	100x100x3-5x5x3	0.00037	—	—	12
$D2_{CPU}$	100x100x1-3x3x1	0.00014	—	—	11
$D2_{CPU}$	100x100x3-3x3x3	0.00015	—	—	11
$D2_{CPU}$	100x100x1-5x5x1	0.00014	—	—	11
$D2_{CPU}$	1000x1000x1-3x3x1	0.00108	—	—	12
$D2_{CPU}$	1000x1000x3-3x3x3	0.00306	—	—	14
$D2_{CPU}$	3000x3000x1-3x3x1	0.01354	—	—	13
$D2_{ALL}$	1000x1000x1-3x3x1	0.00037	—	—	12
$D2_{ALL}$	3000x3000x1-3x3x1	0.00037	—	—	12
$D2_{ALL}$	1000x1000x1-5x5x1	0.00037	—	—	13
$D2_{ALL}$	3000x3000x1-5x5x1	0.00037	—	—	12
$D2_{ALL}$	1000x1000x3-3x3x3	0.00037	—	—	12

$D2_{ALL}$	3000x3000x3-3x3x3	0.00037	—	—	13
$D2_{ALL}$	1000x1000x3-5x5x3	0.00037	—	—	12
$D2_{CPU}$	1000x1000x1-5x5x1	0.00142	—	—	12
$D2_{CPU}$	3000x3000x3-3x3x3	0.0327	—	—	13
$D2_{CPU}$	3000x3000x1-5x5x1	0.01645	—	—	13
$D2_{CPU}$	100x100x3-5x5x3	0.00017	—	—	11
$D2_{CPU}$	1000x1000x3-5x5x3	0.00339	—	—	17
$D2_{ALL}$	3000x3000x3-5x5x3	0.00037	—	—	12
$D2_{CPU}$	3000x3000x3-5x5x3	0.03486	—	—	15
$E1_{ALL}$	100x100x1-3x3x1	0.0004	29	—	4
$E1_{ALL}$	100x100x1-5x5x1	0.00037	31	—	4
$E1_{ALL}$	100x100x3-3x3x3	0.00043	31	—	5
$E1_{ALL}$	100x100x3-5x5x3	0.00042	32	—	6
$E1_{CPU}$	100x100x1-3x3x1	0.00254	49	—	4
$E1_{ALL}$	1000x1000x1-3x3x1	0.00823	28	—	7
$E1_{ALL}$	1000x1000x3-3x3x3	0.02713	28	—	7
$E1_{CPU}$	1000x1000x3-3x3x3	0.18058	99	—	6
$E1_{ALL}$	3000x3000x1-3x3x1	0.07643	28	—	7
$E1_{CPU}$	3000x3000x1-3x3x1	1.11276	99	—	6
$E1_{CPU}$	100x100x1-5x5x1	0.00237	90	—	5
$E1_{CPU}$	1000x1000x1-5x5x1	0.20401	99	—	6
$E1_{CPU}$	100x100x3-3x3x3	0.00283	67	—	5
$E1_{CPU}$	100x100x3-5x5x3	0.00443	88	—	6
$E1_{CPU}$	1000x1000x1-3x3x1	0.12536	98	—	6
$E1_{ALL}$	1000x1000x1-5x5x1	0.01509	28	—	7
$E1_{CPU}$	1000x1000x3-5x5x3	0.37815	99	—	6
$E1_{CPU}$	3000x3000x1-5x5x1	1.90908	99	—	6
$E1_{ALL}$	3000x3000x3-3x3x3	0.25461	28	—	7
$E1_{ALL}$	3000x3000x1-5x5x1	0.13962	28	—	7
$E1_{ALL}$	1000x1000x3-5x5x3	0.04367	28	—	7
$E1_{ALL}$	3000x3000x3-5x5x3	0.40449	28	—	7
$E1_{CPU}$	3000x3000x3-3x3x3	1.68961	99	—	6
$E1_{CPU}$	3000x3000x3-5x5x3	3.2552	99	—	6
$S1_{ALL}$	100x100x1-3x3x1	9e-05	1	18	912
$S1_{ALL}$	100x100x1-5x5x1	9e-05	1	19	904
$S1_{ALL}$	100x100x3-3x3x3	0.0001	1	23	908
$S1_{ALL}$	100x100x3-5x5x3	0.00011	1	26	910
$S1_{CPU}$	100x100x1-3x3x1	0.00091	1	0	794
$S1_{CPU}$	100x100x3-3x3x3	0.00063	2	—	778
$S1_{CPU}$	100x100x1-5x5x1	0.00069	2	—	787
$S1_{CPU}$	1000x1000x1-3x3x1	0.00475	60	—	908
$S1_{CPU}$	1000x1000x3-3x3x3	0.00793	64	—	895
$S1_{CPU}$	3000x3000x1-3x3x1	0.03057	80	—	946
$S1_{CPU}$	1000x1000x1-5x5x1	0.00716	66	—	917
$S1_{CPU}$	1000x1000x3-5x5x3	0.00976	73	—	920
$S1_{CPU}$	3000x3000x1-5x5x1	0.04023	84	—	950
$S1_{CPU}$	100x100x3-5x5x3	0.00048	5	—	798
$S1_{ALL}$	1000x1000x1-3x3x1	0.00015	1	98	1145
$S1_{ALL}$	3000x3000x1-3x3x1	0.00139	1	98	1170
$S1_{ALL}$	1000x1000x1-5x5x1	0.00021	1	98	1187
$S1_{ALL}$	3000x3000x3-3x3x3	0.0023	1	98	1211

$S1_{ALL}$	3000x3000x1-5x5x1	0.0019	1	98	1205
$S1_{ALL}$	1000x1000x3-3x3x3	0.00025	1	98	1178
$S1_{ALL}$	1000x1000x3-5x5x3	0.00045	1	98	1199
$S1_{ALL}$	3000x3000x3-5x5x3	0.00394	1	98	1235
$S1_{CPU}$	3000x3000x3-3x3x3	0.03946	84	—	957
$S1_{CPU}$	3000x3000x3-5x5x3	0.06007	84	—	1007
$S3_{ALL}$	100x100x1-3x3x1	8e-05	3	22	—
$S3_{ALL}$	100x100x1-5x5x1	0.00011	2	19	—
$S3_{ALL}$	100x100x3-3x3x3	0.00011	2	24	—
$S3_{ALL}$	100x100x3-5x5x3	9e-05	2	37	—
$S3_{ALL}$	3000x3000x3-3x3x3	0.00623	2	98	—
$S3_{ALL}$	1000x1000x1-3x3x1	0.00044	2	97	—
$S3_{ALL}$	1000x1000x3-3x3x3	0.0007	2	98	—
$S3_{ALL}$	1000x1000x1-5x5x1	0.00057	2	98	—
$S3_{ALL}$	1000x1000x3-5x5x3	0.00111	2	98	—
$S3_{ALL}$	3000x3000x1-3x3x1	0.00391	2	98	—
$S3_{ALL}$	3000x3000x1-5x5x1	0.00514	2	98	—
$S3_{ALL}$	3000x3000x3-5x5x3	0.00995	2	98	—
$S4_{ALL}$	100x100x1-3x3x1	9e-05	2	19	—
$S4_{ALL}$	100x100x1-5x5x1	9e-05	1	20	—
$S4_{ALL}$	1000x1000x1-3x3x1	0.00031	2	98	—
$S4_{ALL}$	1000x1000x1-5x5x1	0.00046	2	98	—
$S4_{ALL}$	3000x3000x3-3x3x3	0.00546	2	97	—
$S4_{ALL}$	3000x3000x1-5x5x1	0.00431	2	98	—
$S4_{ALL}$	100x100x3-3x3x3	0.00012	1	20	—
$S4_{ALL}$	100x100x3-5x5x3	9e-05	2	32	—
$S4_{ALL}$	1000x1000x3-3x3x3	0.00051	2	97	—
$S4_{ALL}$	1000x1000x3-5x5x3	0.00103	2	98	—
$S4_{ALL}$	3000x3000x1-3x3x1	0.00291	2	98	—
$S4_{ALL}$	3000x3000x3-5x5x3	0.00947	2	98	—
$S5_{ALL}$	100x100x1-3x3x1	5e-05	3	31	—
$S5_{ALL}$	100x100x3-3x3x3	6e-05	2	35	—
$S5_{ALL}$	100x100x1-5x5x1	7e-05	3	26	—
$S5_{ALL}$	1000x1000x1-3x3x1	0.00016	2	97	—
$S5_{ALL}$	1000x1000x3-3x3x3	0.00026	2	97	—
$S5_{ALL}$	3000x3000x1-3x3x1	0.0014	1	97	—
$S5_{ALL}$	1000x1000x1-5x5x1	0.00021	1	97	—
$S5_{ALL}$	100x100x3-5x5x3	6e-05	3	38	—
$S5_{ALL}$	1000x1000x3-5x5x3	0.00045	2	97	—
$S5_{ALL}$	3000x3000x3-3x3x3	0.0023	1	97	—
$S5_{ALL}$	3000x3000x1-5x5x1	0.00191	1	97	—
$S5_{ALL}$	3000x3000x3-5x5x3	0.00395	1	97	—
$S6_{ALL}$	100x100x1-3x3x1	7e-05	0	19	—
$S6_{ALL}$	100x100x1-5x5x1	7e-05	0	23	—
$S6_{ALL}$	100x100x3-3x3x3	7e-05	0	31	—
$S6_{ALL}$	100x100x3-5x5x3	7e-05	0	39	—
$S6_{ALL}$	1000x1000x1-3x3x1	7e-05	0	45	—
$S6_{ALL}$	1000x1000x3-3x3x3	0.0001	0	98	—
$S6_{ALL}$	3000x3000x1-3x3x1	0.00024	0	98	—
$S6_{ALL}$	1000x1000x1-5x5x1	7e-05	0	78	—
$S6_{ALL}$	1000x1000x3-5x5x3	0.00014	0	98	—

$S6_{ALL}$	3000x3000x3-3x3x3	0.00086	0	98	—
$S6_{ALL}$	3000x3000x1-5x5x1	0.00044	0	98	—
$S6_{ALL}$	3000x3000x3-5x5x3	0.00123	0	98	—

Table A.2: tf-conv2d-dwise

ID	Variant	Time (s)	CPU (%)	GPU (%)	Power (W)
$D1_{ALL}$	100x100x3-3x3x3	0.00012	—	5	87
$D1_{ALL}$	100x100x3-5x5x3	0.00012	—	9	83
$D1_{CPU}$	100x100x3-3x3x3	0.00107	—	—	58
$D1_{CPU}$	100x100x3-5x5x3	0.00279	—	—	59
$D1_{ALL}$	1000x1000x3-3x3x3	0.00017	—	98	126
$D1_{ALL}$	3000x3000x3-3x3x3	0.00179	—	98	130
$D1_{CPU}$	1000x1000x3-3x3x3	0.02237	—	—	69
$D1_{ALL}$	1000x1000x3-5x5x3	0.00044	—	98	125
$D1_{CPU}$	3000x3000x3-3x3x3	0.20519	—	0	69
$D1_{CPU}$	1000x1000x3-5x5x3	0.05805	—	—	69
$D1_{ALL}$	3000x3000x3-5x5x3	0.00413	—	98	130
$D1_{CPU}$	3000x3000x3-5x5x3	0.51513	—	—	69
$D2_{ALL}$	100x100x3-3x3x3	0.00072	—	—	12
$D2_{CPU}$	100x100x3-3x3x3	0.0002	—	—	11
$D2_{CPU}$	100x100x3-5x5x3	0.00022	—	—	11
$D2_{CPU}$	1000x1000x3-3x3x3	0.00459	—	—	13
$D2_{ALL}$	100x100x3-5x5x3	0.00072	—	—	12
$D2_{ALL}$	1000x1000x3-3x3x3	0.00072	—	—	12
$D2_{CPU}$	3000x3000x3-3x3x3	0.04486	—	—	13
$D2_{ALL}$	1000x1000x3-5x5x3	0.00072	—	—	12
$D2_{CPU}$	1000x1000x3-5x5x3	0.00621	—	—	13
$D2_{ALL}$	3000x3000x3-3x3x3	0.00071	—	—	12
$D2_{CPU}$	3000x3000x3-5x5x3	0.0597	—	—	13
$D2_{ALL}$	3000x3000x3-5x5x3	0.00071	—	—	12
$E1_{ALL}$	100x100x3-3x3x3	0.00087	—	—	5
$E1_{ALL}$	100x100x3-5x5x3	0.00081	—	—	5
$E1_{CPU}$	100x100x3-3x3x3	0.00472	—	—	5
$E1_{CPU}$	100x100x3-5x5x3	0.01002	—	—	4
$E1_{CPU}$	1000x1000x3-3x3x3	0.13268	—	—	6
$E1_{CPU}$	3000x3000x3-3x3x3	1.19578	—	—	6
$E1_{CPU}$	1000x1000x3-5x5x3	0.30221	—	—	6
$E1_{ALL}$	1000x1000x3-3x3x3	0.01467	—	—	7
$E1_{ALL}$	3000x3000x3-3x3x3	0.14098	—	—	7
$E1_{ALL}$	1000x1000x3-5x5x3	0.03869	—	—	7
$E1_{ALL}$	3000x3000x3-5x5x3	0.359	—	—	7
$E1_{CPU}$	3000x3000x3-5x5x3	2.75709	—	—	6
$S1_{ALL}$	100x100x3-3x3x3	0.00018	—	2	915
$S1_{CPU}$	100x100x3-3x3x3	0.00142	—	0	787
$S1_{CPU}$	100x100x3-5x5x3	0.00352	—	—	781
$S1_{CPU}$	1000x1000x3-3x3x3	0.00498	—	—	922
$S1_{CPU}$	3000x3000x3-3x3x3	0.0446	—	—	948
$S1_{CPU}$	1000x1000x3-5x5x3	0.01234	—	—	942

$S1_{ALL}$	100x100x3-5x5x3	0.0002	—	4	915
$S1_{ALL}$	1000x1000x3-3x3x3	0.00018	—	34	1023
$S1_{ALL}$	1000x1000x3-5x5x3	0.00018	—	70	1158
$S1_{ALL}$	3000x3000x3-3x3x3	0.00058	—	97	1256
$S1_{CPU}$	3000x3000x3-5x5x3	0.10628	—	—	970
$S1_{ALL}$	3000x3000x3-5x5x3	0.00125	—	97	1246
$S3_{ALL}$	100x100x3-3x3x3	0.0002	—	3	—
$S3_{ALL}$	100x100x3-5x5x3	0.0002	—	7	—
$S3_{ALL}$	1000x1000x3-3x3x3	0.00018	—	97	—
$S3_{ALL}$	1000x1000x3-5x5x3	0.00042	—	97	—
$S4_{ALL}$	100x100x3-3x3x3	0.00019	—	2	—
$S4_{ALL}$	3000x3000x3-3x3x3	0.00174	—	97	—
$S4_{ALL}$	100x100x3-5x5x3	0.00018	—	4	—
$S4_{ALL}$	1000x1000x3-3x3x3	0.00018	—	40	—
$S4_{ALL}$	1000x1000x3-5x5x3	0.00035	—	97	—
$S4_{ALL}$	3000x3000x3-5x5x3	0.00355	—	98	—
$S5_{ALL}$	100x100x3-3x3x3	0.00012	—	4	—
$S5_{ALL}$	100x100x3-5x5x3	0.00012	—	7	—
$S5_{ALL}$	1000x1000x3-3x3x3	0.00012	—	49	—
$S5_{ALL}$	3000x3000x3-3x3x3	0.00059	—	97	—
$S5_{ALL}$	1000x1000x3-5x5x3	0.00013	—	97	—
$S5_{ALL}$	3000x3000x3-5x5x3	0.00126	—	98	—
$S6_{ALL}$	100x100x3-3x3x3	0.00012	—	5	—
$S6_{ALL}$	100x100x3-5x5x3	0.00012	—	8	—
$S6_{ALL}$	1000x1000x3-3x3x3	0.00012	—	39	—
$S6_{ALL}$	3000x3000x3-3x3x3	0.00039	—	97	—
$S6_{ALL}$	1000x1000x3-5x5x3	0.00012	—	86	—
$S6_{ALL}$	3000x3000x3-5x5x3	0.00098	—	97	—

Table A.3: tf-max-pool2d

ID	Variant	Time (s)	CPU (%)	GPU (%)	Power (W)
$D1_{ALL}$	100x100-3	3e-05	8	16	73
$D1_{CPU}$	100x100-3	0.00042	8	—	48
$D1_{CPU}$	100x100-5	0.00091	8	—	51
$D1_{ALL}$	100x100-5	3e-05	8	28	65
$D1_{ALL}$	1000x1000-3	0.00012	8	98	122
$D1_{ALL}$	1000x1000-5	0.00022	8	98	128
$D1_{CPU}$	1000x1000-3	0.04099	8	—	49
$D1_{CPU}$	1000x1000-5	0.09254	8	—	49
$D1_{ALL}$	3000x3000-3	0.00113	8	98	133
$D1_{CPU}$	3000x3000-3	0.38979	8	0	51
$D1_{ALL}$	3000x3000-5	0.002	8	97	128
$D1_{CPU}$	3000x3000-5	0.84608	8	—	50
$D2_{CPU}$	100x100-3	0.00014	—	—	11
$D2_{CPU}$	100x100-5	0.00139	—	—	16
$D2_{CPU}$	1000x1000-3	0.00276	—	—	15
$D2_{ALL}$	100x100-3	0.00029	—	—	12
$D2_{ALL}$	100x100-5	0.00029	—	—	12

$D2_{ALL}$	1000x1000-3	0.00029	—	—	12
$D2_{ALL}$	1000x1000-5	0.00029	—	—	12
$D2_{CPU}$	1000x1000-5	0.1263	—	—	18
$D2_{ALL}$	3000x3000-3	0.00029	—	—	12
$D2_{CPU}$	3000x3000-3	0.0264	—	—	15
$D2_{ALL}$	3000x3000-5	0.00029	—	—	12
$D2_{CPU}$	3000x3000-5	1.14141	—	—	18
$E1_{ALL}$	100x100-3	0.00018	30	—	4
$E1_{CPU}$	100x100-3	0.0028	27	—	3
$E1_{ALL}$	100x100-5	0.00018	30	—	6
$E1_{CPU}$	100x100-5	0.00627	27	—	3
$E1_{CPU}$	1000x1000-3	0.28908	26	—	3
$E1_{ALL}$	1000x1000-3	0.01093	29	—	6
$E1_{ALL}$	1000x1000-5	0.01871	31	—	7
$E1_{CPU}$	1000x1000-5	0.64538	27	—	3
$E1_{ALL}$	3000x3000-3	0.09898	32	—	7
$E1_{CPU}$	3000x3000-3	2.64793	26	—	3
$E1_{ALL}$	3000x3000-5	0.16747	29	—	7
$E1_{CPU}$	3000x3000-5	5.88563	25	—	3
$S1_{CPU}$	100x100-3	0.00055	1	—	786
$S1_{CPU}$	100x100-5	0.00127	1	—	783
$S1_{CPU}$	1000x1000-3	0.05205	1	0	790
$S1_{ALL}$	100x100-3	5e-05	1	8	926
$S1_{ALL}$	100x100-5	5e-05	1	12	905
$S1_{ALL}$	1000x1000-3	5e-05	1	66	1162
$S1_{ALL}$	1000x1000-5	6e-05	1	98	1229
$S1_{CPU}$	1000x1000-5	0.12708	1	—	786
$S1_{ALL}$	3000x3000-3	0.00034	1	98	1234
$S1_{CPU}$	3000x3000-3	0.51154	1	—	786
$S1_{ALL}$	3000x3000-5	0.00061	1	98	1227
$S1_{CPU}$	3000x3000-5	1.18815	1	—	787
$S3_{ALL}$	100x100-3	5e-05	2	9	—
$S3_{ALL}$	100x100-5	5e-05	2	17	—
$S3_{ALL}$	1000x1000-3	0.00012	2	98	—
$S3_{ALL}$	1000x1000-5	0.00021	2	98	—
$S3_{ALL}$	3000x3000-3	0.00118	2	98	—
$S3_{ALL}$	3000x3000-5	0.00197	2	98	—
$S4_{ALL}$	100x100-3	4e-05	2	8	—
$S4_{ALL}$	100x100-5	5e-05	2	13	—
$S4_{ALL}$	1000x1000-3	0.00011	2	98	—
$S4_{ALL}$	1000x1000-5	0.0002	2	97	—
$S4_{ALL}$	3000x3000-3	0.00087	2	98	—
$S4_{ALL}$	3000x3000-5	0.0019	2	98	—
$S5_{ALL}$	100x100-3	3e-05	2	14	—
$S5_{ALL}$	100x100-5	3e-05	6	22	—
$S5_{ALL}$	1000x1000-3	4e-05	6	97	—
$S5_{ALL}$	1000x1000-5	7e-05	5	97	—
$S5_{ALL}$	3000x3000-3	0.00035	6	97	—
$S5_{ALL}$	3000x3000-5	0.00062	2	97	—
$S6_{ALL}$	100x100-3	3e-05	0	18	—
$S6_{ALL}$	100x100-5	3e-05	0	26	—

$S6_{ALL}$	1000x1000-3	3e-05	0	77	—
$S6_{ALL}$	1000x1000-5	5e-05	0	98	—
$S6_{ALL}$	3000x3000-3	0.00023	0	97	—
$S6_{ALL}$	3000x3000-5	0.00044	0	98	—

Table A.4: tf-relu

ID	Variant	Time (s)	CPU (%)	GPU (%)	Power (W)
$D1_{ALL}$	100x100	1e-05	8	17	78
$D1_{CPU}$	100x100	1e-05	8	—	51
$D1_{CPU}$	1000x1000	0.00111	86	—	62
$D1_{ALL}$	1000x1000	6e-05	8	98	125
$D1_{ALL}$	3000x3000	0.00063	8	98	132
$D1_{CPU}$	3000x3000	0.01357	69	—	71
$D2_{ALL}$	100x100	0.00012	—	—	12
$D2_{CPU}$	100x100	4e-05	—	—	11
$D2_{CPU}$	1000x1000	0.00041	—	—	14
$D2_{ALL}$	1000x1000	0.00012	—	—	12
$D2_{ALL}$	3000x3000	0.00012	—	—	12
$D2_{CPU}$	3000x3000	0.00371	—	—	15
$E1_{ALL}$	100x100	0.0001	31	—	4
$E1_{CPU}$	100x100	7e-05	26	—	3
$E1_{ALL}$	1000x1000	0.00198	29	—	7
$E1_{CPU}$	1000x1000	0.00248	89	—	6
$E1_{ALL}$	3000x3000	0.01835	28	—	7
$E1_{CPU}$	3000x3000	0.02877	95	—	6
$S1_{ALL}$	100x100	2e-05	1	9	904
$S1_{CPU}$	100x100	2e-05	1	—	798
$S1_{CPU}$	1000x1000	0.00021	57	0	929
$S1_{ALL}$	1000x1000	2e-05	1	93	1089
$S1_{ALL}$	3000x3000	0.00026	1	97	1082
$S1_{CPU}$	3000x3000	0.01161	52	—	843
$S3_{ALL}$	100x100	2e-05	2	9	—
$S3_{ALL}$	1000x1000	4e-05	2	98	—
$S3_{ALL}$	3000x3000	0.00041	2	97	—
$S4_{ALL}$	100x100	2e-05	2	9	—
$S4_{ALL}$	1000x1000	4e-05	2	98	—
$S4_{ALL}$	3000x3000	0.00036	2	97	—
$S5_{ALL}$	100x100	1e-05	1	16	—
$S5_{ALL}$	1000x1000	3e-05	1	98	—
$S5_{ALL}$	3000x3000	0.00027	1	97	—
$S6_{ALL}$	100x100	1e-05	0	20	—
$S6_{ALL}$	1000x1000	1e-05	0	46	—
$S6_{ALL}$	3000x3000	0.00016	0	98	—

Table A.5: tf-cnn-inference

ID	Variant	Time (s)	CPU (%)	GPU (%)	Power (W)
$D1_{ALL}$	224x224	0.02157	8	3	73
$D1_{CPU}$	224x224	0.02706	15	—	52
$D1_{ALL}$	32x32	0.02111	8	1	72
$D1_{CPU}$	32x32	0.02049	9	0	44
$E1_{ALL}$	224x224	0.1895	26	—	4
$E1_{CPU}$	224x224	0.2293	41	—	4
$E1_{ALL}$	32x32	0.16821	28	—	3
$E1_{CPU}$	32x32	0.16007	28	—	3
$S1_{ALL}$	224x224	0.04531	1	1	894
$S1_{CPU}$	224x224	0.0346	4	0	776
$S1_{ALL}$	32x32	0.04535	1	—	905
$S1_{CPU}$	32x32	0.02578	1	—	783
$S3_{ALL}$	224x224	0.02851	2	2	—
$S3_{ALL}$	32x32	0.02913	2	—	—
$S4_{ALL}$	224x224	0.02982	2	1	—
$S4_{ALL}$	32x32	0.029	2	0	—
$S5_{ALL}$	224x224	0.03866	1	1	—
$S5_{ALL}$	32x32	0.03781	1	0	—
$S6_{ALL}$	224x224	0.03653	0	1	—
$S6_{ALL}$	32x32	0.03631	0	—	—

Table A.6: tf-cnn-train

ID	Variant	Time (s)	CPU (%)	GPU (%)	Power (W)
$D1_{ALL}$	32x32	8.31664	14	24	59
$D1_{CPU}$	32x32	13.14728	64	—	72
$D2_{ALL}$	32x32	9.22301	—	—	11
$D2_{CPU}$	32x32	11.39815	—	—	13
$S1_{ALL}$	32x32	5.34824	2	26	919
$S1_{CPU}$	32x32	8.6568	27	0	868
$S3_{ALL}$	32x32	125.86583	0	2	—
$S4_{ALL}$	32x32	9.21191	2	15	—
$S5_{ALL}$	32x32	8.24562	2	16	—
$S6_{ALL}$	32x32	6.96886	1	17	—

Table A.7: tf-yolov3-inference

ID	Variant	Time (s)	CPU (%)	GPU (%)	Power (W)
$D1_{ALL}$	full-416	0.03269	8	75	131
$D1_{CPU}$	full-416	0.26587	66	0	70
$D1_{ALL}$	tiny-416	0.00633	8	46	122
$D1_{CPU}$	tiny-416	0.03963	51	0	63
$D2_{ALL}$	full-416	1.21497	—	—	12
$D2_{CPU}$	full-416	0.40159	—	—	13
$D2_{ALL}$	tiny-416	0.1763	—	—	12

$D2_{CPU}$	tiny-416	0.05508	—	—	14
$E1_{ALL}$	full-416	1.44267	30	—	7
$E1_{CPU}$	full-416	3.87777	81	—	6
$E1_{ALL}$	tiny-416	0.185	30	—	7
$E1_{CPU}$	tiny-416	0.42994	78	—	6
$S1_{ALL}$	full-416	0.04655	1	26	942
$S1_{CPU}$	full-416	0.17416	36	0	895
$S1_{ALL}$	tiny-416	0.01066	1	16	923
$S1_{CPU}$	tiny-416	0.03633	27	—	838
$S3_{ALL}$	full-416	0.05322	2	34	—
$S3_{ALL}$	tiny-416	0.0089	2	32	—
$S4_{ALL}$	full-416	0.04798	2	26	—
$S4_{ALL}$	tiny-416	0.00939	2	19	—
$S5_{ALL}$	full-416	0.03464	3	35	—
$S5_{ALL}$	tiny-416	0.00679	1	25	—
$S6_{ALL}$	full-416	0.04572	0	17	—
$S6_{ALL}$	tiny-416	0.00786	0	18	—

Bibliography

[1] H. Andrade, L. E. Lwakatare, I. Crnkovic, and J. Bosch. Software challenges in heterogeneous computing: A multiple case study in industry. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 148–155, 2019.

[2] Maneesh Ayi and Mohamed El-Sharkawy. Rmnv2: Reduced mobilenet v2 for cifar10. In *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0287–0292. IEEE, 2020.

[3] S. Bianco, R. Cadene, L. Celona, and P. Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.

[4] Egbert J.W. Boers and Herman Kuiper. Biological metaphors and the design of modular artificial neural networks, 1992.

[5] André Brodtkorb, Christopher Dyken, Trond Hagen, Jon Hjelmervik, and Olaf Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18:1–33, 10 2010.

[6] S. Cass. Taking ai to the edge: Google's tpu now comes in a maker-friendly package. *IEEE Spectrum*, 56(5):16–17, 2019.

[7] S. Chatterjee and S. Sen. Cache-efficient matrix transposition. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*, pages 195–205, 2000.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[9] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 225–236, 2010.

[10] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. Understanding performance differences of fpgas and gpus. pages 93–96, 04 2018.

[11] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, page 6374, New York, NY, USA, 2010. Association for Computing Machinery.

[12] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor, and S. Areibi. Caffeinated fpgas: Fpga framework for convolutional neural networks. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 265–268, 2016.

[13] Pedro Felzenszwalb, David McAllester, and Deva Ramanan. A discriminatively trained, multiscale, deformable part model. In *2008 IEEE conference on computer vision and pattern recognition*, pages 1–8. IEEE, 2008.

[14] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[15] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. Gpgpu processing in cuda architecture. *arXiv preprint arXiv:1202.4347*, 2012.

[16] Daniel Graupe. *Principles of artificial neural networks*, volume 7. World Scientific, 2013.

[17] Tien Ho-Phuoc. Cifar10 to compare visual recognition performance between deep neural networks and humans. *arXiv preprint arXiv:1811.07270*, 2018.

[18] Bodun Hu and Christopher J. Rossbach. Mirovia: A benchmarking suite for modern heterogeneous computing. *CoRR*, abs/1906.10347, 2019.

[19] A. K. Jain, Jianchang Mao, and K. M. Mohiuddin. Artificial neural networks: a tutorial. *Computer*, 29(3):31–44, 1996.

[20] M. Johnson. Superscalar microprocessor design. In *Prentice Hall series in innovative technology*, 1991.

[21] Michael Jones and Paul Viola. Fast multi-view face detection. *Mitsubishi Electric Research Lab TR-20003-96*, 3(14):2, 2003.

[22] A. Karki, C. Palangotu Keshava, S. Mysore Shivakumar, J. Skow, G. Madhukeshwar Hegde, and H. Jeon. Tango: A deep neural network benchmark suite for various accelerators. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 137–138, 2019.

[23] Peter Kogge, S. Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, and Robert Lucas. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Techinal Representative*, 15, 01 2008.

[24] Alex Krizhevsky and Geoff Hinton. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*, 40(7):1–9, 2010.

[25] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908916, July 2003.

[26] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In *Proceedings of*

the 37th Annual International Symposium on Computer Architecture, ISCA '10, page 451460, New York, NY, USA, 2010. Association for Computing Machinery.

[27] Juliane Liepe, Chris Barnes, Erika Cule, Kamil Erguler, Paul Kirk, Tina Toni, and Michael P.H. Stumpf. ABC-SysBio-approximate Bayesian computation in Python with GPU support. *Bioinformatics*, 26(14):1797–1799, 06 2010.

[28] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. *CoRR*, abs/1708.02002, 2017.

[29] Q. Mao, H. Sun, Y. Liu, and R. Jia. Mini-yolov3: Real-time object detector for embedded applications. *IEEE Access*, 7:133529–133538, 2019.

[30] Chris McClanahan. History and evolution of gpu architecture a paper survey. 2011.

[31] Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4), July 2015.

[32] C. Murphy and Y. Fu. Xilinx all programmable devices : A superior platform for compute-intensive systems gpu origins and target workloads. 2015.

[33] Michael A Nielsen. *Neural networks and deep learning*, volume 2018.

[34] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 514, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[36] Kalin Ovtcharov, Olatunji Ruwase, J. Kim, Jeremy Fowers, K. Strauss, and E. Chung. Accelerating deep convolutional neural networks using specialized hardware. 2015.

[37] Abel Paz and Antonio Plaza. A new morphological anomaly detection algorithm for hyperspectral images and its gpu implementation. volume 8157, 09 2011.

[38] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 873880, New York, NY, USA, 2009. Association for Computing Machinery.

[39] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[40] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[41] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[42] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and G. Rodgers. Achieving exascale capabilities through heterogeneous computing. *IEEE Micro*, 35(4):26–36, 2015.

[43] Jacob T. Schwartz. The new connectionism: Developing relationships between neuroscience and artificial intelligence. *Daedalus*, 117(1):123–141, 1988.

[44] Sagar Sharma. Activation functions in neural networks. *Towards Data Science*, 6, 2017.

[45] Alfred Spector and David Gifford. The space shuttle primary computer system. *Commun. ACM*, 27(9):872900, September 1984.

[46] Stephan Spitz. Neue mobiltelefone: Geldbörse und bankfiliale zugleich. *Datenschutz und Datensicherheit-DuD*, 36(3):185–188, 2012.

[47] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '02, page 409, USA, 2002. IEEE Computer Society.

[48] Vignesh Thakkar, Suman Tewary, and Chandan Chakraborty. Batch normalization in convolutional neural networksa comparative study with cifar-10 data. In *2018 fifth international conference on emerging applications of information technology (EAIT)*, pages 1–5. IEEE, 2018.

[49] Dimitrios Tsifakis, Alistair P. Rendell, and Peter E. Strazdins. Cache oblivious matrix transposition: Simulation and experiment. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, pages 17–25, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[50] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 235–246, 2010.

[51] S. Zafeiriou, C. Zhang, and Zhengyou Zhang. A survey on face detection in the wild: Past, present and future. *Comput. Vis. Image Underst.*, 138:1–24, 2015.

[52] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. TBD: benchmarking and analyzing deep neural network training. *CoRR*, abs/1803.06905, 2018.