Heterogeneous system-on-chip for AI computing

Understanding the Nvidia Tegra Xavier architecture for running deep learning inference using TensorRT

Joakim Foss Johansen

Thesis submitted for the degree of
Master in Programming and system architecture
60 credits

Department of Informatics
The Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2022

Heterogeneous system-on-chip for AI computing

Understanding the Nvidia Tegra Xavier architecture for running deep learning inference using TensorRT

Joakim Foss Johansen

Heterogeneous system-on-chip for AI computing

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

In a time where the world is getting more and more automatized we have seen an increase in using heterogeneous systems-on-chip, to solve problems of automation. This has lead to an increased pressure on creating both good hardware and software for problems such as robotics, image classification, speech recognition and more. To explore these issues we have looked into one of these heterogenous systems-on-chip, the Nvidia Tegra Xavier and by getting a better understanding its architecture we have managed to optimize deep neural networks for both inference time and power usage, by utilizing Nvidia frameworks such as TensorRT and CUDA. By taking advantage of these frameworks along with understanding the architecture of the Nvidia Tegra Xavier, we have managed to get major reductions in inference time as well as vast improvements to the power consumption.

Acknowledgments

First and foremost I would like to thank my official supervisor Håkon. I am grateful for all the good advices and friendly conversations. I would also like to thank you for showing a lot of compassion and understanding during some difficult times.

Finally I would like to thank my wife, Priscila for her support and encouragement during these long days of writing and frustration.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

With the increasing use of embedded systems on chips for their use in autonomous machines, such as commercial robots, medical instruments, autonomous vehicles and more. There is a need to understand the architecture of these systems to see where they excel and their limitations. We see that heterogeneous systems on chips have started to replace more traditional microcontrollers in these types of systems, for example, the Tegra X1 in the Nintendo Switch and Exynos in the Samsung Galaxy A series.

For the last decade, the usage of these heterogeneous systems-on-chip has exploded. In the world of AI and automotive cars, we continuously hear of self-driving, lane detection, object detection, etc. However, beneath all of these buzzwords, there are hardware and software designed to optimize the speed, safety and power consumption of such systems. As an example, Nvidia and Audi have gone together to transform the automotive future through innovation, digitalization, electrification and improved sustainability[35]. In collaboration with Audi, Nvidia has developed hardware and software for object detection, signs classification and line detection in cars. Nvidia Drive embedded supercomputing are platforms that process data from cameras, radars and lidar(light detection and ranging) sensors[35]. These are used to perceive the surrounding environment and localize the car to map out a safe path forward, all to make driving as safe and energy-efficient as possible. The hardware used for these processes is different types of Nvidia platforms such as Nvidia DRIVE Hyperion, NVIDIA DRIVE Orion and Nvidia DRIVE AGX Xavier. This platform incorporates different types of Nvidia Systems-on-chip, such as the Orion SoC or the Xavier SoC, which are Nvidias systems-on-chip designed for AI and autonomous machines[11].

In 2016 Nvidia created the PilotNet single deep learning network(DNN), which takes pixels as an input to create the desired trajectory for a self-driving vehicle. This deep learning model ran on the Nvidia DRIVE AGX platform and could average out 500 kilometres of autonomous steering before requiring a person to take over[3]. As opposed to older

applications of artificial intelligence which used handcrafted rules, the PilotNet model uses neural networks, which uses an extensive collection of pre-defined examples to solve the issue with autonomy[3].

The realization that GPUs can help accelerate the time of training Neural Networks, and processing image pixels from a camera or a sensor, has played a significant role in encouraging the use of neural networks for autonomous vehicles[3]. Nvidia has developed various systems on chips specialized in AI computing, including the Tegra Xavier, which we will look deeper into in this thesis.

However, autonomous driving is not the only field of AI that takes advantage of the system-on-chip architecture. Another example is utilizing a system on chip for detecting melanoma cells proposed by Afifi et al[31]. Melanoma is a very aggressive type of deadly skin cancer, and early treatment can substantially increase the survival risk. By integrating a system-on-chip to a low-cost handheld device, doctors have a better chance of catching the disease without waiting for expensive tests that spend a long time coming up with results[31].

Many real-life applications exist for using a systems-on-chip architecture, whether in mobile phones, self-driving cars or medical equipment. For this reason, we see motivation in understanding these architectures better depending on the problems they can help us solve. In autonomous vehicles, we are worried about safety and battery lifetime. So inference speed in detecting objects on the road might be vital. However, in other cases, such as in mobile phones or a robotic lawnmower, there might be a benefit in trading speed for lower power to provide a better user experience or sustainability.

1.2 Problem Statement

Inspired by the background and motivation in the former section, we have decided to focus on the architecture of the Nvidia Tegra Xavier for running inference on some AI workloads. We see this as an important case to study as the popularity of SoCs in different systems is increasing, be it automobiles, robotics or even smartphones. The increasing popularity of deep learning also plays an essential role in defining this project as we see their use increase in medical instruments, electric cars, drones and other systems. Because of this, we see the importance of understanding the architecture to deploy better and safer AI models.

We have decided to investigate how the different components on the Tegra Xavier affect the speed and power efficiency when running inference. We will also dive deeper into Nvidia's frameworks, such as CUDA and TensorRT.

While running inference, we are often interested in processing speed, but the power consumption during real-time inference might be just as important. For electrical systems that run on batteries, we might be as or even more interested in the power consumption of the system. An example could be models running on a robotic lawnmower or a smartphone. In

these cases, it would be preferable if the system did not drain the battery too quickly. So the power usage while running inference will be an essential part of this thesis.

This thesis can be split into three main sections.

- Understanding the different processing units on the Tegra Xavier and their usage. This section will mainly focus on how the Tegra Xavier is built and how to utilize the different processing units optimally. Such as the central processing unit(CPU), the graphics processing unit (GPU) and the neural processing unit(NPU. Also known as a deep learning accelerator). All of these have different strengths and weaknesses, and our task is to understand how to utilize them best.
- Utilizing Nvidia frameworks such as TensorRT and CUDA to optimize inference of deep learning models. Nvidia TensorRT and CUDA are frameworks developed by Nvidia and used for deep learning optimization and graphics card programming. Since the task is focused on the Nvidia Tegra Xavier architecture, using the different frameworks developed by Nvidia will be crucial in fully utilizing the capabilities of the SoC.
- Power managing on the Tegra Xavier while running deep learning models. In this section, we will mainly look into the power consumption of the Tegra Xavier while running inference.

Thus the question of this thesis can be defined as follows:

Can we optimize deep learning models on the Nvidia Tegra Xavier with emphasis on the trade-off between speed and power, using TensorRT and Jetson Xavier power management

1.3 Limitations

There exists a number of SoCs out there, such as Teslas FSD-Chip(Full self-driving chip), Exynos developed by Samsung or the Apple Silicon series, which is the basis of most new Mac computers as well as iPhones and Ipads[5]. However, this thesis will be limited to the Nvidia Tegra Xavier architecture. This has to do with accessibility to a Jetson AGX Xavier developer kit as opposed to other heterogeneous systems-on-chip. Nvidia is also considered one of the world leaders in manufacturing graphics cards, and they redefined modern computer graphics and revolutionized parallel computing[2]. In recent years when the popularity of deep learning exploded, Nvidia GPUs were there to take full advantage of AI computing. Nvidia GPUs and SoC are widely used in automotive systems, robotics, gaming, health care and more[2].

Nvidia also provides their users to take advantage of Nvidia's own ecosystem, which includes frameworks such as CUDA and TensorRT. These frameworks are widely used in AI development and are exclusive

for Nvidia graphics cards and SoCs[2]. So my thesis will be limited to using mainly these frameworks for benchmarking the Tegra Xavier. We will go deeper into these in the following chapters.

We will limit the task to the Resnet50 model when running inference on the Tegra Xavier. Resnet50 is a neural network developed by Kaiming He et al. and is one of the most cited neural networks ever created and won the ImageNet 2015 competition [16]. Resnet is also the base for many other neural networks and is widely used in transfer learning[16]. We will go deeper into the details of Resnet and why we chose it in chapter 5. We will also limit the project to the Resnet50 model trained on the ImageNet data set. This makes it easier to focus mainly on TensorRT and CUDA and see how these can be used to optimize already created deep learning networks. So the task will be limited to optimizing already trained deep learning models utilizing Nvidia frameworks, not developing neural networks or training them.

There are more components on the Nvidia Tegra Xavier than the GPU, CPU and DLA. One of these is the programmable vision accelerator(PVA). This unit is commonly used when dealing with processing computer vision. We will be looking into some computer vision in the by running inference on images with the Resnet50 model. However, since since we are focusing mainly on the trade-off between inference time and power consumption, studies done on the PVA will not be a part of this thesis.

1.4 Main Contribution

Throughout this thesis, we have researched how the Nvidia Tegra Xavier heterogeneous system-on-chip architecture can be used to optimize deep learning models with TensorRT. As such, we have contributed to several details regarding these issues.

First and foremost, we have seen how TensorRT reduces the inference speed on the Resnet50 neural network. Also, we have shown that using a different precision value while running inference can reduce the power consumed by the GPU and the SOC as a whole, all while keeping the speed of inference low.

We have compared the CPU to the GPU and explored why it is preferable to run deep learning inference on a graphics processing unit instead of a central processing unit. With this knowledge, we have shown how we can improve inference further using TensorRT to optimize already existing models.

The architecture of the Nvidia Tegra Xavier also includes the deep learning accelerator, a neural processing unit(NPU). We have proved how this unit can help reduce the power load and memory usage on the GPU and the SOC. By offloading layers from the optimized Resnet50 TensorRT engine from the GPU over to the DLA with FP16 precision, we have reduced the power consumption by up to nine times.

Also, by looking at the power modes of the Tegra Xavier, we have seen how we can limit the amount of power the SOC is allowed to use.

This shows we can configure the SOC to perform within a specific limit, allowing us to limit the power usage to different sorts of problems. One might want to limit the power in cases where we have to decide between speed or power usage.

We have proven that the Tegra Xavier is a versatile and robust system-on-chip. Specialized in AI workloads, it has shown to be able to fit many different scenarios. With the help of TensorRT, one can get the most out of the architecture. This has proven that the compatibility between Nvidia Tegra Xavier hardware and Nvidia's deep learning framework can give us high inference speeds along with low power usage.

1.5 Research Method

The research methodology used for this thesis is built mainly around a design methodology. This methodology is based on the design paradigm proposed in 1989 by Denning et al., from ACM [9].

For this project, we will look into subjects such as machine learning, parallel programming on the GPU, image processing and computer vision. These subjects are somewhat based on mathematics, electrical engineering and software architecture.

The experimental part of this thesis will be rooted in developing a hypothesis and constructing a model around this hypothesis. Then, designing an experiment based on the model before collecting data and analyzing the results.

We will look into how the architecture of the Tegra Xavier performs machine learning algorithms and how the different components of the architecture can be used for optimization, mainly focusing on speed and power. By implementing a program to run deep learning models on the Tegra Xavier and later analyzing the results, we believe we can better understand the architecture.

1.6 Outline

This thesis is structured into seven different chapters. The first two chapters are made as an introduction and explain the background of the research conducted. Chapter three through six explains the background of the work done and the theory behind it. The last chapter focuses on the research results and a final conclusion.

Chapter 2 introduces the reader to systems on chips and mainly the Tegra Xavier architecture. It gives some introduction to parallel computing on the GPU as well as the frameworks used in this project.

Chapter 3 Looks into running inference on the Tegra Xavier using TensorRT, as well as giving some introduction to TensorRT using the MNIST dataset. It also gives the reader an introduction to the workflow of TensorRT and the design pattern of how we will be implementing the code.

Chapter 4 In this chapter, we will look into the NVDLA itself and see what motivations Nvidia had for implementing it. As well as make some assumptions on how it will affect inference regarding speed and power.

Chapter 5 focuses on the running Resnet50 on the Tegra Xavier and how we have proceeded with using TensorRT for optimization. This chapter gives a detailed description of the process of creating software that runs TensorRT on the architecture. We also explain other frameworks used in the program and our justification for choosing them. The chapter also gives a thorough background of the Resnet50 model and explains why we think it is a relevant model for this thesis.

Chapter 6 shows the reader some technical information about the power monitoring on the Jetson Xavier AGX developers kit. Also, some related tools and techniques for managing the power on the SoC.

Chapter 7 Brings it all together by analyzing the research results. Here we draw up our final conclusion and give suggestions for further improvements.

Chapter 2

NVIDIA Tegra Xavier

In this chapter, we will introduce the Nvidia Tegra Xavier architecture. We will look into the Tegra Xavier components commonly used when running inference on deep learning models. Later we will discuss some of the tools used for this thesis. We will also discuss the benefits of using the GPU for doing specific calculations instead of using the CPU. These tasks include problems related to image processing and inference on deep learning models. Which are problems that are parallel in nature, and we will see why it might be beneficial to solve these types of calculations on the GPU.

2.1 Heterogeneous systems on chip

A heterogeneous system architecture is a system architecture that integrates multiple processing units, such as a central processing unit and a graphics processing unit, onto the same system bus, with shared memory. This form of architecture aims to reduce the latency between the different units on the chip. The system architecture makes the devices more compatible by giving a framework to move data between the different processing units.

2.2 NVIDIA Tegra Xavier

The Nvidia Tegra Xavier is a heterogeneous system-on-chip designed by Nvidia. It was introduced in 2018 and is marketed for AI computing. It comprises eight 64-bit ARMv8 cores and a Volta GPU with 512 CUDA cores; parallel processors are used to process the data fed in and out of the GPU, typically used for any graphics computing, such as video game rendering or image processing. The Volta GPU is specified for AI and improving [14].

A detailed list of specifications can be seen on table2.1.

We will now go deeper into the different processing units on the Tegra Xavier. We have chosen to mainly focus on the processing units that are most relevant for deep learning inference. These units inlcude the Volta

Table 2.1: NVIDIA Tegra Xavier specifications

GPU	512-core Volta GPU with Tensor Cores
CPU	8-core ARM v8.2 64-bit CPU, 8MB L2 + 4MB L3
Memory	32GB 256-Bit LPDDR4x | 137GB/s
Storage	32GB eMMC 5.1
PCIe	x8 PCIe Gen4/x8 SLVS-EC
NPU(DLA)	2 Nvidia Deep Learning Accelerators

Graphics processor, the octa-core ARM Carmel CPU cluster and the Nvidia Deep learning accelerator. For an illustration of the Xavier architecture, see figure 2.1 borrowed from WikiChip[8].

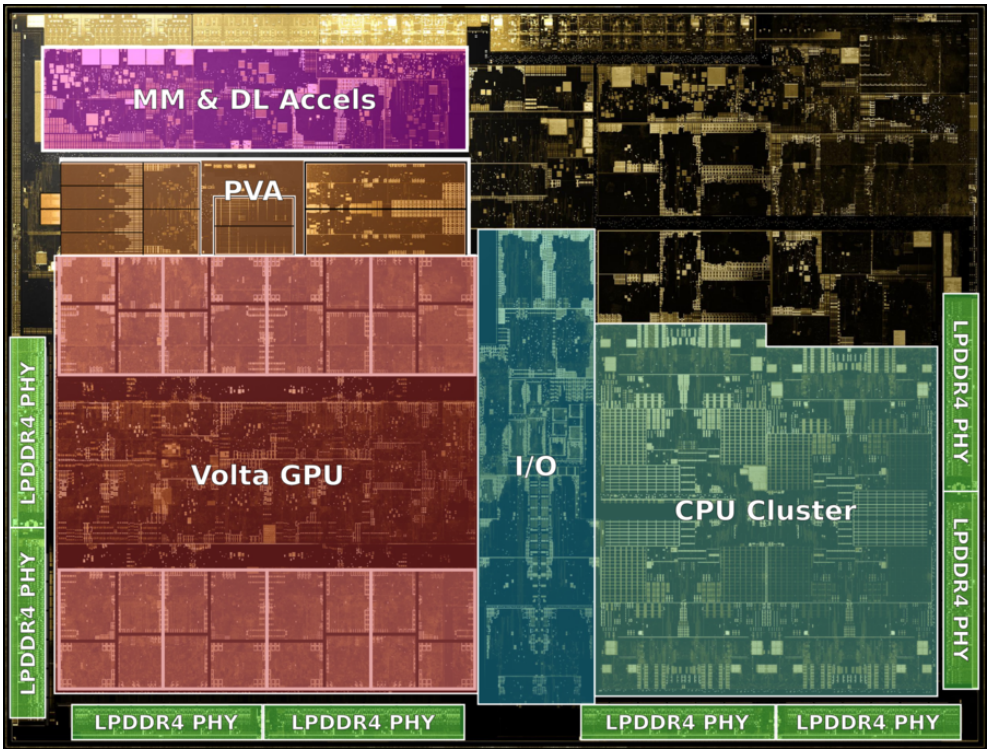

Figure 2.1: A diagram of the Tegra Xavier[8]

2.2.1 The Volta Graphics processor

The Tegra Xavier is a system-on-chip explicitly designed for AI computing, and we can not talk about AI computing without digging deeper into the GPU. With the Volta architecture, Nvidia has developed a new Streaming Multiprocessor(SM) optimized for deep learning [14]. According to Nvidia, the new Volta SM was created to enable a significant performance boost. It has been built with programmable Tensor cores, whose purpose is to run deep learning tensor operations for INT8, FP16 and FP32 precision[24]. We will get deeper into what these precision points mean in later chapters. We can now describe them as instructions to accelerate integer and

mixed-precision matrix-multiply-and-accumulate operations (IMMA and HMMA)[36]. As we will see, these operations are essential when running inference on the Xavier. As of now, we can content ourselves with saying that these operations perform D= AxB + C where A, B, C and D are 4x4 matrices of floating-point 16 or 32. The Volta streaming multiprocessor also provides an enhanced L1 data cache for higher performance and lower latency, promising higher clocks and power efficiency[36]. The Tensor Cores on the Volta GPU have been integrated with cuBLAS, cuDNN, and TensorRT and are programmable through CUDA[15].

The Volta graphics processor has a dedicated hardware block where most graphics functions are performed; this is called the Graphics Processing cluster. This cluster comprises four Texture processing clusters(TPC), which contain two Volta SMs. These SMs units create, manage, and execute instructions for parallel threads[36].

The Volta Graphics processor is specialized for running deep learning networks. See figure 2.2, taken from the Nvidia Jetson AGX Xavier webinar in 2019[15], for an illustration of the Volta architecture.

To see how the GPU compares to the CPU, we are now going to create a program to perform matrix multiplication over matrices of various sizes. Hopefully, this will introduce GPU programming, which will play a significant role when we later want to run inference on neural networks.

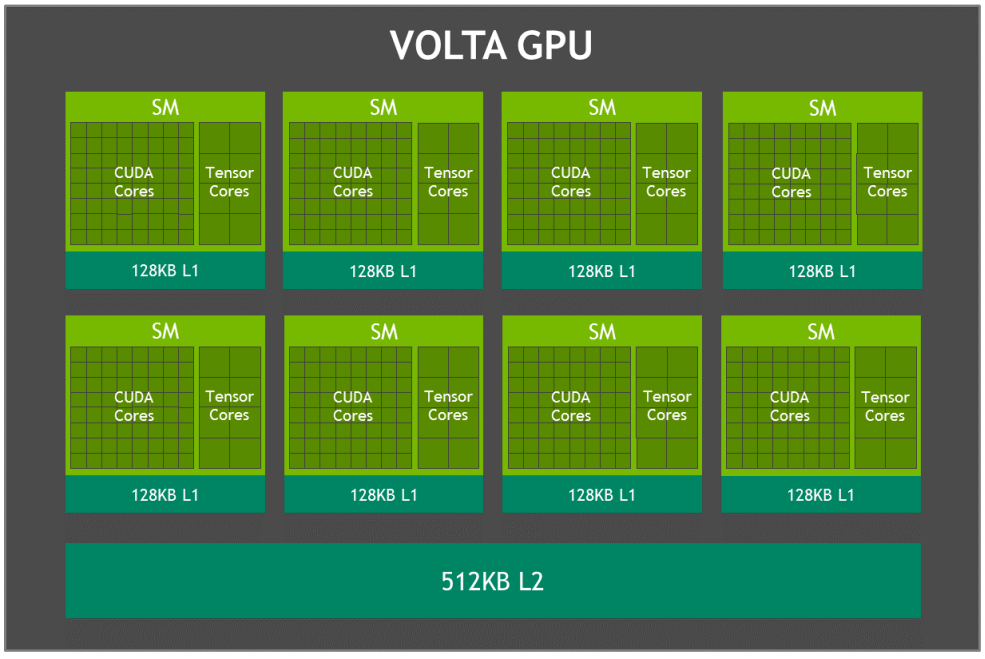

Figure 2.2: The Volta GPU with 8 Streaming Multiprocessors[15]

2.2.2 Carmel processing unit

The Nvidia Tegra Xavier chip features eight Carmel cores, which is Nvidia's 64-bit ARM core. These cores are implemented with ARMv8.2

and include a dual execution mode.

The cluster consists of 4 duplexes sharing a 2 MB of L2 cache. The L2 cache is the cache memory of the CPU and is located outside and separated from the chip core. At the edge, the CPU complex has a 4MB L3 cache. This cache is a bit slower but holds a higher memory level and is shared among the whole complex[24]. All the cores are cache coherent to avoid data inconsistency. The Carmel CPU complex provides a high-performance System Coherency Fabric(SCF), which connects all CPU clusters and enables simultaneous operations of all CPU cores. This creates a true heterogeneous multi-processing(HMP) environment[24]. This coherence is extended to the chip's GPU and other accelerators. For an illustration of the Xaviers CPU cluster, have a look at figure 2.3 borrowed from the Nvidia webinar on the Jetson Xavier[15].

The CPU also supports power management with multiple power domains. We will get a little deeper into this in chapter six. However, briefly explained, it allows the user to change the power mode depending on the situation. As an example, running on different power modes changes the number of active CPUs and their maximum frequency. By default, the system has four cores running with a max frequency of 1200Hz[24]. All the cores consume about 500 - 1500 mW each[15].

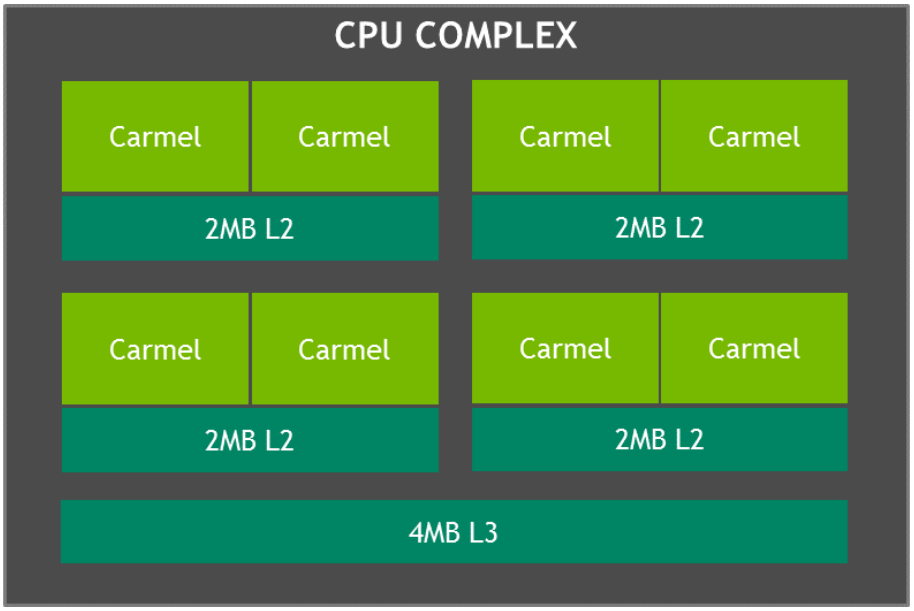

Figure 2.3: Carmel CPU complex[15]

2.2.3 Nvidia Deep Learning Accelerator

The Nvidia deep learning accelerator(NVDLA) is a type of neural processing unit known as a deep learning accelerator. The DLA is an electronic circuit designed for deep learning algorithms. Each Xavier platform has two of these DLA engines integrated and has a peak performance of 5 Tera operations per second (TOPS) for int8 precision and 2.5 Tera floating points

operations per second (TFLOPS) for fp16 precision. These cores optimize power efficiency and consume about 500 - 1500 mW[15]. The engines are programable with TensorRT and are one of this thesis's main points of interest. We will go deeper into their architecture in chapter 4.

2.2.4 Comparing the different processing units

In this thesis, we will look into the different processing units that have been optimized for deep learning. We hope to see differences in power usage and speed of inference when running deep learning models. However, to give a taste test of the differences between the GPU and CPU, we will create a simple program that runs a matrix multiplication algorithm on the CPU using the C programming language. Then we will compare this to a matrix multiplication algorithm on the GPU using CUDA, a C++ framework developed by Nvidia for programming on the GPU[21].

2.3 Matrix multiplication on the Tegra Xavier

To get to know the Tegra Xavier, we implemented a simple matrix multiplication algorithm in CUDA. This was to test how a typical parallel algorithm can be optimized using the GPU. The benefit of using the GPU to process such algorithms is that the GPU is specialized in computing highly parallel computations, which is what graphics rendering is all about. Programs that process large data sets can be optimized using a parallel programming model[21]. See the code below.

```
void matrixmul(int *a, int *b, int *c, int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            for(int k = 0; k < size; k++)
            {
                c[i * size + j] += a[i * size + k] * b[k * size + j];
            }
        }
    }
}
```

This code uses C to run matrix multiplication on the Carmel CPU.

```
__global__ void gpu_matrixmul(int *a, int *b, int *c, int size)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y ;
    int column = blockIdx.x * blockDim.x + threadIdx.x ;

    int sum = 0;
    if((row < size) && (column < size))
    {
        for(int i = 0; i < size; i++)
        {
            sum += a[row * size + i] * b[i * size + column] ;
        }
    }
    c[row * size + column] = sum ;
}
```

This code uses CUDA to run matrix multiplication on the Volta GPU.

2.3.1 Results

Performing a matrix multiplication of the same matrices will have a big-O notation of O(N^3), which can be significantly improved by transferring it to the GPU and running the computation in parallel.

As one can see looking at the code examples on the following page, the original algorithm is a triple for loop doing calculations on the matrices. As mentioned, this algorithm will run in O(N^3) time, and with the increasing size of the matrices, the time to run the algorithm will increase significantly, as seen in table 2.2. We are running the algorithm as a nested for loop on the CPU; one can see how inefficient it is. On a 512x512 sized matrix, the algorithm uses, on average, just above a second and on a 1024x1024, we get over 10 seconds of calculations. From it ramps up, on a 2048x2048 matrix, the algorithm spends almost up to a minute and a half. When we increase the size even further up to 4096x4096, it takes about 14 minutes to run the total calculations. Comparing that amount of time to the GPU run time, we can see where a parallel algorithm's benefits come in. On all sizes calculated, the run time kept pretty consistent at 0.100ms. This makes sense considering the size of the problem and the technique used.

The implementation has been significantly simplified to run in parallel. Decreasing the algorithm from O(N^3) to O(N), shows what benefits one can achieve by optimizing an algorithm by transferring the computations over to the GPU.

Table 2.2: Matrix multiplication

Size	512x512	1024x1024	2048x2048	4096x4096
CPU	1.21s	10.55s	86.98s	887.42s
GPU	0.100ms	0.109ms	0.124ms	0.130ms

2.4 AI computing

There has been an explosion in the progress of AI computing over the last decade. AI computing is the simulation of human intelligence processing done by computers. Typical applications of AI computing include natural language processing, speech recognition and computer vision. The huge popularity growth of the AI world has been Nvidias motivation for developing their heterogeneous systems-on-chip, such as the Nvidia Tegra Xavier[15]. As mentioned introductory to this thesis, Xavier architecture has already seen its use in the automotive industry.

The parallelism capabilities of the GPU have proven able to train large neural networks[18]. AI computing and machine learning are used in many aspects of modern society, from web searches, content filtering and recommendations on e-commerce websites[37].

2.4.1 Deep Learning

Deep learning is a part of the broader field of machine learning and AI computing. Deep learning allows computational models to learn data representations with multiple levels of abstraction[37].

Deep learning uses back propagation algorithms to indicate how to perceive specific structures in large data sets. The limitations of conventional machine learning were that they were very limited in processing data in its raw form. For a long time, constructing pattern recognition was an extremely daunting and engineer-heavy task. However, with representation learning, it became possible to feed with raw data, to discover representations for detection or classification automatically[37]. Deep learning methods are representation learning methods with multiple levels of representation[37], which means that the algorithms use multiple levels of abstraction. By going from simple raw data to transforming it for each layer into a higher, more abstract level[37]. By doing these sorts of transformations layer by layer, very complex patterns can be recognized, and thus very complex functionality can be learned. The key feature of deep learning that separates it from other types of AI computing and machine learning paradigms is that most of the learning is done automatically instead of being heavily designed by engineers. We mean they are learned from data using general-purpose learning procedures or algorithms[37].

As explained, Nvidia has developed GPUs for a long time, and with the increased popularity of deep learning and the benefits of GPU parallelism, Nvidia has become one of the leading companies in AI hardware. Deep learning is also the primary tool used for object detection in NVIDIA's

DRIVE platforms[32].

In this thesis, we will limit ourselves to running inference on models already created and trained on big data sets. We will look a little bit into Deep Learning frameworks. Many different frameworks exist for deep learning, such as Pytorch and TensorFlow. We will satisfy ourselves by looking into TensorFlow to give some background to how deep learning models work, as it will give us a baseline for inference speed. However, the main interest will be in optimizing these models using TensorRT.

2.4.2 TensorFlow

TensorFlow is a framework for expressing deep learning algorithms and can be used for many tasks. In TensorFlow machine learning algorithms are represented as computational graphs[10]. This means we have a data structure where each vertex represents an operation. Such an operation can have zero or more inputs as well as zero or more outputs. These operations can, for example, represent mathematical operations, variables, control flows, I/O operations, network communications ports and more. The edges represent the data flowing between the vertices; in TensorFlow, these are known as Tensors. In TensorFlow, *tensors* are defined programmatically as multidimensional arrays[10]. They are immutable objects; their contents can never be updated; one can only create new ones[33]. To put it into a simple example, we can look at an elementary mathematical function like the Pythagorean theorem $a^2 + b^2 = c^2$ we can picture a simple graph of tensors and operations. The a and b variables would be two tensors of only one dimension; a scalar shape. They would be represented on the graph as the edges connected to a vertex representing the square operation. The tensors representing the output of this operation would be a^2 and b^2. These tensors would be the edges connecting to a new vertex representing the addition operation, and the output operation of this tensor would be c^2. This graph is shown in figure 2.4. Of course, this is a very simplified example of how one would use TensorFlow, but it works for illustrating purposes.

Upon executing the graph, TensorFlow will start at an output node that has been requested and work its way backwards by examining the graphs dependencies [10]. The nodes in the graph can then be assigned to one or many processing units (such as CPU or GPU). Within the execution model for TensorFlow, there are two degrees of scalability[10]. First and foremost, the number of machines doing the computation, as the operations might be distributed over many machines. Secondly, the number of devices there are on one machine(in the case of TensorFlow, devices refer to physical execution units such as CPUs and GPUs).

2.4.3 TensorFlow on heterogenous multi core architectures

As noted in the previous section, a device in the context of TensorFlow refers to the physical unit performing the computation. The simplest way to look at an exectution of a session is on a single device such as

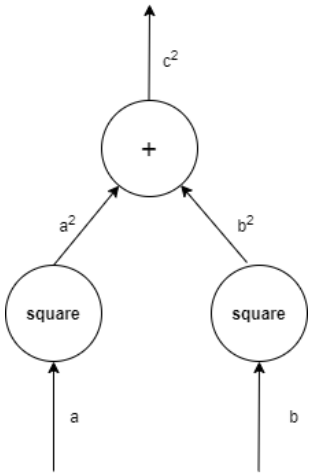

Figure 2.4: Example graph of pythagorean theorem

the CPU. This means that all the vertices of the graph is executed and managed by the device. This means that the master process which keeps track of the nodes in the graph is also responsible for the execution[4]. A more efficient solution is to distribute the execution out to more devices. This means that one can utilize multiple processing units to perform the calculations. *Tensors* are defined as multidimensional arrays, and operations on them can be heavily optimized by running these in parallel. These problems can typically be optimized by running the calculations on the GPU. Considering this, these types of problems should be able to utilize the architecture of the NVIDIA Tegra Xavier to the fullest. A typical TensorFlow process implementation can be seen in figure 2.5 borrowed from the paper by by Abadi et al[4]. Here we see one master process which controls the worker that runs the computation on multiple processing units.

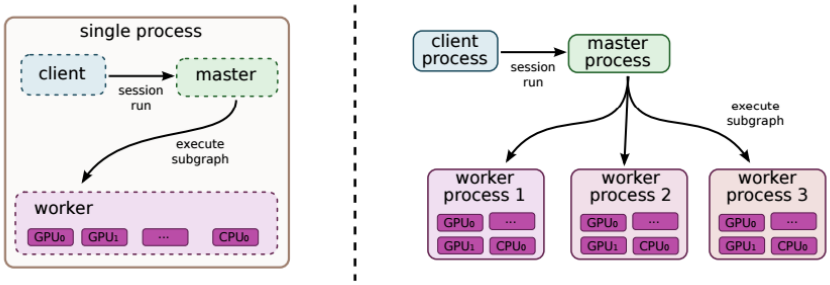

Figure 2.5: Single machine and distributed system[4]

2.4.4 TensorRT

TensorRT is an SDK developed by NVIDIA for high-performance deep learning inference. It is used to optimize neural network models in all major frameworks, such as TensorFlow or Pytorch. It was developed in CUDA and is made to significantly reduce application latency, a requirement for many real-time services[25].

With this in mind, TensorRT should be a perfect candidate for further optimization. Especially considering its tight integration with TensorFlow, this allows one to utilize the full potential of a model created in a deep learning framework like TensorFlow on the Tegra Xavier. In later chapters, we will dig further into what TensorRT is and how it is used in optimizing deep learning models, as it is one of the main problem statements of this thesis.

2.5 The road ahead

First and foremost, it will deal with getting a more thorough understanding and hands-on experience with the frameworks, mainly TensorRT. Utilizing the tools will be a prerequisite for the main task of this thesis: benchmark the Xavier by running inference on Resnet50 and optimize it for speed and power. After that we will dig deeper into the Resnet50 model itself to optimize it using TensorRT. We will then look into the power sensors on the Tegra Xavier and the different power modes to see what trade-offs we can get between the speed of inference and the power usage on the SoC.

Chapter 3

TensorRT for inference on the NVIDIA Tegra Xavier

This chapter will introduce us to TensorRT by running a workload on the MNIST dataset. We will look at the typical workflow when designing code with TensorRT and make some thoughts on how to proceed when we later try to analyze how to utilize TensorRT with the Tegra Xavier architecture best.

3.1 TensorRT

As already mentioned, TensorRT is an inference framework developed by Nvidia. It is considered the fastest inference framework for Nvidia hardware and is designed to be compatible with a list of famous machine learning frameworks such as TensorFlow, Pytorch and MXNet[26]. The framework does not focus on training neural networks but improves inference on newly created networks on Nvidia hardware.

3.1.1 ONNX

ONNX stands for Open Neural Network Exchange and is a way to represent trained machine learning networks. Neural networks created by frameworks such as TensorFlow or PyTorch can be represented in the ONNX model. TensorRT comes with a library to parse the models represented in the ONNX format and allows us to start running inference on the models in TensorRT quickly. Parsing an ONNX file in TensorRT is done in just a few lines of code and allows the programmer to quickly start optimizing without knowing much about the frameworks used to create the model[27].

3.1.2 Workflow

The TensorRT workflow consists of a build phase and a runtime phase. In the build phase, one creates the network to define how to run the model. This can be done in multiple ways. One can construct and define the

network step by step using the TensorRT's Layer and Tensor interfaces. Alternatively, one can use the already built-in ONNX parser interface, IParser, to parse already created models. This is preferred since we will be working on running inference on the Tegra Xavier to test the architecture and not analyze the models themselves. To use the ONNX parser, one needs to include the NvOnnxParser header file and a couple of lines of code.

After creating a network for the parser, one uses the IBuilderConfig interface to specify how TensorRT will optimize. With these two interfaces, it is possible to create the builder, which is called to create the engine. The builder creates a serialized engine, a plan, which can be stored on a disk or deserialized to run immediately. When creating an engine in TensorRT, the engine is created specifically to run on the GPU where it was created, and on the version of TensorRT used. This means it is impossible to create the engine on a Geforce RTX 3060 card and transfer it to the Tegra Xavier, or vice versa.

The runtime phase is the execution phase of TensorRT. This phase starts with deserializing the plan to create an optimized engine. After that, an execution context will be created, where we invoke inference by passing data. This data can, for example, be a picture, such as in the MNIST network. Here we have a .pgm file which represents a handwritten number. The task of the model is to be able to guess which number is represented on the image. We can use the execution context to run inference by passing the buffer to either the execute or the enqueue functions. The execute function runs inference synchronously while the enqueue functions does so asynchronously [26].

The build phase is done sequentially as the builder times algorithms to determine which will be fastest. Since this is the case running the builder in parallel would lead to poor optimization[26]. It has to be done on one GPU and can be stored to disk. After that, we can parse the engine for running inference. Running inference on the execution can be done asynchronously; however, we see no benefits in doing so on such a small data set.

3.1.3 DLA

Deep Learning Accelerators or DLAs are processors dedicated to running inference. These processors are to be seen on many Nvidia SoCs and are supported by TensorRT. It is possible to run parts of the network on the DLAs and the rest on the GPU. The Tegra Xavier SOC is fitted with two of these DLA cores, as seen in figure 2.1. These processors are made for deep learning operations, and it should be interesting to see how they might help optimize for inference. Nvidia has named their DLA the NVIDIA Deep Learning Accelerator(NVDLA). This open-source hardware neural network AI accelerator exists on the Xavier SOC. As the name explains, the NVDLA is just an accelerator, and the process to use it must be scheduled by a CPU or GPU. The TensorRT developer guide NVIDIA explains that "DLA is designed to do full hardware acceleration on convolutional neural networks"[26]. This means that the DLAs are well

suited for deep learning models dealing with image processing. We will look more into the architecture of the NVDLA in chapter four. DLA can only be run on FP16 or INT8 precision.

3.2 MNIST

As a starter project, we have been looking into the MNIST dataset. MNIST stands for Modified National Institute of Standards and Technology, and it is an extensive database of pictures of handwritten numbers[34]. There exist already pre-trained models on recognizing which number is written on the image. This is a good starting point for getting to understand the workflow of TensorRT and getting familiarized with the libraries.

As mentioned, the phases of TensorRT consist of first building the network from a pretrained model, which would be the MNIST.onnx model. This model comes with installing TensorRT, and the builders create a serialized engine from the ONNX model. During the build phase one can choose the level of precision for the network; FP32, FP16, INT8. In TensorRT, the default is set to FP32. The Tegra Xavier architecture does not support INT8 precision. So we will content ourselves with running the tests on FP32 and FP16.

The program itself follows the typical TensorRT workflow described in the previous section. Firstly it checks if the engine already exists on the disk; it builds the engine before loading the network. If the engine exists on the disk, it skips the building part and jumps directly to loading the network for inference. Lastly, it runs inference.

3.2.1 Inference options

Since there are so many options for running inference using TensorRT, we saw a need to be able to pass options. Currently, the options list only contains a few choices, and these are made in a struct called Configurations. The options available for the moment are the decision to choose whether one is using 16 floating-point precision, maxWorkspaceSize and DLA core.

```
struct Configurations {
    //Using 16 point floats for inference
    bool FP16 = false;
    //Max GPU memory allowed for the model.
    int maxWorkspaceSize = 40000000;
    // DLA
    int dlaCore = 0;
};
```

3.2.2 Engine on disk

As mentioned, there is a possibility to write the engine to disk. We decided to serialize the name using the engine configurations to check if an engine

with the same configurations already exists. This means an engine can be stored to and loaded from disk using the name based on the options. All engines start with the string "trtengine", and after that, we append the options used to the name. An example name can be something like *trt.engine.fp16.dla*; this means that one uses floating-point 16 and a DLA core to run inference. We found this naming convention very useful and will continue with it for the rest of this thesis.

During the build phase, the program checks if an engine already exists. If the engine exists, it simply jumps to the loading phase. By saving the engine on disk, we do not have to build from scratch, which saves us some time since the build phase can be a little time-consuming. However, we met a problem regarding running inference on a network loaded from a disk. For some reason, the inference became slower than when one built the engine to run inference on it.

At this stage in our testing, we did not yet understand why it executed slower inference while loading pre-built engines from disk. It was not until we started looking at the Resnet50 model in chapter 5 that it became clear. What happens is that CUDA, along with its libraries CuDNN and CuBLAS, takes some time to initialize. During the build phase, TensorRT uses CUDA to test for the engine's best possible optimizations. However, when we load the engine from storage, CUDA has not yet initialized, making this happen first in the inference phase. For this reason, it is common to throw away the first inference before running the main execution.

3.2.3 Build phase

Should there not be an engine already stored on the disk, the program will build it based on the configurations passed by the user. The build phase parses the ONNX model before it builds the engine and then stores the serialized engine to disk before it is loaded and runs inference.

3.2.4 Loading the network

Loading the network is straightforward. It loads the network and preps it for inference by creating a CUDA engine. It uses this engine for making the execution context, and the execution context is used for running the inference.

3.2.5 Inference

The inference phase is the most exciting part for this thesis, as our goal is to optimize deep learning inference on the Tegra Xavier.

As mentioned earlier, while we ran inference on the MNIST model, we got much longer inference times when we loaded from the disk than when we built the engine by parsing the ONNX file. When loading the FP32 engine from the disk we got an inference time of 1.35 seconds as opposed to 2.056 milliseconds while building it from scratch. As we said, at this point in time it was not clear to us why this happened. However, with later

FP32	DLA	FP16
2.056ms	x	2.016ms

Table 3.1: Avarage inference time in milliseconds on the MNIST dataset

knowledge, we now know the inference times when building the engine from scratch best represent the inference time of TensorRT. So in table 3.2.5 we decided to show these numbers instead of the inference times that took over a second. One thing that would have solved this issue is running inference multiple times with the same execution context. By throwing away the first one and getting an average on multiple images would have been better. However, at this point, we were not aware of this behaviour.

By looking at table 3.2.5 we see that the inference time of FP32 and FP16 are about the same. This is likely due to the size of the test data. We will look into bigger workloads in chapter 5.

When it came to using the DLA cores, we were disappointed that the MNIST model did not support offloading the GPU. The layers in this model were incompatible with computation on the DLA, which led to a fallback to the GPU, making it simply run with FP16 precision.

We also learned that the Tegra Xavier architecture does not support INT8 precision. When we tested it out we got an error and the program would not compile. Saying INT8 was unsupported

3.2.6 Further testing

Further on, we will try to run inference on an image classification model such as Resnet50. This model is more prominent in size, leading us to believe there will be more gain from testing different precision and benchmarking the power usage of multiple inferences. Also, since the Tegra Xavier has DLA cores designed for deep learning inference, it will be interesting to see their benefits on models that allow for DLA cores to be used. Both consider power management and time efficiency.

Chapter 4

NVDLA

This chapter will look a little into the details of how Nvidia has designed their NVDLA architecture. We will see what the NVDLA was designed for and what problems it aims to solve. We will also make up some thoughts on how we will use the NVDLA when we look into how we can utilize it for the tests we will be doing in the following chapter.

4.1 NVIDIA Deep Learning Accelerator

NVIDIA Deep Learning Accelerator(NVDLA) is an open-source architecture created by Nvidia, which is made as a way to design deep learning inference accelerators. NVIDIA describes NVDLA as "a free and open architecture that promotes a standard way to design deep learning inference accelerators. With its modular architecture, NVDLA is scalable, highly configurable, and designed to simplify integration and portability. The hardware supports a wide range of IoT devices."[20]

4.1.1 Architecture of DLA

The DLA hardware comprises different components, such as the Convolution Core, an optimized high-performance convolution engine. A convolution engine is a programmable processor specified for data flows that are typically convolutional. Data from image/video processing and computer vision are examples of this. The convolutional engine maps parameters of different sizes onto the hardware[20].

The single data point processor(SDP) is a single point lookup engine for activation functions. This means it has a lookup table for implementing non-linear functions and bias and scaling for linear functions. This makes it possible for the SDP to support the most common activation functions such as ReLU, linear activation and PReLU[20]. The ResNet50 model, which we will look further into later, is implemented with for example, ReLU[16].

In convolutional neural networks, there is a technique to generalize features in images. This technique is called pooling. By extracting pieces of an image, we can teach the network to recognize these features, and by stacking multiple convolutional layers, we can make it possible for the

network to recognize complex structures and objects in an image. There are different types of pooling, such as max pooling or average pooling. By this, we simply mean we run a filter over the image to extract data[20]. Let us say we have a filter of a 2x2 matrix, which is the most common. By running this filter over an image of 216x216 pixels, we can extract the highest number that occurs inside the filter at a given time. Then we stride the filter by two and do the same here. This gives us an output matrix of the highest numbers that occur inside the filter at each iteration. This technique is called max-pooling and is commonly used to recognize a picture value in different places in the image[20]. This technique is common when dealing with convolutional neural networks. The Nvidia DLA contains a processor dedicated to these types of operations, called the Planar Data Processor(PDP) and supports common spatial operations in CNNs[20].

Often when performing inference on a CNN, we need to reshape or reconfigure the tensors and copy configurations. For example, when finding different features of an image, such as the spatial regions, which is the process of filtering regions of the image using simple geometric shapes and boolean combinations of shapes[20]. This makes it possible only to process pixels found with specific shapes, and spatial region filtering does this by specifying regions of the images. The DLA architecture uses a data-reshape engine process to do memory-to-meory transformation for reshaping and changing tensors[20].

Lastly, the NVDLA architecture uses Bridge DMA for data transfer. Images and processed results are stored in the external Dynamic Random Access Memory(DRAM), but the bandwidth and latency are generally insufficient to allow the NVDLA to work optimally. For this reason, NVDLA uses a secondary memory interface to on-chip Static Random Access Memory(SRAM)[20]. SRAM is often faster than DRAM but also more expensive and is often used for the cache and internal registers of a CPU, compared to DRAM, which is more commonly used for computers' main memory[20]. To utilize the SRAM one needs to move the data between the DRAM and the SRAM efficiently. For this purpose, Nvidia uses Bridge DMA to transfer data between the two. There are two independent parts, and one copies data from external DRAM to internal SRAM, and vice versa. It also allows for transfer between external DRAM and external DRAM as well as internal SRAM to internal SRAM. Both directions can't work simultaneously[20]. See figure 4.1

4.1.2 Running layers on DLA using TensorRT

The DLA is designed for full hardware acceleration of convolutional neural networks, and TensorRT has implemented DLA support for running specific layers on the SOCs DLA[25]. Running layers on the DLA should help optimize the workload on the SoC, both on inference time and power usage.

There are some general restrictions when using DLA for inference with TensorRT. First of all, when running on any layers, there is a maximum batch size of 4096. Also, the DLA does not support dynamic dimensions

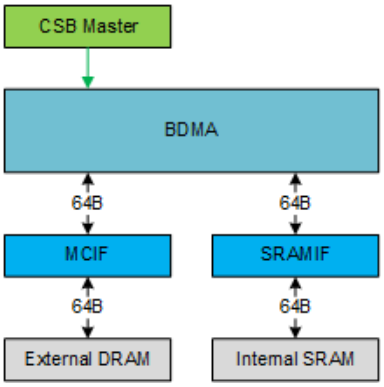

Figure 4.1: Bridge DMA[20]

such as wildcard dimensions for the optimization profiles, which means the min, max and opts values need to be equal. The DLA also only supports FP16 and INT8 precision, and this means that the standard value of FP32 will not work with DLA mode. The Tegra Xavier also does not support INT8 precision, leaving only one option when building our network with TensorRT, which is running the engine with FP16 precision and DLA. Of course, it will be interesting to see how this compares to running the engine with FP32 precision.

Not all layers and layer combination in a network is applicable with DLA. With most networks, this is also the case. This means that during the build phase, the program will check if it is possible to run a layer on the DLA or whether the architecture itself has DLA support. To ensure that it will not result in an error, one can allow the layer to fall back to running on the GPU. This is done by simply setting the builder to allow for GPUFallbackMode and will produce a warning when trying to run on the DLA and throw the layer back to run on the GPU instead. This is great when working on multiple platforms or when one is unsure about the different types of layers on the network.

Chapter 5

Resnet50 on Tegra Xavier

This chapter will take a deep look into the Resnet50 model. We will look into the background of Resnet50 to see how it works. Then we will go into how we use TensorRT to parse an ONNX model of Resnet50 that we can save to the disk to run inference. We will look in detail at how we have designed the code that we will later use to run tests on the Tegra Xavier.

5.1 Background

Resnet, short for Residual networks, is a convolutional neural network(CNN) used for image classifications. This network was introduced to make it easier for the learning algorithm to find a solution in the presence of very deep neural networks. It does this by introducing something called skip connections [18]. When neural networks started to have more and more layers, it stopped being able to learn what it was supposed to, given a higher capacity than 18 layers[16]. This resulted in the network being unable to learn what it was supposed to, and it occurs because of problems such as vanishing/exploding gradients[16].

When deeper networks were able to converge, a degradation problem occurred; with the depth of the networks increasing, accuracy quickly becomes saturated and starts decreasing [16]. Adding more layers to these networks only seemed to increase the training error, although the idea was that more layers should only enrich the models. This problem was solved in Resnet by introducing something called skip connections[16]. This concept makes learning the identity function easier for neural networks. The two main reasons for introducing skip connections are to avoid the vanishing gradients or mitigate the accuracy saturation[16]. Skipping layers in the initial stages simplifies the network and reduces the problem of the vanishing gradients[16]. Then it gradually restores the model, which is called residual learning, because it is learning based on the remainder after greater parts of the network have been skipped before being reintroduced[16]. See figure 5.1 and 5.2. Both borrowed from the article by He et al. [16] .

With Resnet-152, it was possible to introduce 152 layers to the network. However, "only" 50 layers have been shown to be sufficient in practice[18].

For this section, we will look into using the ResNet50 with TensorRT to look for optimizations for speed, power usage and the limits and benefits of NVDLA-cores.

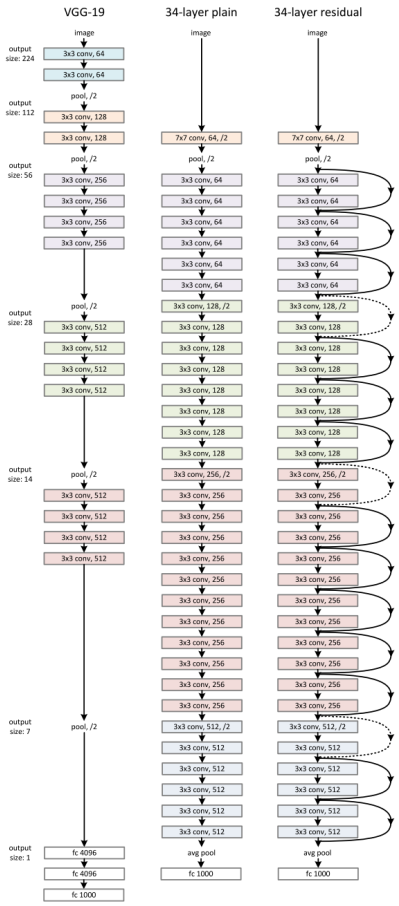

Figure 5.1: Residual learning, on the right with skip connection[16]

5.1.1 Usage of Resnet50

Resnet is, as described, an image classification CNN and is used to extract specific information from an image to be able to classify it. This can, for example, be used to classify animals or objects in an image or whether the image contains certain characteristics interesting to a researcher. An example can be a group of images of areas where people live to see whether it is a rural or suburban area.

The original Resnet50 was proposed by He. K et al[16]. was initially trained on the imagenet data set and was used to classify over a thousand different images. However, Resnet50 has also proven to be very useful in Transfer learning[16], the idea in machine learning that you can use knowledge of solving one problem by applying it to a different issue. In 2019 Ishrat Zahan Mukti and Dipayan Biswas used transfer learning with Resnet50 to classify plant diseases on 38 different classes of plant leaf

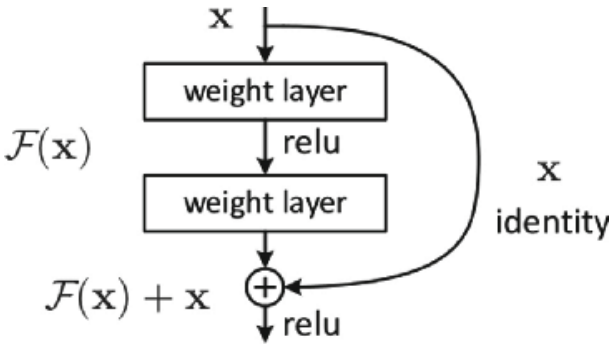

Figure 5.2: Skip Connection[16]

images with the best performance of 99.80 accuracy[38]. The procedure was done using a dataset of different categories of healthy and sick plants and was used to train the model to categorize the leaves. The motivation behind this is that food security is threatened by different factors, such as climate change, pollinator degeneration, plant diseases and more[38]. Dealing with plant disease costs farmers much time and financial resources, and it is hard for the human eye to detect early. Suppose machine learning can be used to diagnose plants as quickly as possible. In that case, it will be much easier to enact countermeasures, which can increase productivity and save the farmers from unnecessary spendings[38].

Another example of using Resnet50 has been seen in biometric recognition systems such as fingerprint analysis. A study from 2018 showed that it was possible to run a simple fingerprint recognition model for classifying fingerprints with Resnet50[29]. It was shown that it was possible to maintain the model's accuracy without adjusting the model except for a change in the input size.

A more recent and highly relevant usage of image classification with Resnet50 was classifying pneumonia cells in x-ray pictures of lungs[6]. In 2021 Çınar et al. used an improved Resnet50 model to classify pneumonia cells in images with a high level of accuracy[6]. Pneumonia is one of the most common causes of death, especially in children under five years old. Where about 15% deaths are caused by pneumonia. So it is evident that early diagnosis is vital to start treatment as quickly as possible. The most common way to diagnose pneumonia is by analyzing images of chest x-rays. These analyses are quite complex and need to be examined by specialists, and this can be problematic in cases where there are not enough experts. So by using digital aid in the form of image classification, there might be a possibility of earlier diagnosing and speeding up treatment[6].

5.2 Motivation for analysing Resnet50 on the NVIDIA Tegra Xavier architecture

There exist real-life examples of usage for the Resent50 model, either by tweaking it to fit the goal or using it as is. So Resnet is an excellent base for other models and thus a good model for transfer learning. The Tegra Xavier is an architecture specially designed for artificial intelligence. Since Resnet allows some of its layers to be run on DLA, it can also be used to benchmark how the different components behave.

5.3 Resnet50 with TensorFlow

Running the Resnet50 neural network is a simple task with TensorFlow and Keras. Keras is an open-source library for Python. It focuses on artificial neural networks and is only supported by TensorFlow. On their webpage, they write, "Keras is an API designed for human beings, not machines. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of action required for common use cases... " [17]. Keras have hundreds of already implemented commonly used neural networks, and Resnet50 is one of them. Thus we can create a simple Python script to easily import and run inference on Resnet50 on some sample images to classify them. We have chosen images of Leopards, Cheetahs and Jaguars. The reason behind this is that they are very similar animals, and it will be interesting to see if they get classified successfully, especially after optimizing with TensorRT. Having an optimized model does little good if the correctness is lost.

The reason for checking how Resnet50 works with TensorFlow is to have a baseline to compare the optimized model in TensorRT. Other frameworks exist, for example, Pytorch or SONNET, as well as other variations of TensorFlow. Our reason for deciding to use TensorFlow, with some help from Keras, is simply because it is a popular framework, and we already have some introductory knowledge of it. As well as being one of the most popular frameworks, it also has easy access to Keras and its models. Resnet50 already exists in Keras, and one can quickly upload the model and run inference on a dataset.

5.3.1 OpenCV

To be able to run inference on some images using Resnet50, there is a need to be able to process these images efficiently. For this, we have chosen to use OpenCV as it is considered one of the most popular open-source frameworks for computer vision. It makes it easy to process images from the disk without doing too much trickery. Also, it comes with a vast library of algorithms for use in computer vision [28]. It has a C++ and Python interface, making it easier to implement with this project.

Figure 5.3: Stock images used to infer big spotted cats

5.3.2 NHWC and NCHW

The first issue we have experienced with OpenCV and TensorRT is that TensorRT stores images on the NCHW form and OpenCV on the NHWC. The way to understand the difference between the two is to see what each letter signifies. The N signifies the number of images in the batch. C stands for the channels of the image, which will be whether it is RGB, greyscale etc. RGB has C equals 3 and greyscale 1, which makes sense considering how the images are layered. W and H stand for width and height of the image [7]. When it comes to NHWC compared to NCHW, it speaks of in what shape the image is stored. RGB has already been mentioned; this means that the picture is made of three channels; Red, Green and Blue. With NCHW for each colour, the pixels are stored together with their respective channels. This means all read pixels are stored together, the green pixels and finally all the blue pixels. NHWC on the on the other hand stores each pixel in RGB values, which means the first pixel is stored in R[0], G[0] and B[0] order. See Figure 3.1. Since TensorRT expects pictures on the NCHW format and OpenCV is in NHWC we will have to do a transformation. See figure 5.4.

5.4 TensorRT on Resnet50

When optimizing the Resnet50 Onnx model with TensorRT, we expect that there will be a decent upgrade in the inference time performance.

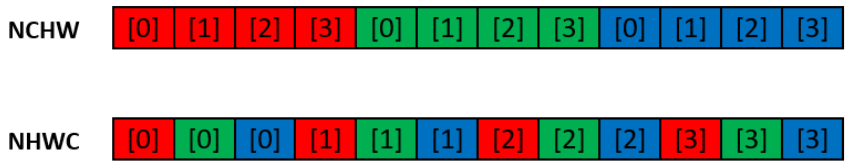

Figure 5.4: Difference between NCHW and NHWC [7]

The reason for believing this is that when one creates a network with a framework like TensorFlow or Pytorch, the network consists of a certain number of layers. TensorRT aims to fuse certain layers into one. This is called Kernel Fusion and is done to improve GPU utilization[26]. With fewer kernel launches, there should be an improvement in memory usage and bandwidth. There are two primary fusions. Vertical fusion; combines sequential kernel calls, and Horizontal fusion; combines the same kernels with a common input and output[26].

Another aspect of TensorRT is the idea of precision calibration. Most deep neural networks are trained with full precision(FP32). While running inference, we do not backpropagate an algorithm to supervise the learning of the artificial network[19]. This step is hard to keep stable and often needs a higher level of precision. Since there is no backpropagation in running inference on the model, we can allow for a lower level of precision. FP32 has a dynamic range of -3.4x10^{38} to 3.4x10^{38}, compared to FP16 and INT8, which have -65504 to 65504 and -128 to 127, respectively; we can create a lot smaller model sizes. This will result in lower memory utilization and latency, hence higher throughput[19]. Especially when running operations on the Nvidia Tensor Cores, which are, as explained by Nvidia, "programmable fused matrix-multiply-and-accumulate units that execute concurrently alongside CUDA cores."[23]. The Tensor Cores are designed to accelerate dense linear algebraic computations, signal processing and DL inference[23]. Nvidia calls these matrix-multiply-and-accumulate units. This simply is the idea that these cores implement instructions for running half-precision matrix multiply and accumulate(HMMA), which is for running with FP16 or FP32 precision, as well as the integer matrix multiply and accumulate(IMMA), for INT8 or INT16[19]. Nvidia libraries such as cuBLAS, CUDNN, and TensorRT have been updated to utilize these operations internally. Considering the 64 Tensor Cores and 512 Cuda cores on the Tegras Volta GPU, we expect significant gains in running inference with the Resnet50 model.

When optimizing deep learning models with TensorRT, there is the idea of kernel auto-tuning. Nvidia has implemented multiple-low level implementation for typical operations[26]. When building a network using TensorRT, it selects the optimal kernel based on different parameters, such as precision, workspace size and the target platform. This means that when creating a TensorRT model on the Tegra Xavier, one will not be able to port it to another architecture. It is also based on the which CUDA and TensorRT

release one uses when building the network[26].

When creating the TensorRT engine one the Tegra Xavier, we will look into the inference time and power usage performance. We decided to use the C++ API for solving this part of the project over Python. This is mainly because of the experience we have with C++ compared to Python as well as the fact that the Python API is not portable to any Windows platform. However, we will be using a Linux-based operating system(Ubuntu), but it is a benefit to have the option of porting the code.

5.4.1 Programming the Resnet50 model

For programming the Resnet50 model on the Tegra Xavier, we will use the same procedure mentioned in chapter 3. We will use the C++ TensorRT API to parse a pre-trained Resnet50 ONNX model and write it to disk so we can reuse it for later inference. To make the engine file from certain configurations, such as precision level, DLA and Workspace size, we will use a configuration struct. The name of the created engine will be saved to disk with the name serialized from the active configurations; trt.*precision.batchsize.DLAcore.workspacesize*. An example could be trt.fp16.d1.400000, which would mean trt engine with floating-point 16 precision, running on DLA core with max workspace size of 400000 bytes. If the name exists, an engine has already been created, and the program will jump to the load phase. The configurations are given by the user when running the program.

The TensorRT engine will be implemented as an engine class, consisting of three public methods; build, loadNetwork and inference. We figured that the TensorRT engine in itself seemed like a natural object. So using a class to map out the problem seems like a natural solution. However, it would not necessarily be needed. Nevertheless, this lets us generalize the problem to other models as well. It also makes it simpler to adjust and modify the program for further use.

The engine class also consists of some private methods needed for building, loading and running inference. As well as some private properties needed by TensorRT, such as the built engine itself and execution context for running inference, with some other properties needed by the network. Such as input height and width, channel number and batch size. OpenCV needs this to be able to parse the input images. We will use the same three images for testing inference time as we used with Resnet50 in TensorFlow, see figure 5.3. These are images that the Resnet50 model has been trained to see along with about 1000 other classes of objects. We believe that using these images would be sufficient to test the system.

Although we have given a brief walkthrough of the build, load and inference phases for the MNIST model, we will now give a much deeper walkthrough for the Resnet50 model. Because the different choices and configurations will likely have a higher impact on this model, we see it essential to explain why we have made our choices thoroughly.

5.5 Build phase

The build phase of TensorRT is a very generalized operation. In this phase, we parse the ONNX file and the configurations given by the user. Such as precision number, DLA core and workspace size. When running inference using TensorRT, the build phase can be pretty long and may consume a bit of GPU memory, as well as having a high power consumption. This is because TensorRT uses information from the configurations, the version of CUDA, and the system architecture to find which layers to fuse and how to best optimize the engine for later inference.

For this reason, it is essential to be able to store an already created engine on the disk. Power consumption is crucial, especially if using an embedded system running on a battery. If the engine has already been created with the given configurations, this phase will return true and jump to the load phase.

Should the engine not already exist, one will have to be created. Firstly we create the builder object using the IBuilder interface. Here one sets the max batch size for the system and does some checks to see if the platform has FP16 and INT8 precision or running on DLA cores. As we already know, the system does not support INT8 and has two DLA cores. Creating the network is done using the builder and is later used by the parser to parse the ONNX model. It parses the model to a C++ vector buffer, and should there be any errors while parsing, they will be written out to the console. Then it creates the IBuilderConfig object, which is used to add the optimization options. This object is made from the builder and is where we add the optimization profile, precision, DLA, and CUDAstream for asynchronous running. Firstly we take the optimization profile. This profile describes the dimensions for each input and the dimensions for the auto-tuner to use for optimization. When defining the optimization profile, one uses the dimensions defined by the model, which are the input dimension(NCHW). These can be obtained from the network by calling the getInput method on the network object. Then one can call the getDimensions method on the input object to get the input dimensions. The channel, height and width will be stored in the input dimension on indexes 1, 2 and 3, respectively, which is in CHW format. On the optimization profile, one then sets the dimensions. Which will be the input name, kMin, kOpt and kMax profiles. Setting a default optimization profile is done in this way for kMin, kOpt and kMax. The belowe code snippet shows how to set it for the minimum dimension. It is similar for optional and max dimension.

```
defaultProfile->setDimensions(inputName,
    OptProfileSelector::kMIN, Dims4(1, channel, height, width));.
```

We were also planning to set different optimization profiles, which would be changing the 1 value between optional values and the max value based on the batch sizes. However, this does not work as the Resnet50 model only allows for static values. Since this, unfortunately, does not

work with Resnet50, it will have to be something to test for future projects. After this, the building method will look to see if the precision flag has been set for FP16 or INT8. As the Tegra Xavier does not allow for INT8 precision, there is not much need to set it. However, to generalize this code to other platforms, we left it in as there was no point in removing it. If the user tries to set it to INT8, the program will issue a warning and just run it on FP32, which is the standard. If no flag is set, it will run on FP32. We then check if the user has set the engine to run on DLA. Here it is important to check whether the flags for FP16 or INT8 have been set as DLA does not allow for fp32. If the user has allowed for FP16 and added the DLA flag, it will set the device type on the IBuilderConfig object to kDLA and set it to run on DLA. It is also essential to allow GPU fallback should the layers in the network not allow running on DLA. Should this be the case, it will simply print out a statement saying that the layer does not work on DLA and fall back to be run on the GPU. Finally, one adds the workspace limit of the engine, which the user adds, which is it for the configurations.

The last part of the building phase is to create a serialized model from the network and configuration. This serialized model can be deserialized immediately, loaded for inference, or stored on the disk. We have decided to always store the models on the disk to be loaded. This is to save the building time for running the program multiple times. Since we store the engine based on which configurations the user has put in, we think it is essential not to build another model each time one wants to run inference—especially considering the time, memory and power consumption of the build phase. This way of doing it also translates better to real-life deployment. After the engine has been written to disk, the build phase is over, and should there not have been any errors along the way; it will return true and go on to the loading phase.

5.6 Loading phase

The loading phase finds the engine on the disk based on the engine name. As described earlier, the engine name is based on the user's configurations to ensure that each created engine has a unique name. If the program finds the serialized engine stored on the disk, it will simply parse it into a buffer for loading the network; loading one first needs to create a runtime object using the IRuntime class. This class allows for deserializing a functionally serialized unsafe engine. By unsafe, we mean an engine stored on the disk but not ready for running inference. The runtime object is created using the ILogger object created by the Engine class. Then one sets the main device for running inference, which will be the GPU. This is done using the cudaSetDevice method. In most cases, this will be device index 0. However, one can add another device if one wants to run on multiple GPUs.

After all this, we can finally create the CUDA engine for running inference. We do this by calling the deserializeCudaEngine method on the network object using the buffer that has stored the serialized model

as parameters. This engine has stored all the data from the building phase. Here we can get some information from the engine needed for running inference. We have stored these as private members in the Engine class. These members are the input name, output name, batch size, input channel, input height and width, also known as input dimensions. Similar to how it was done in the build phase.

We meat a small inconvenience here. When creating the engine to store on disk these members can be initiated during the build phase. However, how we designed the program makes it so that it does not always build the engine directly from the ONNX model. This means that the build phase gets skipped every time an engine is already stored on the disk. Because of this, we have to initiate these members here as well. This means that if we build the engine from the ONNX model, the members will be initiated again, and since they are made using the same model, they will be re-initiated to the same values. It does not affect the program that much but is a little annoying. However, we need these values both for the building and inference phases. So we do not see another option.

After the CUDA engine has been created and the dimensions have been initiated, we can go on to create the execution context. The execution context is responsible for running inference on the given input, and the execution context is created by calling the createExecutionContext method on the CUDA engine. After that, we can create a CUDA stream should we wish to run the program asynchronously.

5.7 Inference

The building and loading phases are problems that are easily generalized to all ONNX models and require minor tweaking to build other networks. However, things get a little more tricky when one wants to run inference, and it is here that we have met the majority of our difficulties.

When running inference, one needs more knowledge of the task at hand. The Resnet50 model is used for classifying images, making knowing some image processing valuable. We want to run inference of a group of images we know have been trained on the Imagenet data set. For this, we chose the three pictures of different spotted cats. See 5.3. We chose these three because we knew the model should be able to classify them, and we wanted some different pictures for running inference. Since the three cats in question are quite similar, we also thought this would tell us something about the model's accuracy.

The first part of the inference procedure is to load the images, modify them to fit the model, and then transfer them to the GPU to run inference. First, we need to get the input dimensions of Resnet50. This is the metadata we stored earlier, height, width, channel, and other data such as the number of bindings. The bindings are the dimensions of the model. Such as input and output bindings. The Resnet50 model has two, one for input and one for output. Other models could have more. Then we create a buffer, which is a void vector, with the size of the number of bindings. This

buffer is where we store the input to inference and the output of the results. Then we get the total size of the bindings, which is the size of dimensions multiplied by the batch size we are running inference on. The binding buffers are transferred to the GPU using cudaMalloc or cudaMallocAsync. Soon we are ready to run inference on the model, but first, we need to do some preprocessing to make the images work on Resnet50. This procedure is called resize and normalize and is quickly done in OpenCV.

5.7.1 OpenCV with CUDA

For preprocessing the images, we chose to do it in OpenCV because of their vast library of common image processing procedures. OpenCV has also released a library to run image-processing on GPU called OpenCV with CUDA. Here one can easily upload the images, which in OpenCV is a class called cv::Mat. An object created by OpenCV to present images.

One can upload the frame to the GPU by creating a GpuMat object using the OpenCV with the CUDA library. This is done in one simple line;

```
cv::cuda::GpuMat gpu_frame = gpu_frame.upload(frame);
```

where a frame is an image loaded from a disk. Then the image will be loaded to the GPU. Then similarily to running OpenCV on the CPU, we can use the cv::cuda namespace to run functions such as resize, subtract and divide, but instead of running them on the CPU, we run them on the GPU.

Our reasoning for choosing this way of preprocessing the image was mainly for simplicity, considering we will not have to create our own CUDA kernels for the processing step. We wanted to do the preprocessing steps on the GPU because image processing is a typical parallel task, and running on the GPU should yield some performance bonuses.

5.7.2 Resize and normalize

The pre-processing procedure starts with resizing the images to a size that fits with Resnet50. The Resnet50 model needs to process images of size 216x216 pixels. However, the images we are running inference on come in different sizes, so we need to resize them to fit the model. This is quickly done with OpenCV with CUDA; call the resize method on a GPU frame and pass the frame we uploaded to the GPU along with the model required dimensions. OpenCV has its size object called cv::Size. The constructor takes to parameters for the height and width and then creates a size object for OpenCV to work with. This is solved by getting the sizes from the TensorRT engine, done by passing the input dimensions on the CHW format. This means that we can find the height and width of the input dimensions on indexes 2 and 3. The code below shows us the resize procedure done with OpenCV.

```
auto input_width = dims.d[3];
auto input_height = dims.d[2];
auto channels = dims.d[1];
auto input_size = cv::Size(input_width, input_height);
// resize
cv::cuda::GpuMat resized;
cv::cuda::resize(gpu_frame, resized, input_size, 0, 0,
                 cv::INTER_NEAREST);
```

After resizing the image, we have to normalize it. Image normalization is changing the range of pixels' intensity value. In the case of Resnet50, it means the mean R, G and B values aggregated over all the pixels of the image. Here we normalize using the mean, standard values proposed by He et al. [16], shown in the code snippet below.

```
//Normalize
cv::cuda::GpuMat flt_image;

resized.convertTo(flt_image, CV_32FC3, 1.f / 255.f);
cv::cuda::subtract(flt_image, cv::Scalar(0.485f, 0.456f, 0.406f),
    flt_image, cv::noArray(), -1);
cv::cuda::divide(flt_image, cv::Scalar(0.229f, 0.224f, 0.225f),
    flt_image, 1, -1);
```

As one can see, this is also done using the cv::cuda namespace, which allows us to offload this to the GPU.

Lastly, we need to copy the data to an output float pointer. Which we do channel by channel. Now the images are ready for inference.

5.7.3 Running inference

After having preprocessed the images, we can run inference. This is done either asynchronously or synchronously. Suppose one wants to do it asynchronously using a CUDA stream created earlier for the engine, which allows for stream-ordered CUDA memory allocators. Now we can allocate and deallocate memory with other work launched into a CUDA stream, which makes it possible to launch kernels and copy memory asynchronously. This should improve the application performance by taking advantage of the stream-ordered semantics to reuse memory allocations [21]. However, this is only available on applications using CUDA version 11.2 or later. After testing this on the TensorRT Resnet50 model using a Geforce 3060ti graphics card. It did not seem to yield any particular benefits for this problem. On the computer using the Geforce 3060ti card that already had CUDA 11.6 installed, we decided it was worth trying here because we only had CUDA 10.2 installed on the Jetson Xavier. So, instead of reinstalling CUDA, which would also have us reinstall OpenCV and TensorRT, we decided to test for it on another graphics processing unit first. Since it did not give the performance optimization we hoped for; we decided to let it be for now. Should there be more time, we might pursue this in later projects.

However, suppose one wants to run inference asynchronously with TensorRT. In that case, one needs to initiate the CUDA stream and allocate the buffers for GPU using the cudaMallocAsync instead of the standard cudaMalloc method. When one wants to copy and free up the memory, one uses cudaMemcpyAsync and cudaFreeAsync. For running inference, one then calls the enqeueV2 method on the execution context. As mentioned earlier, the engine creates the execution context and is the object for running inference. Running inference is done using this method by passing the buffers containing the input images and output one wants to pass the data obtained by running inference as well as the cuda stream. Then passing the output buffer to do the post-processing. The code below shows how to run inference asynchronously.

Running inference synchronously is done almost identically. The difference is simply by using standard cudaMalloc, cudaMemcpy and cudaFree functions. Inference itself is then run using executeV2 instead of enqueueV2. The code for running inference synchronously or asynchronously can be seen in appendix A.

After running inference, one needs to post-process the data to see whether the inference was made correctly.

5.7.4 Processing the output

After running inference we have an array of a 1000 floating point numbers. We need to do the post processing step by copying the data stored on the GPU back to the CPU. This is a simple step done by using cudaMemcpy with the cudaMemcpyDeviceToHost parameter, from the gpu output array to the cpu output array. However before that we need to load the classes that Resnet50 are trained on. These are stored on a text file containing names of all these classes. So by parsing this file into a vector conatining a thousand class names of different image classifications. After parsing these classes we can have each class corresponding to a certain index on the array. Then one can map these indices to a class name to get the classification. For example whether it is a jaguar, cheetah or leopard.

In nerual network this is often done by calculating the softmax formula. By calculating the softmax you can get a vector of different values transformed to values between 0 and 1, so that they can be interpreted as probabilites. Such as what's the chance that the big spotted cat is a jaguar, leopard or cheetah. Then after calculating the softmax we can sort the array so that the highest probability goes first. By doing this we can get an idea whether the images contains a jaguar, cheetah, leopard or maybe even a wardrobe. Depending on which network it has been trained on. In our case we can then see that a picture of a leopard will end up being the most probable whenever we pass the picture of the leopard, the next probable will then be a cheetah, jaguar, tiger etc. And probably having the wardrobe at the very end of the array. This is also one of the reasons I chose the three spotted cats. Because they are so similar and I wanted to see whether the next likely candidate when passing a jaguar would be a leopard or a cheetah and not some other random class.

5.7.5 Inference in a nutshell

That is, in a nutshell, how inference is made in TensorRT. After building and loading the engine, one needs to process the input to fit the model before calling the inference function; execeuteV2 or enqueueV2 depending on whether one is running it synchronously or asynchronously. Then ending by sorting the most likely candidates into an array mapped to the classes that the model is trained with. The tricky part is managing the pre- and post-processing steps. Further on, we will look into how inference runs on the Tegra Xavier and how to get an overview and manage the SoC's power consumption.

Chapter 6

Tegra Xavier power overview and management

This chapter deals with the power management of the Jetson Xavier development kit. We will look into how we read the power data from sensors integrated into the Jetson Xavier before we look into how we can set different modes to control the power. Finally, we will look into the Jetson Clocks script.

6.1 Overview

An important aspect to look into when running AI workloads on the Tegra Xavier is managing the power. A lot of electrical instruments have to run on batteries with limited lifetimes. Imagine a drone flying over a field, taking pictures of the crops to look for plant diseases, as in the example earlier. The optimal would be for the drone to be able to do at least a whole day's work on one battery lifetime.

We also talk a lot about battery life on electric cars, so power management is critical when running an AI model to detect objects on the road or road lines. So creating models that reduce as much power consumption as possible will be optimal. However, in some instances, for example, when it comes to safety, there might be an emphasis on inference speed instead of power consumption. This is dependent on the system's goal; a drone that makes inferences on plant disease will likely emphasise power more than speed. It all depends on the goals of the system.

6.1.1 Monitoring on Jetson Xavier

The Jetson Xavier has many features that help improve power management and is integrated with a three-channel INA3221 power monitor[30], which is a monitor developed by Texas instruments and is a three-channel high-side current and bus voltage monitor. It senses Bus voltages from 0V to 26V and is used for computers, telecom equipment, battery charges and more[12]. The Jetson Xavier developer kits have integrated this INA3221 monitor and allow us to monitor and manage the power usage of the

Table 6.1: Jetson Xavier naming convention for sysfs nodes[30]

rail_name_<N>	Sets/get the rail name.
in_current<N>_input	Gets rail current in milliamperes.
in_voltage<N>_input	Gets rail voltage in millivolts
In_power<N>_input	Gets rail power in milliwatts
crit_current_limit_<N>	Sets/gets rail instantaneous current limit in milliamperes.
warn_current_limit_<N>	Sets/gets rail average current limit in milliamperes.

Where <N> is a channel number 0-2.

Xavier. In this thesis, we will use this monitor to access the power information of the Tegra Xavier when running inference. We will see how the GPU and CPU behave with TensorRT optimization and look into the power usage of the GPU especially, when offloading some layers on the GPU over to the NVDLA cores. We will now give further details on the INA3221 power monitor and how to take advantage of it on the Jetson Xavier.

This monitor provides us with information that can be read using sysfs nodes[30], a pseudo-file system on Linux that provides an interface to kernel data structures. On the Jetson Xavier, this can provide information that allows us to track the power consumption while running inference on the system. The following table is borrowed from the Nvidia documents page for the Jetson Xavier. It describes the information that will help us track the power, voltage and current[30]. See table 6.1.

The power monitor can be accessed at I2C addresses 0x40 and ox41[30]. At these addresses, we can read the voltage, current and power. We can read the sensors at the addresses
/sys/bus/i2c/drivers/ina3221x/1-0040/iio:device0,
/sys/bus/i2c/drivers/ina3221x/1-0041/iio:device1
Where we see the sensor for device 0 are given to us at address 0x40 and device 1 at address 0x41. The information we can read from these two devices is a little different. At device 0 we find the GPU, CPU and SoC power rails, and device 1 gives us the CV, VDDRQ and SYS5V power rails. Looking at table 6.2 we can see the address and the channels of the different sensors. For this thesis, we will mainly look at the power rails at the address 0x40, the CPU, GPU and SOC.

Accessing these rails can be done simply by using the cat command on the strings
/sys/bus/i2c/drivers/ina3221x/1-0040/iio:device0/in_current1_input,
/sys/bus/i2c/drivers/ina3221x/1-0040/iio:device0/in_voltage1_input,
/sys/bus/i2c/drivers/ina3221x/1-0040/iio:device0/in_power1_input.
These commands will give us the information in milliampere, millivolt and milliwatts. To be able to read these we will need sudo access.

By checking these rails while running Resnet50 with TensorRT, we can monitor the power consumption while building the engine and running inference. It will also be quite interesting to see how the inference is affected while running on DLA instead of on the GPU.

Table 6.2: Address and Channel for INA3221 on the Jetson Xavier[30]

Power Rail	Address	Channel	Power Rail	Address	Channel
VDD_GPU	0x40	0	VDD_CV	0x41	0
VDD_CPU	0x40	1	VDD_VDDRQ	0x41	1
VDD_SOC	0x40	2	VDD_SYS5V	0x41	2

6.2 Command line tools for performance and energy

On the Jetson Xavier, a couple of command-line tools exist to set the performance and energy characteristics of the SOC. Two of these tools are the jetson_clocks- and nvpmodel-tool. With these two tools, we can configure the SoC's energy usage to optimise performance. By configuring the frequency of the CPU, GPU or the DLA we can effectively define the power usage of the Xavier [30].

6.2.1 NVPModel

The command-line tool nvpmodel allows us to define the limitations of the power on the SOC. We have eight CPU cores on the chip, but not all of these need to be active at a given moment. When running at default mode, which is when the system runs on 15W, we have four CPU cores online, two DLA cores, and four GPU texture processing clusters(TPC)[30]. Of course, there are ways of setting different modes on the Jetson Xavier. This can either be done in the terminal using the command
$sudo nvpmodel -m [mode index].[30]
As an example, $ sudo nvpmodel -m 2 will set the power mode to the 15W default.

Nvidia also provides the user with a nvpmodel GUI, which can be accessed in the upper right corner of our screen if using a Linux-based operating system. Here one can easily change the mode with just a click of a button. See figure 6.1 borrowed from Nvidia [30].

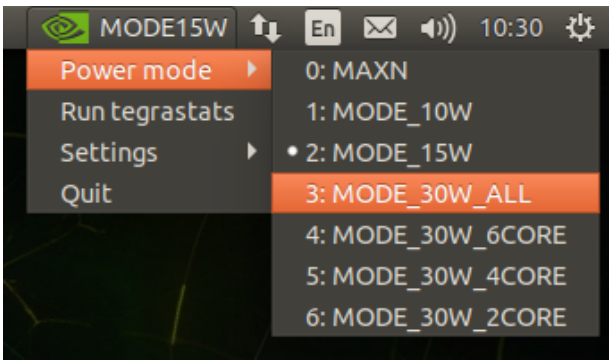

Figure 6.1: NVPMode gui[30]

Depending on the mode we set on the Jetson Xavier, the power output and frequency of the CPU, GPU and DLA will change. This should mean

Table 6.3: Nvidia power mode table.

Index	Power mode	CPU cores	GPU(TPC)	DLA cores
0	MAX	8	4	2
1	10W	2	2	2
2	15W	4	4	2
3	30W	8	4	2
4	30W	6	4	2
5	30W	4	4	2
6	30W	2	2	2

Where default mode 15W is highlighted in green

Table 6.4: Nvidia frequency table in Mega Hertz.

Index	Power mode	CPU max frequency	GPU max frequency	DLA max frequency
0	MAX	2265.5	1377	1395.2
1	10W	1200	520	550
2	15W	1200	670	750
3	30W	1200	900	1050
4	30W	1450	900	1050
5	30W	1780	900	1050
6	30W	2100	900	1050

Where default mode 15W is highlighted in green

if we run it to the fullest on the maximum power mode. We should get the highest speed optimization. That is, of course, useful if speed is the top priority; however, that is seldom the case on real-time software. So we will look further into how the different modes will run our Resnet50 model on the Xavier. Have a look at table 6.3 and table 6.4 for a more detailed description of the different modules. These tables are derived from an Nvidia webinar Dustin Franklin gave in 2019[15].

6.2.2 Jetson Clocks

The Jetson Xavier also provides a script for maximizing the device's performance by setting the maximum frequency for all the processors, CPU, GPU and EMC clocks. This script is called Jetson_clocks.sh and can be found in the /usr/bin/jetson_clocks address. This script allows us to maximize the power output and set the maximum PWM fan speed[30].

PWM stands for pulse width modulation, and PWM fans are fans found on some CPUs and GPUs. They are integrated circuits to control the speed of a fan and can provide cooling to the system based on the temperature. PWM works like a switch and turns on and off while controlling the power delivered to the fan[30].

So by using the jetson_clocks script, we can get some real power delivered to the system. Thus we believe it will help increase the time of inference. The script has four options: display the current clock settings; store the current settings to a file; restore that restores the saved settings;

finally, fan, which sets the maximum PWM fan speed[30]. It is a good idea to run the script with the store flag before planning to maximize the power output. If the user has not already done this, the restore flag will have no effect as there is no configuration file to get the data. Should this be the case, a system reboot will be required.

So to maximize the Jetson Xaviers performance, while in the unrestricted power mode, we can run the command: $ sudo /usr/bin/jetson_clocks.
However, should we want to maximize performance and fan speed we run the command: $ sudo /usr/bin/jetson_clocks - -fan.

6.3 Power tuning

With all that taken care of, we can test the Jetson Xavier's capabilities. We believe that by maximizing the power of the SOC, we will get a significant speed up on running inference and building the engine. However, we are just as interested in running the Xavier on a lower power mode, which will likely reduce the inference speed.

When talking about training deep learning models and running inference on these, we often talk about time, time of training and time of inference. This is because we want an algorithm that runs as fast as possible. These models are often run on GPUs connected to the power outlet, and because of this, speed becomes the main priority. However, as we have mentioned before, this might not be the essential aspect to look for in many real-time cases. Power consumption is as, if not in some instances, more important than the speed it takes to run inference.

Further on, we will use the tools described to benchmark inference on the Resnet50 model on the Jetson Xavier, using TensorRT. We will mainly look into the trade-offs between speed and power.

Opening the files that store the power rail information requires sudo level permission. So for running the program we have created, we will have to run the engine with sudo permission. Some more details of the C++ code for reading and monitoring the power consumption can be seen in appendix B.

Chapter 7

Results discussion

In this chapter, we will look into the results of the tests we have run on the system. We will see how the different precision has affected the speed and power usage of GPU, CPU and the rest of the SOC, as well as what benefits we might get from offloading to the DLA. We will start this chapter by introducing the tests before going into the results themselves. Finally, we will discuss and analyze the results that we have gotten.

7.1 Introduction to the tests

We have decided to compare four of the six power modes to analyze the Jetson Xaviers' inference speed and power usage. We will, of course, start looking into the 15W power mode, as it is the default. This will also be a good baseline to compare the other power modes.

We decided that analyzing all four 30W modes is unnecessary, and instead, we focus on the 30W with four CPU cores active. This is so that we get a better picture of the performance of each of the processing units. Since it has the same number of active cores as the default 15W mode with four CPUs, four TPCs and two DLAs, the only difference is the performance of the active processing units. So by running on 30W with four CPUs, we can compare the performance of the processors, and the difference will not be based on how many running processors there are but rather on the performance of the units themselves.

We will also look at the 10W mode, which only has two CPUs, two TPCs and two DLAs active. This is what could be called the power save mode and is expected to perform the slowest. However, if the speeds are still decent and the power saved sufficient, it might be a good alternative for specific problems. It runs on the same CPU frequency as the 15W mode but on half the CPUs and has a somewhat lower GPU and DLA max frequency.

To see the maximum performance of the Jetson Xavier, we will run tests on the unrestricted power mode for maximum power. This mode allows us to see what the Jetson Xavier is capable of. Here we expect to see some speed-ups in inference time, but of course, at the cost of increased power.

Lastly, we will run the Jetson Clocks script for the full effect. This script locks the clocks to their maximum as defined by their nvpmodel. We will

only run this script on unrestricted power mode, and this should give us the absolute performance of the Jetson Xavier. There are not many reasons to clock the other power modes to the top as the main reason for using them are their limitations.

7.1.1 How the tests are run

All the tests are run using the Resnet50 model, as explained in detail in chapter five. We can quickly change the different configurations by editing the script based on precision and active DLA. We have run the tests on all these configurations on the Jetson Xavier, one mode at a time.

The script runs inference on a collection of 1000 images of the three different cats, see figure 5.3. Depending on which cat is being classified, we have tested for correctness. So if it is a leopard, it will write out leopard, and if it is a cheetah, it will classify it as a cheetah. We have tested with all different configurations to see that changing to FP16 and DLA would create errors or mistakes in inference. However, it shouldn't because precision is not as crucial for inference as it is for training. We have also tested images of other objects, such as wardrobes, coffee cups and more. To make sure the code runs correctly. However, we use the spotted cats on the inference tests to have consistency.

By running inference on 1000 images, we can get a good average on the inference time and the power consumption. Then we ran the tests 20 times on each of the configurations at different times to ensure they stayed consistent and were not affected by other processes on the operating system.

Nevertheless, before we get into how TensorRT optimizes Resnet50 and runs inference on the GPU, we will first look at running an un-optimized version using TensorFlow. With TensorRT, running on the CPU is impossible because TensorRT optimizes the code for the GPU it will be running on. TensorRT uses CUDA to build the engine and run inference; thus, it only allows for GPU and DLA for the inference part. The final results will be for inference on the Volta GPU on FP32, FP16 and FP16 with DLA offloading. However, before that, we want to get a baseline for inference time by running the model on TensorFlow. First and foremost, we will look at how TensorFlow compares to TensorRT when it comes to inference time and the power running on the GPU on different power modes. However, we will also benchmark how the ARM CPUs compare to Volta GPU using TensorFlow and thus see why it might be preferable to run inference on a GPU compared to a CPU.

7.2 TensorFlow on the Xavier architecture

As mentioned when we introduced TensorFlow in earlier chapters, it is a Python framework developed by Google to train deep learning models. However, it allows us to load many popular deep learning models using Keras. This way, we can quickly get our Resnet50 model down and ready

to go. After loading the model, we can run inference on the same set of images as we would using TensorRT. We will try to keep the test as similar for TensorFlow as we would for TensorRT; this means running inference on 1000 images of spotted cats. We will try to do this about 20 times to get a variety of executions to get a more wholesome picture and ensure it is not affected too much by other processes on the operating system. However, we might have to adjust to the expected time increase while computing on the CPU.

7.2.1 Inference time

The first thing to notice while looking at the graphs 7.17.2 is the fact that we did not run inference on 10W mode on the CPU. This happened because it took too long to wait for the tests to finish. We only did the test once on 15W, which took us 12 minutes to run on the CPU, and two times on 30W. It was slow even on unrestricted mode, so we decided not to run it more than five times here. With that in mind, we can have a look at the graph.

First and foremost, on unrestricted performance, the inference time is about 223ms, leading to a little more than 3.5 minutes of total runtime. This is not that bad, considering we are on the CPU, but relatively slow. On 30W, we see that the average inference time is more than doubled, being 460ms which accumulates to 8 minutes on 1000 images. For 15W, we get an average time of 747ms, which is drastically bigger and spends 12 minutes to run on the entire data set.

When running on the GPU, we also saw that the time to run the tests got increasingly heavy. When we ran it on 10W, we saw that the inference took 360ms per image, which on 1000 images ends up being 6 minutes. 15W had an inference time of 245ms per image, which leads to a total running time of about 4 minutes, 30W had 185ms, and unrestricted had 145ms, which leads to a total running time of approximately 3 and 2.5 minutes, respectively.

Comparing the unrestricted power mode on the CPU and GPU, we see that the inference time only varies by about 80ms. However, this increases by about 80 seconds when working on 1000 images. So there is a good gain on running on GPU on more considerable data sets. When we go down in power mode to 30W, we start seeing a more substantial difference in inference time. On the GPU, we have 275ms gain compared to the CPU, which is an increase of about 4.5 minutes, and on 15W, the difference is 502ms. This accumulates to 8 minutes on the full data set. Finally, on 10W, we only have the GPU time, which is still 387 ms faster than the CPU on the power mode above.

7.2.2 Power usage

If we look to figure 7.3 we can see how much power we use running inference with TensorFlow. The GPU represents most of the power consumption in the highest power mode. However, when we go down in power modes, we see that the SOC starts taking over the majority, with the GPU following closely behind. The CPU keeps a stable reduction on

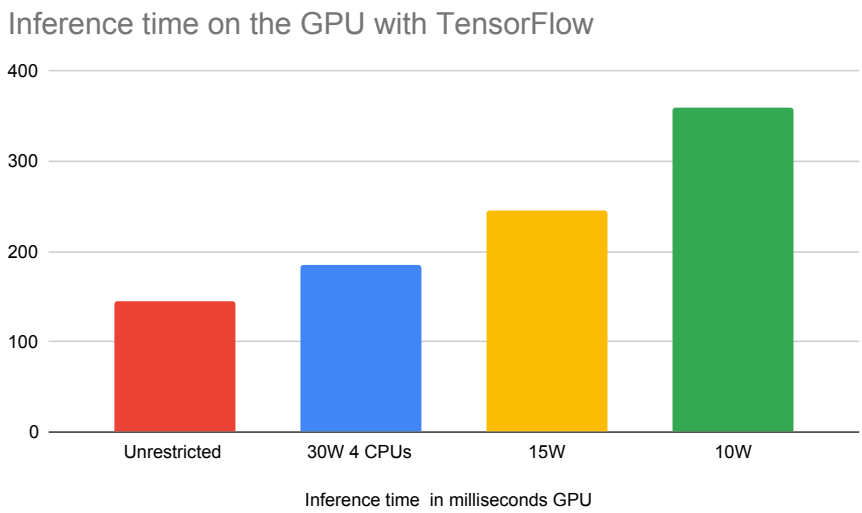

Figure 7.1: Inference time in milliseconds on the different power modes using TensorFlow on the GPU

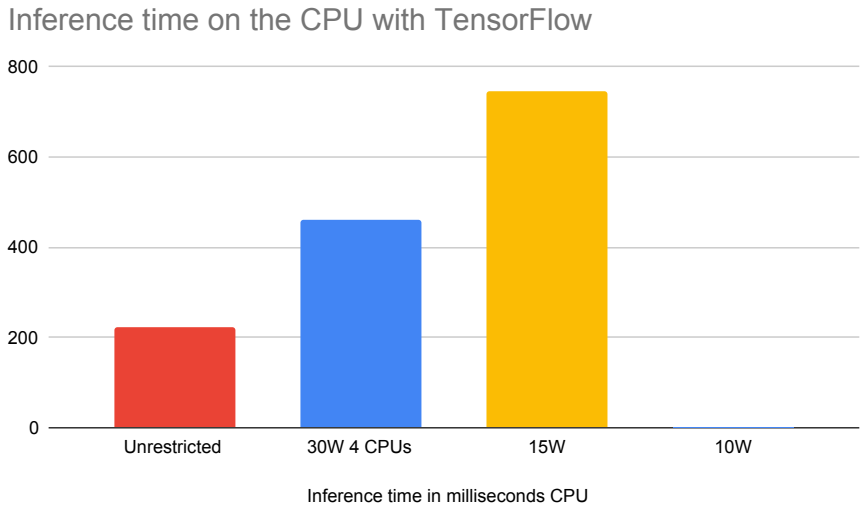

Figure 7.2: Inference time in milliseconds on the different power modes using TensorFlow on the CPU

all the power modes. This is likely because the CPU always have some background processes running. No matter how much we try not to affect the tests and is not affected by running inference.

If we compare these results to figure 7.4 we see that the CPU has had a massive increase in power usage. If we stick to the unrestricted power mode, the average power consumption is about 6500mW. This is a considerable increase compared to running on the GPU, where we only had about 2800mW on the CPU. Seemingly, the power consumed on the GPU is zero; this is not the case. However, it is almost irrelevant compared to the CPU and SOC power consumption. The GPU averaged 30mW on the highest power mode, 15mW on 30W and 18mW on 15W. The reason for it being about the same on 30W and 15W is that it was not doable to run many tests on these modes. Since the power is so low, it is likely due to only running background processes and not being affected by the inference.

On this graph, however, we only did five runs with the highest power mode, two on 30W with 4 CPUs and one on 15W. This is because the time of inference was way too high. 10W is zero because we saw that it was not feasible to run it here as the time it takes to do one test is too high. So the graph only gives us a simple illustration of how the power is consumed when we run on the CPU.

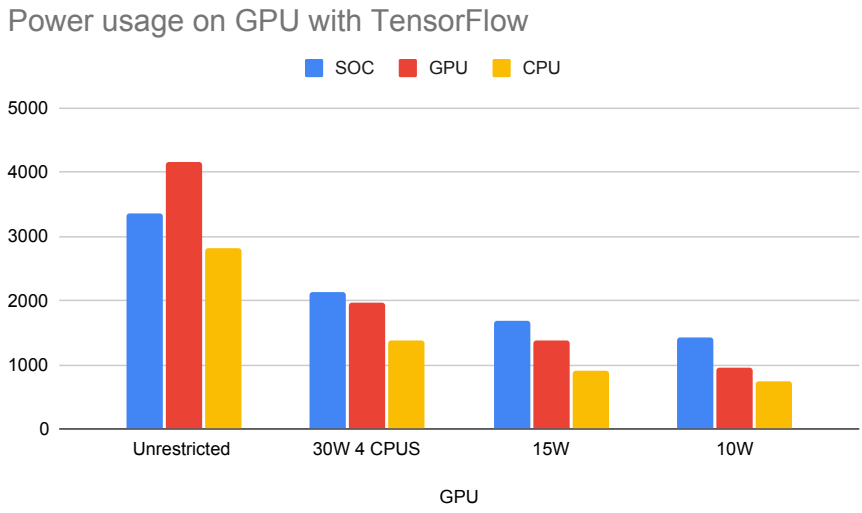

Figure 7.3: Power consumption in milliwatt on the different power modes using TensorFlow on the GPU

7.2.3 Takeaways from running on GPU and CPU

As expected, we saw a substantial increase in inference time as we ran the workload on the GPU compared to the CPU. The lower power mode we are working with also changes the inference time. However, on the GPU, the increase was not as drastic depending on the power modes as on the CPU. This was also as expected since we have fewer CPUs to work with.

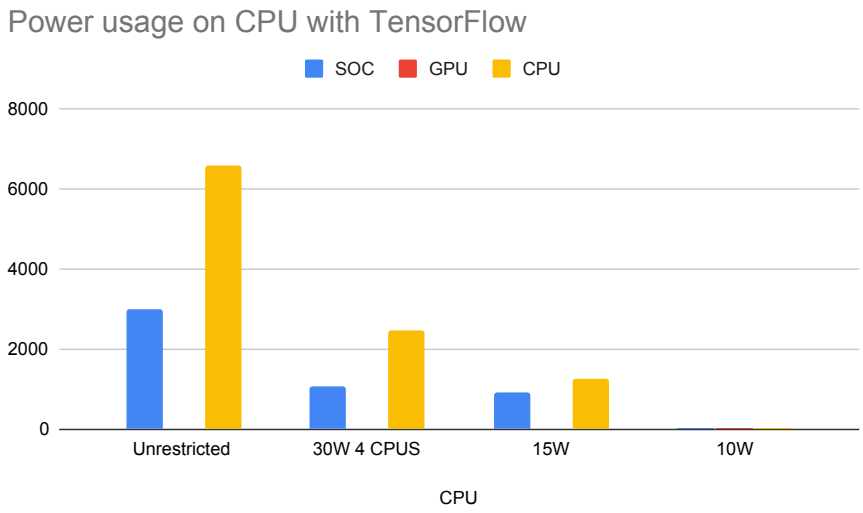

Figure 7.4: Power consumption in milliwatt on the different power modes using TensorFlow on the CPU

Having two CPUs instead of four will decrease the speed as we have fewer processors to run in parallel. They are working on lower frequencies, which also helps decrease the time. On the GPU, we have fewer clusters to help us, but we still have many more cores to run the code in parallel compared to the CPU. Which shows us that running inference is a parallel task best suited for GPUs

Inference on the unrestricted power mode is still possible on the CPUs. At least when working with a medium-sized workload, 1000 images. However, switching over to the GPU will give an increase in inference time. As we go into the lower modes, we see that running on the CPU is no longer a good option. A run time of 8 - 12 minutes on 1000 images is too slow, thus making the CPU a lousy option.

Even though running on the GPU is a lot faster, it is still relatively slow if we think of this in a real-time perspective. In the following sections, it will be interesting to see how TensorRT compares to TensorFlow.

We will also have a deeper look at the power usage compared to TensorRT. On TensorRT, we will also look into different precision levels and see how these different levels compared to that of TensorFlow.

7.3 15W mode

The default power mode on the Jetson Xavier is the 15w power mode and is an excellent place to start. On This mode, there are 4 CPUs active with a max frequency of 1200MHz, and it also has four GPU TP clusters active and two DLAs. The GPU has a maximum frequency of 670MHz and a DLA of 750MHz.

7.3.1 Inference time

The standard precision in TensorRT is the 32FP precision, and by looking at figure 7.5, we can see that it is the slowest of the three configurations. It has an average time of 24.5 milliseconds. We then see that running inference on FP16 reduces the inference time to 16.32 milliseconds, which can be pretty substantial in some instances. Especially in real-time systems where time is of the essence. On average, that is a time increase of about 33%. However, looking at the DLA, we can see that the time has increased a little. The DLA runs a little slower than when running with just the FP16 precision. The DLA averages 19.47 milliseconds, an improvement from the FP32 but still slightly slower than FP16.

7.3.2 Power consumption

Looking at figure 7.6 we can see how the different configurations perform considering the power consumption. We can also see which of the processing units consume the highest amounts of power. If we start on the FP32, we see that the power consumption on the GPU is very high. On average, it reaches the amount of 4033 milliwatts, which is almost twice the amount of the SOC and the CPU combined. Them having 1821.93mW and 629.95mW, respectively. It shows us that running inference is a GPU-heavy task, as expected.

In contrast to FP16 precision, we can observe that the GPU power consumption has decreased by over 50%, now down to 1672.75mW. Now more even with the other power rails. We see the CPU perform with an average of 633mW, about the same as on FP32. The SOC power rails have been reduced to 1506mW. Neither that much of a reduction in power consumption. This shows us how much of a GPU-heavy task running inference is and that by reducing the precision to 16 floating-points, we get some improvements to the power.

Lastly, we have inference when offloading to the DLA. Here we see even more reduction in the power on the GPU. Now it is down to 428mW, which is 9.4 times, almost ten times lower, than running with FP32. This is a considerable improvement. We also see that the power consumption on the CPU and the SOC rails have relatively small reductions, now being 1482mW and 540mW, respectively. However, still decreasing a little.

Now we will look into the other nvpmodes to see how the power and speed are affected. We imagine we will see the same trend of reduction. However, depending on the mode, we might have some more drastic improvements.

7.4 10W Mode

The 10W power mode has half the amounts of active CPUs and TPCs running and a lower frequency on the TPC and the DLAs than the 15W mode. We expect a slight increase in inference time on this mode but then working with lower effects.

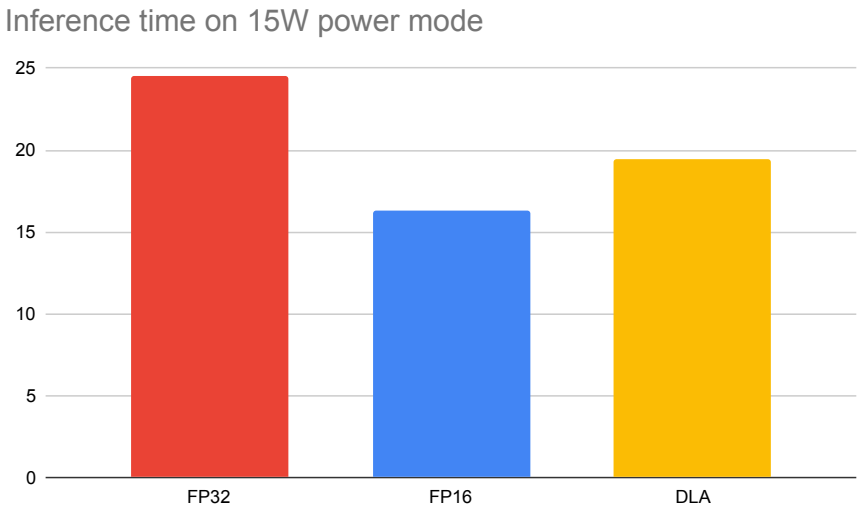

Figure 7.5: Inference time in milliseconds on 15W power mode

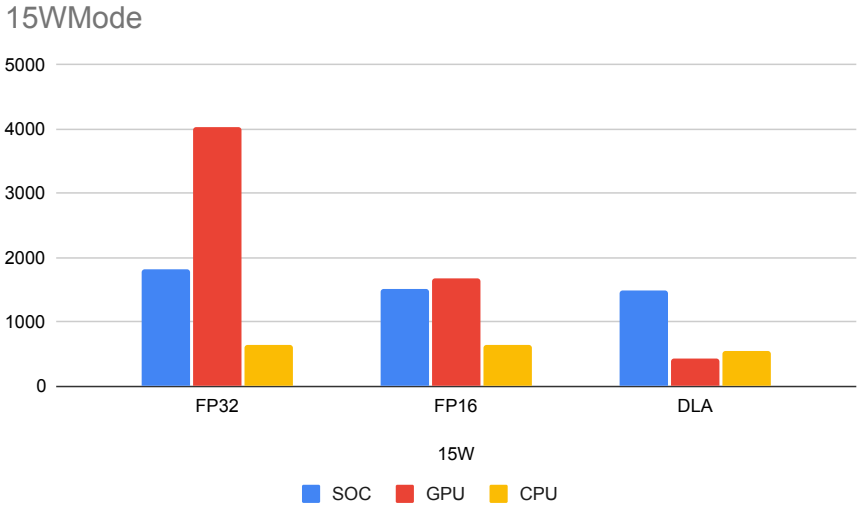

Figure 7.6: Power consumption in milliwatt on 15W power mode

7.4.1 Inference time

We see the same trends here in the 10W mode as in the 15W mode. The slowest configuration is expected while running on FP32, with FP16 being faster and DLA being a bit slower than FP16 but faster than FP32. FP32 averages out on 46.1ms, which is quite high. Almost double the amount of running on 15W mode. However, when changing to FP16, we are down to 18.05 ms, which is ca. 2ms slower than FP16 running on 15W. We see the same situation when running on the DLA. On 10W, it has an average of 23.78ms, again very close to the same situation on 15W. So by changing to a smaller precision and DLAoffloading, we get almost the same performance as the 15W mode. This makes sense when we consider that the DLA runs on both cores on both modes, and even though their frequency is slightly different, they are still able to offload quite a lot. The GPU also has fewer TPCs to run on. Because of this, reducing the precision should have a more substantial gain than on higher power modes. See figure 7.7

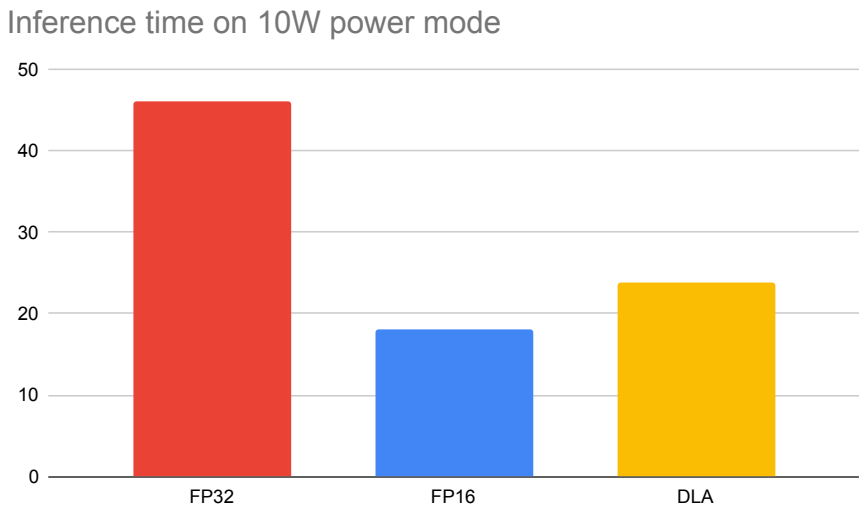

Figure 7.7: Inference time in milliseconds on 10W power mode

7.4.2 Power consumption

Looking at figure 7.8 we see that the power reduction on the GPU from FP32 to FP16 does not have the same considerable power reduction as in nvpmode 15W, which decreased by over 50%. Here the decrease in power is no more than 30%, going down from 2127.9mW to 1502mW. Offloading to the DLA gives us a GPU power of 405mW, which is a significant improvement of about five times the amount of running on FP32. However, compared to 428.35mW from 15W, the gain is not all that much. The same goes with FP16, which is 1672.75mW on 15W and 1502mW on 10W. So the real power gain while running on 10W compared to 15W mode is if we run inference on FP32.

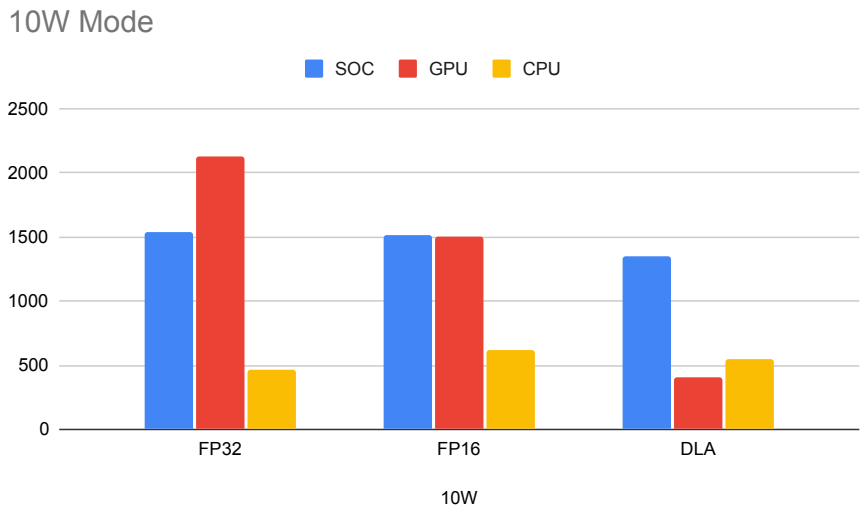

Figure 7.8: Power consumption in milliwatt on 10W power mode

7.5 30W Mode

As mentioned earlier, we decided to run the test on 30W mode with 4 CPUs. The reason is that we have the same amount of CPUs, TPCs and DLA cores as on 15W mode but running on a higher frequency. Here we have four CPUs, four TPCs and 2 DLA cores active, all running on higher frequencies, see tables 6.3 and 6.4.

7.5.1 Inference time

30W mode is a high power mode and should produce some faster inference times as we expect. We can see in figure 7.9 that we have faster inference times on both precision modes and DLAoffloading. FP32 performs quite fast with 20.2ms, which is slightly faster than 15W. Approximately 4ms on average. This is quite a small decrease in time but may be important in certain scenarios where time is of the essence. We see that the FP16 and DLA are 15.8ms and 15.95ms, which compared to 16.34ms and 19.47ms is not that huge of an increase. However, what is quite interesting here is that the inference speed on the DLA evens out with that of FP16. This is a good sign for the DLA should the power decrease follow the same trend as we have seen on 15W and 10W. We will have a look at that in the next section.

7.5.2 Power consumption

The first thing we see looking at figure 7.10 is that it indeed follows the trend we have seen earlier. Here we see that DLA decrease the GPU time from FP32 by about 90%. FP32 has a GPU power consumption of 6013.75mW down to 527.3mW, which is significant. FP16 has a GPU power consumption of about 1821.65mW, which is a decrease of approximately

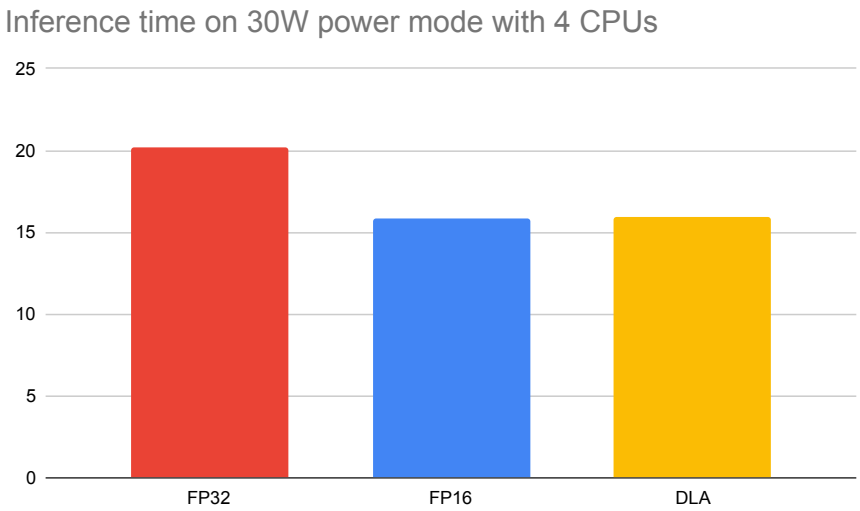

Figure 7.9: Inference time in milliseconds on 30W power mode with 4 CPUs

70% from FP32 and is around 70% higher than that on the DLA. We also see that the SOC power rail gives us powers from 2540.15mW on FP32, 1548.9mW on FP16 and 1887.8 on the DLA. Keeping relatively stable. The same goes for the CPU, which gives 718.1mw on FP32, 750.2mW on FP16 and 579.6mw on DLA, which is relatively stable, having somewhat lower power consumption when running layers on DLA. So the main power benefits are on the GPU, which is what one would want running inference.

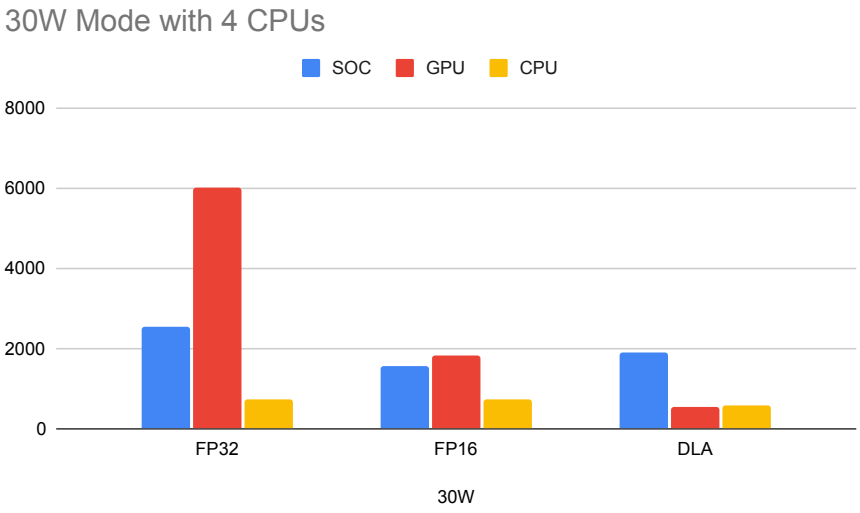

Figure 7.10: Power consumption in milliwatt on 30W power mode with 4 CPUs

7.6 Unrestricted power mode

The final power mode is for maximum performance, which we call unrestricted. As shown in table 6.3 and 6.4, all the CPUs are online, and the GPU works at full capability. This should increase inference time as we have more active processors able to work on higher frequencies. However, we do expect an increased power usage.

7.6.1 Inference time

The inference time in the unrestricted mode can be seen in figure 7.11 and shows an increase from 30W. The increase on FP32 has, on average, gone down from 20.2 ms to 17.78, giving an 11.1% faster inference time per image. We now see that the inference time increase is starting to flatten out. Also, when we look at FP16 compared to FP32, we can see that the time gain is not as dramatic as we have seen earlier FP16 runs on average 15.90ms, which is about the same as the 15.8 on 30W. We are seeing an increase in DLA performance now running faster than that of FP16 only on the GPU. The DLA runtime is now down to 13.89ms. This is quite interesting, considering that the DLA has been slower than the FP16 on GPU up until this point. However, we still see that the gain we are getting is starting to be less significant from power mode to power mode.

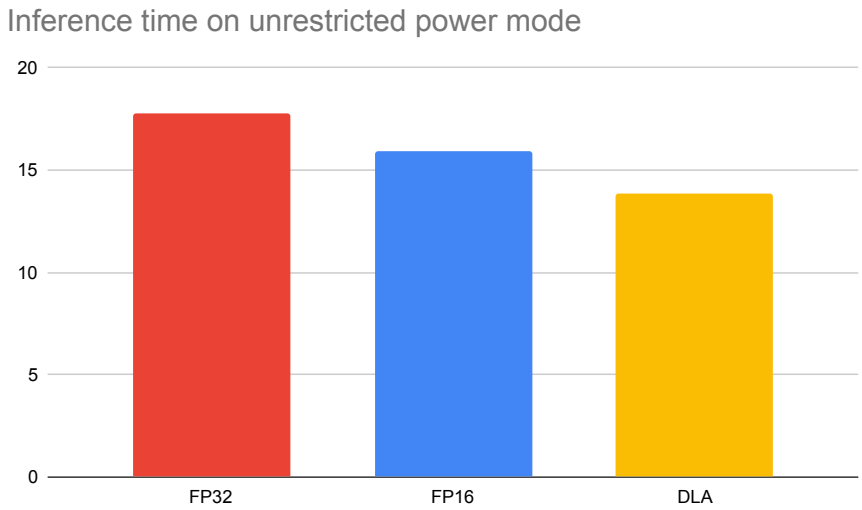

Figure 7.11: Inference time in milliseconds on the unrestricted power mode

7.6.2 Power consumption

Figure 7.12 shows some extreme variations in GPU power usage looking at FP32 compared to that of FP16 and DLA. FP32 has an average power usage of 9894mW on the GPU, an increase of about 40% compared to 30W. This is not a good trade-off compared to the inference time we gain from

30W to unrestricted. Using FP32 on the unrestricted mode is not optimal for running Resnet50. However, running FP16 on mode 0 seems to have similar results to 30W. On this mode, FP16 averaged out on 1953mW and compared to the 1821.65mW on 30W, the trade-off might be worth it in some instances. The real power gain, however, comes while running on the DLA. Here we have reduced the GPU power consumption to 538mW, approximately the same as that of 30W, which was 527mW. Since running on the DLA was even faster than FP16, it seemingly has some excellent gains in this mode. However, the power of the SOC itself increases by about 1100mW from only running FP16 on the GPU. This is as expected since the SOC power contains all the power used by the system that is not on the CPU or GPU—depending on how important it is to offload as much as possible from the GPU. Running inference on the DLA with unrestricted power might perform best.

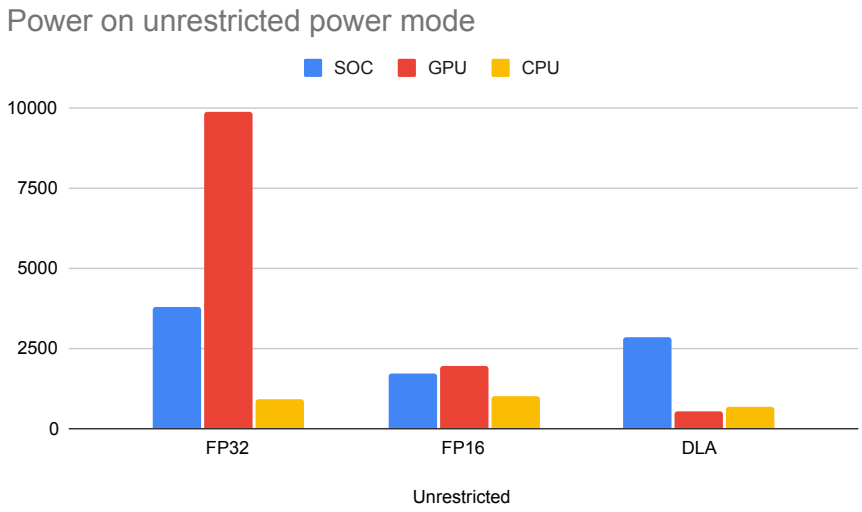

Figure 7.12: Power consumption in milliwatt on the unrestricted power mode

7.7 Jetson Clocks

Jetson Clocks is a script that sets the frequency and power output to the maximum of the power mode we are on. So by running Jetson Clocks with the unrestricted power mode. We get the absolute performance of the Jetson Xavier considering speed. This will set the frequency to 2265.5MHz, the GPU 1377MHz and DLA to 1395.2MHz. It would set the maximum frequency on default mode to 1200MHz on the CPU, 670MHZ on GPU and 750 on DLA. We have decided only to test this script on the highest power mode. With this script, we want to push the performance to the limit, so using this script with power modes that limit the maximum performance does not make sense.

Running the Jetson Clock script on this mode will allow us to see the absolute performance of the Xavier architecture while running Resnet50.

7.7.1 Inference time

If we start by looking at the inference time on figure 7.13 we can see that FP16 is working faster than DLA again. This makes sense, considering the GPU is now working at total frequency. If we now compare DLA time to FP16 time, we can see that the DLA has only increased by about 1ms, now 12.52ms. This is not much of a gain from running without Jetson Clocks. Nevertheless, FP16 now has increased by about 7ms, from 15.96ms to 8.8ms, which is a decent speed gain. The time of FP32 has increased by about 3ms now down to 14.45ms. So again, we are not gaining that much of a speed up anymore on this precision.

The precision and DLA performance are also not too different compared to each other. The gain comes when we look at FP16, which is 6ms faster than FP32 and 4ms faster than offloading to DLA.

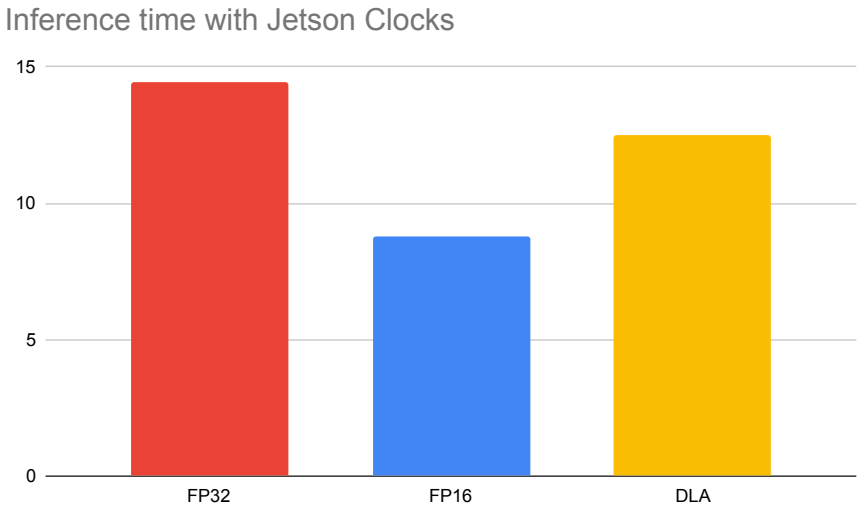

Figure 7.13: Inference time in milliseconds using Jetson Clocks on unrestricted power mode

7.7.2 Power consumption

When we ran the workload on FP32, we utilized a considerable amount of the GPU power. As one can see in figure 7.14 we use on average 14082.95mW on the GPU. Considering the trend we have been seeing from 10W up until now, it fits well. It has gotten very high, and by using a lower precision, we again see a drastic decrease in the GPU's power. On FP16, we use on average 6623.1mW and, as expected, even lower on the DLA, now as low as 1506mW.

Regarding the CPU and the SOC, we see that it keeps rather stable between 4000mW and 3000mW on the SOC and 1700mW and 1000mW on the CPU.

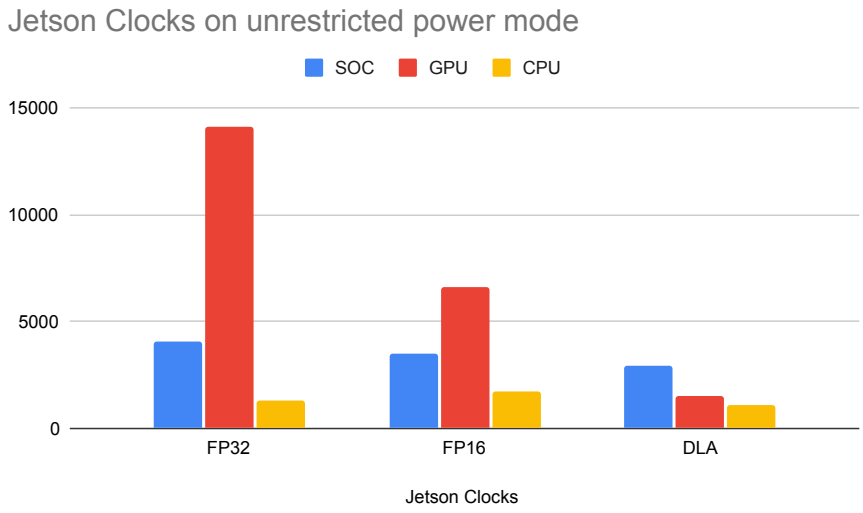

Figure 7.14: Power consumption in milliwatt with Jetson Clocks on unrestricted power mode

7.8 Standard deviation while running inference

When we are testing the inference speed and power consumption on the Tegra Xavier we also need to think about what other processes are being used on the operating system. These processes can give fluctuation in the power consumption done on the different processing units. We have found the average time and power usage while running inference on 1000 images. We have done these tests twenty times do get a good sample for our tests. To see if there is any major variations we have looked at the standard deviation of these tests, which can give us an indication of how stable the data collected is. With a low standard deviation we can assume, that the time and power usage are not too affected by other processes running on the operating system. Since the operating system has to deal with a lot of background processes, and we think that these might affect the data to a certain degree. We will first and foremost check the standard deviation on the unrestricted power mode.

Overall the standard deviation was quite small. The highest we found was at the GPU when running inference with FP32, this had a standard deviation of 1263.55mW and a mean of 9894.45mW a coefficient variation of 0.12. Seeing that the highest of the standard deviation was quite far away from the average, shows us that on general running inference on the system is quite stable. The background processes of the operating system does not affect the performance of the inference by much.

This shows us quite a low level of variation in our data, and is really positive to see. We made the decision to not put the standard deviation on the graphs considering how small they were.

7.9 Discussion

For this section we are going to discuss the results we have seen so far in more detail.

7.9.1 TensorRT compared to TensorFlow

Comparing the results from TensorFlow and TensorRT, we see that the inference time has increased substantially. First, comparing TensorFlow to TensorRT on 15W mode, we see that the speed has increased from 747ms when running on the CPU down to 245ms when running TensorFlow on the GPU, which is about half a second per image. However, when running with FP32, which is the slowest precision in TensorRT, we are down to 24ms. FP16 being 16.34ms and DLA being 19.47ms, we see that TensorRT can speed up inference. This is expected because TensorRT takes the neural network and combines the different layers into fewer, reducing the total size of the neural network. This will, of course, reduce the overall computation that is needed to be done. On FP32, we get a speed up of 31 times that of running on the CPU and about ten times the speed up of the GPU. Since TensorRT works on the GPU, this is the most interesting to compare. There is no need to compare TensorFlow to TensorRT on all the different power modes, as we saw the same tendency. However, we can conclude that TensorRT speeds up the model substantially. We also saw that the GPU is the best processing unit for running inference on the Resnet50 model. However, on unrestricted power mode with TensorFlow, the CPU managed to do a decent job by only being half as slow as the GPU. However, it is preferable to use the GPU for inference.

When we compare the power consumption of TensorRT to TensorFlow, we see that TensorFlow consumes less power than FP32 on TensorRT if we consider the GPU. With the CPU and the SOC power rail, we see that the power usage was quite similar. If we take 15W mode as an example, we see that TensorRT spends 4033mW of GPU power while running the workload. TensorFlow, on the other hand, only uses 1368mW. However, the difference is much lower when we reduce the precision to FP16. Now the power consumed was 1672.75, which is quite comparable, and by offloading some of the layers to DLA, we see that we can reduce the power down to 428mW, which is almost a whole Watt reduction from TensorFlow. That being without much higher power consumption on the SOC as a whole. This reduction can be generalized to all the power modes on the Xavier. The highest mode has an almost 0.5W increase on TensorRT as opposed to TensorFlow.

Another interesting thing is that while running the workload on the unrestricted power mode with TensorFlow, we have a power consumption

that competes with TensorRT running on 15W. However, the speed loss we get in this scenario, 223ms down to 24ms, still makes TensorRT the preferable choice. When default mode TensorRT is ten times as fast as TensorFlow in the unrestricted mode.

So as a conclusion to this section, we can see that TensorRT is such a substantial improvement to TensorFlow that it is better in most scenarios. Since TensorRT easily allows us to change precision, we gain many benefits considering speed and power.

As a side note, this thesis has not looked into ways of improving inference speed with TensorFlow. So there might be ways to get the speed of inference and power usage down by configuring the model in TensorFlow. However, since this thesis focuses on the Xavier architecture and TensorRT being an Nvidia framework to improve deep learning models, we saw it as sufficient to compare TensorRT to a default version of Resnet50 running with TensorFlow.

7.9.2 The benefits of FP16 precision and DLA offloading

As we have seen, reducing the precision down from FP32 to FP16 reduces the time of inference and power consumed on the GPU. The reduction is somewhat expected as FP16 being at a lower precision also takes less space in memory. This leads to the GPU being able to process the images faster, and taking up less space reduces the power needed to compute. The precision level is also not very important while running inference, unlike in training. This reduces the precision without losing correctness and gives us the benefits of higher speeds and lower power usage.

Regarding speed optimization, we saw that the FP32 was quite fast in itself. Using 15W mode as an example, the speed was only reduced by about 8ms when we optimized for FP16. We also saw this tendency on all of the different power modes. We got a speed up but not as much as we might have wished for. When we ran on FP16 but offloaded some of the layers to the deep learning accelerator, we saw a speed decrease compared to FP16 on the GPU. However, compared to FP32, it was still faster.

The real benefits of reducing the precision came from power optimization. On 15W mode, we reduced the power consumption by almost 2.5 times on the GPU. As we looked into in the previous chapter, we saw that it was mainly the GPU that significantly reduced power consumed. The CPU and SOC often got a reduction as well, but seemingly it was not affected much by the level of precision we chose. This makes sense because TensorRT utilizes CUDA for running inference.

7.9.3 The trade-off between power and speed

The different power modes give us various power options for the Xavier. It allows us to set a cap on the amount of power the SOC is allowed to use. If we want to limit the power usage to a minimum, we can set it to 10W mode. This gave us some excellent ways of dealing with different scenarios.

As we saw from the results in the previous section, this gave us a power reduction of about 12000mW on the GPU with FP32 precision while running without any power restriction with Jetson Clocks. It did increase the inference time by about 3.5 times. Nevertheless, in many cases, this might be a beneficial trade-off.

For example, using a robotic lawnmower would not need to run at maximum speed all the time; it is much more beneficial to have it spend longer doing inference and save up on power. This would increase the battery lifetime, and we would argue is preferable for a lawnmower. Another example of times when power consumption would be the priority is if we imagine logistic robots sorting boxes and crates in big storage facilities. These robots would have to move relatively slowly, so it does not need to have an inference time that allows them to process 30 frames per second, 7 to 10 frames per second is probably sufficient. Having a longer life on these robots is, of course beneficial.

On the other side, if we have an object detection system in a car, detecting hindering objects. Then the inference speed would probably be the priority, especially in cases where personal safety is at stake. In this system, we argue that inference time that allows for higher frames per second is preferable. A system like this might want to run without power restrictions and capped with Jetson Clocks. Nevertheless, there are other AI systems in vehicles than detecting objects on the road.

An example could be sign classification, a similar problem that the Resnet50 model is designed to solve. Here we do not need high-speed inference all the time. If the model aims to alert the driver of speed limits, construction work or other information that might be useful to the driver, we could argue that saving power trumps inference speed. Nevertheless, having too low inference speed would not be optimal. If the speed is too slow, the signs might not get enough time to process. Because one moves 22 meters per second while driving at 80KM/H, and much information needs to be processed, the trade between speed and power usage is something that always needs to be addressed.

The Xavier architecture gives us many ways of dealing with the battle between power and speed. By setting the maximum allowed power consumption by using the different power modes, we can find ways to optimize the models to the fullest. This means we can optimize the neural network without being afraid of exceeding the power limit.

We imagine that in most cases, one would want to limit the power consumption to a certain degree, except in those cases where speed is of absolute value. However, this varies from situation to situation.

7.10 Limitations of the tests

These tests have given much insight into optimizing deep learning models with TensorRT on the Jetson Xavier development kit. This allows us to develop software that works optimally with the architecture of Tegra Xavier. We have worked with the Resnet50 model to optimize with

TensorRT to test the capabilities of the Xavier, and Resnet50 is a medium-sized model. When we have worked with a workload of classifying 1000 images, we used quite a lot of power at times, especially while running Jetson Clocks. However, we never managed to cap the maximum wattage on the system. It would have been interesting to see how the system reacted to that.

However, this did not affect how the SOC behaves on the different modes, and we were able to get good data on the system's behaviour. Nevertheless, it would have been interesting to see the behaviour when pushed to the absolute maximum.

Chapter 8

Summary & Conclusions

In this chapter, we summarise the work done for this thesis and make some conclusions on how this can benefit the development of deep learning software on heterogeneous systems-on-chip.

8.1 Summary

The Nvidia Corporation is known as one of the leading companies when it comes to the production of graphic cards. However, they have also become a leading voice in artificial intelligence research in the last decade. Since deep learning training and processing benefit so much from the parallelism capabilities of the graphics card, it was only natural for Nvidia to take part in this trend.

With the NVIDIA's development of their heterogeneous system on chip, which has been integrated into everything from medical instruments to robotics used in manufacturing, we see the need to understand the different components of the architecture and their usage.

The Xavier architecture is an integrated heterogeneous system-on-chip. It has eight ARM Carmel CPU cores, and a 512-core Volta GPU with 64 tensor cores specialized in AI computing. It also has two Nvidia Deep learning accelerators, which have been created to offload the GPU while running inference on deep learning models. In this thesis, we have looked into how to utilize mainly these components because they are specialized for deep learning.

To fully utilize the architecture, we looked into Nvidia's frameworks, such as CUDA and TensorRT. TensorRT is the primary tool for running inference and is developed to optimize deep learning inference by creating a TensorRT engine. This engine is created through a build phase which parses an ONNX or Caffe file. These files are used to serialize already trained networks, to be easily portable. They are also generalized so different frameworks can utilize them, whether it is Pytorch, TensorFlow, OpenCV or TensorRT. For this thesis, we used pre-trained models stored in the ONNX format.

To begin with, we parsed the pretrained Resnet50 ONNX file. This phase is called the build phase, where we build the TensorRT engine. This

is done by specifying the workspace and the precision level. Here we can also tell the engine to run specific layers on the deep learning accelerators. After that, we save the engine on the disk for later use.

We then load the engine so that we can run inference. Here we pre-process the data we want to run inference on. Since we utilized the Resnet50 model, the data being processed was images for image classification. For this OpenCV was a great framework to use. Since it allowed us to easily parse the images and make them ready for TensorRT. OpenCV has an additional library called OpenCV with CUDA, which allowed us to transfer the data over to the GPU and let us do the pre-processing steps here. We preferred to run this on the GPU since processing images is a task that can be parallelized. The images needed to be resized to fit the standard for Resnet50, 216x216 pixels. After this, we ran the inference synchronously on the GPU using TensorRT. This produced some massive speed gains compared to running the un-optimized file.

When running inference, we tested with different precisions. We saw that we got a bit of speed up running on FP16 compared to FP32, which was the standard. Also, by transferring layers over to the DLA with FP16, we saw a speed up compared to FP32. However, compared to just running on FP16, we saw that the speed, in general, was reduced by a little bit. However, the true gain of running on the DLA came in the face of power consumption. We expected a certain improvement in the power, but we were surprised that the DLA would give such a significant power reduction.

We also tested the different power modes on the Jetson Xavier. Out of the six different modes, we decided only to test four of them—these where the 10W, 15W, 30W with four CPUs, and the unrestricted power mode. We wanted to compare the different power modes to compare the speed gains to the power gains, as well as how the different modes were improved by running with different precisions and the DLA. By this, we mean whether the power gain was consistent with the power modes or if we got some really good performance optimization on different modes.

8.2 Conclusion

The problem statement defined for this thesis was to understand the Nvidia Tegra Xavier architecture and utilize it along with TensorRT to optimize deep learning models with an emphasis on speed and power. As we have seen based on the results from the previous chapter, TensorRT can be used to speed up the original model to a significant degree.

We conclude that using TensorRT on the Nvidia Tegra Xavier can optimize deep learning models with better power management and faster inference time than on frameworks such as TensorFlow. TensorRT is a framework developed by Nvidia for optimizing deep learning models, and we saw that it is a great tool to use with the Xavier architecture. It allowed us to take advantage of Xavier's architecture, emphasizing the GPU and the deep learning accelerator. The Volta GPU was created with deep learning in

mind, and by changing the precision level to FP16 for the computations, we saw an increase in both speed and power usage, as opposed to the standard FP32.

The deep learning accelerator has been of significant interest in this thesis. It was developed as a way to optimize deep learning inference on the Xavier. We saw that it significantly reduced power consumption on the GPU, without increasing the overall SOC power consumption as a whole. The tests done in this work have shown that the NVDLA has lived up to its expectation and can improve the overall workload of deep learning inference.

Nvidia Tegra Xavier also comes with different power modes that let us cap the maximum power that the system allows. We have seen that this can significantly reduce the power consumption on the Xavier. By testing out the different power modes, we have seen that they can be used to set limitations to the neural networks we want to run and allow us to specify the amount of power the system is allowed to use. We conclude that they can be a great tool if we want to limit our system in cases where power reduction is of significant interest. This includes cases where battery life is essential.

With the Jetson Clocks script and running without any power restriction, we can get the most out of the system. By doing this, we got the system to optimize the inference speed all while allowing us to manage the power consumed. By allowing the system to use more power, we also improved the time it took to run the entire workload.

With all this in mind, the Tegra Xaiver is a heterogeneous system-on-chips which allows optimizing deep learning models to a high degree. By utilizing frameworks developed by Nvidia, such as CUDA and TensorRT, in line with the knowledge of the SOC architecture, we can get the best performance out of the different deep learning scenarios that face us.

8.3 Contribution

With this thesis, we have made several contributions to understanding the Nvidia Tegra Xavier architecture, emphasizing issues we face with deep learning neural networks. Deep learning has become a tool to solve issues related to artificial intelligence. These problems range from autonomous driving and object detection to more straightforward gadgets such as autonomous lawnmowers and logistic robots. Deep learning is likely to expand even further in the future. By utilizing system architectures specifically designed for these purposes, we can make systems that are safer, faster and low power usage.

By understanding the components of the Xavier explicitly designed for deep learning, we have been able to explore where we can gain improvements in inference speed and power consumption. We have explored how the deep learning accelerator drastically can reduce the time of inference and power consumed by the system. Using TensorRT to define and configure how the deep learning models should run, we have

seen how we can use these tools to make good designs depending on the problem. This contributes in allowing programmers to create software that are optimized for deep learning on heterogenous systems-on-chip.

We have performed several tests to see how the power is used on the Tegra Xavier; by trying out different precision and utilizing the DLA, we have seen that we can implement TensorRT engines that are optimized for the Xavier. This has given insight into how to develop deep learning software that takes full advantage of the Tegra Xavie architecture.

8.4 Future work

Throughout this thesis, we have examined how the Nvidia Tegra Xavier architecture has been optimized for machine learning, especially how the different components work when using TensorRT on deep learning models. We have seen how TensorRT speeds up the Resnet50 model compared to running it on TensorFlow. Also, we have seen how the power can be offloaded between the GPU, CPU and the DLA. This has shown that by offloading layers from the Resnet50 engine, built with TensorRT, can give significant power benefits. Also, by reducing the precision level, we have seen a reduction in inference time, which has given us an indication of the Nvidia Tegra Xavier's capabilities.

However, further work on the subject should be considered, and in this section, we will look into topics of interest.

8.4.1 Nvidia Orin

However, as of July 2022, the Jetson AGX Orin series will be available [13]. Future work would be to benchmark this new architecture, which Nvidia claims to be eight times faster than the Jetson AGX Xavier[13]. The new Orin will have an AI performance of 200-275 TOPS compared to Xavier's 30-32. Depending on whether one looks at the 32GB or the 64GB Orin, it will utilize 8-12 Core ARM Cortex CPUs, a 1792 - 2048 core GPU, with 56-64 tensor cores. This new GPU is based on the Ampere architecture[22]. It has 2 DLA cores of NVDLA version 2. According to Nvidia, Orin hits 275 TOPS at INT8[13] as we could not run any TensorRT engines on the Jetson Xavier with INT8 precision as the architecture did not support it. This would be interesting to test on the Orin module.

8.4.2 PCIe express

The Jetson AGX Xavier developers kit is equipped with PCIe generation 4. PCIe 4.0 provides a 16GT/s bit rate and is a full-duplex, which means it can send and receive data simultaneously and has a bandwidth of 64GB/s.

So by this standard, the Jetson Xavier should be able to send and receive quite a high level of data. It would be interesting to see what the platform is capable of when connecting multiple Xaviers over PCIe. If we utilize some bigger workloads, we might be able to get some speed up and power reductions by connecting multiple Xaviers.

8.4.3 Bigger workloads

There are many AI models out there. We utilized the Resnet50 model because it is widely used as a base for many other neural networks. It was also big enough for us to test out the capabilities of TensorRT and the Xavier architecture. However, a wide variety of other models would be interesting to try out. Resnet50 is an image classification model, which means it takes information from an image to classify what it consists of. As mentioned, this could be to classify cancer cells in patients, plant diseases and certain objects like Cats, dogs and Leopards. However, it is not an object detection model. An object detection model finds different objects in the image; this is very useful in, for example, autonomous driving or just for safety in cars to see hindrances on the road.

An example of such a model would be the Yolo algorithm. Yolo is an acronym which stands for "you only look once". This is considered to be the state-of-the-art algorithm for object detection[1]. With the release of Yolo version 4 in 2020, this could be a great contender for bigger deep learning models to test with TensorRT and the Jetson Xavier.

Appendix A

Asynchronous and synchronous inference

A.1 Asynchronous inference

The code belowr shows how to run inference asynchronously. As explained earlier we decided not to use asynchronous inference.

```
vector<nvinfer1::Dims> input_dims;
vector<nvinfer1::Dims> output_dims;
//Input and output buffers
vector<void*>buffers(m_engine->getNbBindings());

for (size_t i = 0; i < m_engine->getNbBindings(); ++i)
{
    auto bindingSize = getSizeByDim(m_engine->getBindingDimensions(i))
        * batchSize * sizeof(float);
    cudaMallocAsync(&buffers[i], bindingSize,m_cudaStream);

}

resizeAndNormalize(image, (float*)buffers[0], m_inputDims);
m_context->enqueueV2(buffers.data(), m_cudaStream, nullptr);
calculateProbability((float*)buffers[1], m_outputDims, batchSize);
for (size_t i = 0; i < m_engine->getNbBindings(); i++)
{
    cudaFreeAsync(buffers[i], m_cudaStream);
}
auto status = cudaStreamSynchronize(m_cudaStream);
if(status != 0)
{
    std::cout << "Unable to synchronize cuda stream" << std::endl;
    return false;
}
return true;
```

The code belowr shows how to run inference synchronously. This is the code used in this thesis and can be found at https://github.com/Joa2506/Resnet50.

A.2 Synchronous inference

```
vector<nvinfer1::Dims> input_dims;
vector<nvinfer1::Dims> output_dims;
//Input and output buffers
vector<void*>buffers(m_engine->getNbBindings());

for (size_t i = 0; i < m_engine->getNbBindings(); ++i)
{
    auto bindingSize = getSizeByDim(m_engine->getBindingDimensions(i))
        * batchSize * sizeof(float);
    cudaMalloc(&buffers[i], bindingSize);

}

resizeAndNormalize(image, (float*)buffers[0], m_inputDims);
m_context->executeV2(buffers.data(),nullptr);
calculateProbability((float*)buffers[1], m_outputDims, batchSize);
for (size_t i = 0; i < m_engine->getNbBindings(); i++)
{
    cudaFree(buffers[i]);
}

return true;
```

Appendix B

Monitoring power on the Tegra Xavier

All the code in this appendix are snippets from the code used in this thesis.
It can be found at https://github.com/Joa2506/Resnet50

```
//Functions and variables for power monitoring
string powerRailCPU =
"/sys/bus/i2c/drivers/ina3221x/1-0040/iio:device0/in_power1_input";

//Lists to read power
std::vector<int> powerListCPU;

string powerRailGPU =
"/sys/bus/i2c/drivers/ina3221x/1-0040/iio:device0/in_power0_input";

std::vector<int> powerListGPU;

string powerRailSOC =
"/sys/bus/i2c/drivers/ina3221x/1-0040/iio:device0/in_power2_input";

//Lists to read power, current and voltage
std::vector<int> powerListSOC;

//outputfiles to store the power read from the sensors
string powerFileSOCFP32 = "outputfiles/powerSOCFP32.txt";
string powerFileSOCDLA = "outputfiles/powerSOCDLA.txt";
string powerFileSOCFP16 = "outputfiles/powerSOCFP16.txt";

string powerFileCPUFP32 = "outputfiles/powerCPUFP32.txt";
string powerFileCPUDLA = "outputfiles/powerCPUDLA.txt";
string powerFileCPUFP16 = "outputfiles/powerCPUFP16.txt";

string powerFileGPUFP32 = "outputfiles/powerGPUFP32.txt";
string powerFileGPUDLA = "outputfiles/powerGPUDLA.txt";
string powerFileGPUFP16 = "outputfiles/powerGPUFP16.txt";
```

The code above is a method to read the power from the power files stored on the SOC Information gotten from the Engine.hpp header file can be seen below

```
bool Engine::powerMonitor()
{

    string lineSOC, lineCPU, lineGPU;

    ifstream fileSOC(powerRailSOC);
    getline(fileSOC, lineSOC);
    powerListSOC.emplace_back(stoi(lineSOC));
    //printf("%d\n", powerListSOC[0]);
    fileSOC.close();

    ifstream fileCPU(powerRailCPU);
    getline(fileCPU, lineCPU);
    powerListCPU.emplace_back(stoi(lineCPU));
    //printf("%d\n", powerListSOC[0]);
    fileCPU.close();

    ifstream fileGPU(powerRailGPU);
    getline(fileGPU, lineGPU);
    powerListGPU.emplace_back(stoi(lineGPU));
    //printf("%d\n", powerListSOC[0]);
    fileGPU.close();
    return true;
}
```

This code opens the sensor files and adds their content them to a vector. We later calculate the average of them over 1000 iterations.

Appendix C

Loading from disk and getting meta data for inference

The code in this appendix are snippets from the code used in this thesis. This code shows how to get the meta data such as channel, height, width and batch size, used for running inference on the Resnet50 model. The full code can be found at: https://github.com/Joa2506/Resnet50.

C.1 Load phase

```
vector<char> buffer(size);
//Runtime object
unique_ptr<IRuntime> runtime{createInferRuntime(m_logger)};
//Set device
auto ret = cudaSetDevice(m_config.deviceIndex);
//Let's create the engine
m_engine = shared_ptr<nvinfer1::ICudaEngine>(
    runtime->deserializeCudaEngine(buffer.data(), buffer.size()));

//Getting input and output names plus dimensions
m_inputName = m_engine->getBindingName(0);
m_outputName = m_engine->getBindingName(1);

m_inputDims = m_engine->getBindingDimensions(0);
m_outputDims = m_engine->getBindingDimensions(1);
//Getting the meta data needed to process the image
m_batchSize = m_inputDims.d[0];
m_inputChannel = m_inputDims.d[1];
m_inputHeight = m_inputDims.d[2];
m_inputWidth = m_inputDims.d[3];

m_context = shared_ptr<nvinfer1::IExecutionContext>
    (m_engine->createExecutionContext());
```

Bibliography

[1] Bochkovskiy. A, Wang. C and Liao. H. H. 'YOLOv4: Optimal Speed and Accuracy of Object Detection'. In: (2020).

[2] *About Nvidia*. URL: https://www.nvidia.com/en-us/about-nvidia/.

[3] Bojarski. M et al. 'The NVIDIA PilotNet Experiments'. In: (2020).

[4] Martín Abadi. et al. 'TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems'. In: *Google Research* (2016).

[5] *Apple announces Mac transition to Apple silicon*. URL: https://www.apple.com/newsroom/2020/06/apple-announces-mac-transition-to-apple-silicon/.

[6] A. Çinar, M. Yildrim and Y Eroğlu. 'Classification of Pneumonia Cell Images Using Improved Resnet50 Model'. In: (2020).

[7] *Deep Learning Performance Documentation*. URL: https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html.

[8] *Diagram of Tegra Xavier architecture*. URL: https://en.wikichip.org/wiki/File:nvidia_xavier_die_shot_(annotated).png.

[9] Comer. D. E et al. 'Computing as a Discipline'. In: (1989).

[10] Peter Goldsborough. 'A Tour of TensorFlow'. In: (2016).

[11] *HARDWARE FOR SELF-DRIVING CARS*. URL: https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/.

[12] *INA3221 Triple-Channel, High-Side Measurement, Shunt and Bus Voltage Monitor with I 2C- and SMBUS-Compatible Interface*. Texas Instruments. Mar. 2016.

[13] *JETSON AGX ORIN*. URL: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/#:~:text=NVIDIA%5C%20Jetson%5C%20AGX%5C%20Orin%5C%20modules,other%5C%20autonomous%5C%20machine%5C%20use%5C%20cases..

[14] *Jetson AGX Xavier*. URL: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/.

[15] *JETSON AGX XAVIER AND THE NEW ERA OF AUTONOMOUS MACHINES*. URL: https://developer.download.nvidia.com/embedded/webinars/webinar-jetson-agx-xavier-new-era-autonomous-machines.pdf?AinKLpjfnqUWPD4GAPAmgjtQls9pTt3gtCXYc2Lctapvj7UbSDJCzuroZS9NEOyJtg8BQf1vuWW8zBcxpJ4cjJS4BuX0UNRVo_3DcB8tis4a_nsjTLxyHJOJcDnd3qaIG63GAp71_deLbVQZuzbohuuICfpoW8QkC9PZIvD.

[16] He. K, Zhang. X and Ren. S. 'Deep Residual Learning for Image Recognition'. In: (2015).

[17] *Keras*. URL: https://keras.io/.

[18] Ekman. M. *Learning Deep Learning*. Addison-Wesley, 2021.

[19] *Mixed precision training*. URL: https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html.

[20] *NVDLA*. URL: http://nvdla.org/.

[21] *NVIDIA CUDA C Programming Guide*. NVIDIA, 2012.

[22] *NVIDIA JETSON AGX ORIN DEVELOPER KIT*. NVIDIA Corporation. 2022.

[23] *NVIDIA Jetson AGX Xavier Delivers 32 TeraOps for New Era of AI in Robotics*. URL: https://developer.nvidia.com/blog/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/#:~:text=The%5C%20Jetson%5C%20AGX%5C%20Xavier%5C%20integrated,a%5C%20compute%5C%20capability%5C%20of%5C%20sm_72..

[24] *NVIDIA Jetson Xavier AGX System-on-Module*. NVIDIA Corporation.

[25] *Nvidia TensorRT*. URL: https://developer.nvidia.com/tensorrt.

[26] *Nvidia TensorRT developer guide*. URL: https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.

[27] *ONNX Webpage*. URL: https://onnx.ai/.

[28] *OpenCV*. URL: https://opencv.org/about/.

[29] Nahar. P, Tanwani. S and Chaudhari. N. S. 'FINGERPRINT CLASSIFICATION USING DEEP NEURAL NETWORK MODEL RESNET50'. In: (2018).

[30] *Power Management for Jetson Xavier NX and Jetson AGX Xavier Series Devices*. URL: https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3261/index.html#page/Tegra%5C%20Linux%5C%20Driver%5C%20Package%5C%20Development%5C%20Guide/power_management_jetson_xavier.html#wwpID0E0GN0HA.

[31] Afifi. Shereen, Hosseini. H. G and Sinha. R. 'A system on chip for melanoma detection using FPGA-based SVM classifier'. In: (2019).

[32] *SOFTWARE FOR SELF-DRIVING CARS*. URL: https://www.nvidia.com/en-us/self-driving-cars/drive-platform/software/.

[33] *TensorFlow*. URL: https://www.tensorflow.org/.

[34] *The MNIST database*. URL: http://yann.lecun.com/exdb/mnist/.

[35] *Transforming the Automotive Future: Innovations in Digitalization, Electrification, and Sustainability*. URL: https://resources.nvidia.com/ent-gtcs21/gtcs21-e32520.

[36] *Volta Tuning Guide*. URL: https://docs.nvidia.com/cuda/volta-tuning-guide/index.html.

[37] LeCun. Y, Bengio. Y and Hinton. G. 'Deep Learning'. In: (2015).

[38] Mukti. I. Z and Biswas. Dipayan. 'Transfer Learning Based Plant Diseases Detection Using ResNet50'. In: (2019).