CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Design and Evaluation of a Software Abstraction Layer for Heterogeneous Neural Network Accelerators

Master's thesis in Embedded Electronic System Design

Aishwarya Sreedhar
Naga Sarayu Nagarajan

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Design and Evaluation of a Software Abstraction Layer for Heterogeneous Neural Network Accelerators

Aishwarya Sreedhar
Naga Sarayu Nagarajan

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Design and Evaluation of a Software Abstraction Layer for Heterogeneous Neural Network Accelerators
Aishwarya Sreedhar & Naga Sarayu Nagarajan

Supervisor: Miquel Pericas, Department of Computer Science and Engineering, Chalmers University of Technology
Company advisor: Christofer Kanljung, Volvo Cars
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering, Chalmers University of Technology

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2022

Design and Evaluation of a Software Abstraction Layer for Heterogeneous Neural Network Accelerators

Aishwarya Sreedhar
Naga Sarayu Nagarajan
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Machine learning is becoming increasingly important across a wide range of hardware platforms. Current frameworks rely on vendor-specific operator libraries and cater to a small number of server-class GPUs. To be able to support a variety of hardware accelerators from various suppliers, which may vary over time, it is critical to abstract the hardware in order to deploy the core neural network algorithms across this heterogeneous hardware with minimal effort. There are various vendor specific consortiums and standards available in the market by the respective vendors. But to make the software portable, an abstraction layer should be build over the vendor proprietary standards. In this thesis, we have used a compiler that provides an abstraction level above CUDA and OpenCL so that we don't bother to know the details about CUDA/OpenCL programming, One such type of a compiler is Apache TVM, which is a open source machine learning compiler framework for CPUs, GPUs and other hardware accelerators. We have performed a comprehensive comparison between the model compiled using Apache TVM framework and native compilation for two different hardware vendors such as Nvidia and Qualcomm. Framework models are fed into deep learning compilers, which provide optimised code for a range of deep learning hardware. It exposes graph and operator-level optimisations to enable deep learning workloads with performance portability across a variety of hardware backends. TVM tackles deep learning-specific optimization problems like high-level operator fusion, mapping to arbitrary hardware primitives, and memory latency hiding. It also uses a evolutionary, learning-based cost modeling method for quick exploration of code to automate the optimisation of low-level programs to hardware features. Experiments show that TVM delivers performance comparable to state-of-the-art, hand-tuned libraries for low-power CPUs, mobile GPUs, and server-class GPUs across hardware back-ends. TVM's ability to target new accelerator back-ends, such as the GPU-based generic deep learning accelerator using CUDA and OpenCL is also demonstrated.

Keywords: Deep machine learning, Apache TVM, GPU , OpenCL, CUDA, thesis, self-driving cars, Nvidia Jetson, Qualcomm, performance, native programming.

Acknowledgements

Throughout our master thesis work, we have received continuous support and essential advice from both Chalmers University of Technology and Volvo Cars Corporation. We would like to sincerely thank everyone, especially:

- Miquel Pericas (Academic Supervisor)
- Christofer Kanljung (Company Advisor)
- Per Larsson-Edefors (Examiner)
- Pirah Noor Soomro (Thesis Reviewer)

Aishwarya Sreedhar, Naga Sarayu Nagarajan, Gothenburg, August 2022

Contents

List of Figures

List of Tables

1
Introduction

Smart applications that aid modern driver assistance systems and self-driving vehicles are increasingly being deployed to many devices, from cloud servers to edge devices. One of the primary technologies that enable self-driving cars is deep learning (DL). The DL models have become more efficient in getting better accuracy by handling large datasets, thus, making the computer systems capable.

The drawback of most of these frameworks is that they focus on a restricted number of server-class GPU devices and defer target-specific optimisations to vendor-specific operator libraries that are well designed. It requires more manual adjusting and is too optimised and complex to move quickly from device to device.

Any efforts that are taken to increase the performance of these techniques are valuable. In this case, DL accelerators are used, which are hardware architectures that are specifically designed to increase the performance of the hardware that is running the DL algorithm. Vendors like Intel [1] and Xilinx [2] already provide accelerators for various applications, including convolutional neural networks (CNN).

For an optimised deployment of current DL frameworks on different vendors' GPU targets, we can use parallel computing platforms and programming models such as CUDA [3] and OpenCL [4] to abstract the efforts required in writing GPU code into independent modules. Nvidia's CUDA and Khronos Group's OpenCL are widely used frameworks for writing applications that can run on GPUs. CUDA can be used only on Nvidia GPUs, but OpenCL can be used on many GPU device vendors.

The heterogeneity of the hardware characteristics in various hardware targets from different vendors differ in the architecture of Instruction Set Architecture (ISA), memory and compute unit organisations, which makes mapping DL workloads to the hardware target difficult. Nevertheless, it is essential for software developers to have an abstraction layer that enables simple deployment of the model on different hardware vendor targets without much alteration in the application itself and less focus on low-level details of the hardware target vendors. Furthermore, the challenge in creating an abstraction is finding a hardware-independent, general programming model that can handle different sets of functionality that should be performed by accelerators [5].

Accelerators from different vendors, such as Nvidia and Qualcomm, can use OpenCL for abstraction, as OpenCL supports both vendors. However, its generality may

result in a performance penalty. For example, CUDA performs better in data transferring to and from GPU on Nvidia devices when compared with OpenCL [6].

The ultimate goal of this thesis is to develop an accelerator abstraction layer for accessing the CNN application without being exposed to the low-level details of the accelerator technology and compare its performance with the native implementation of the model. The accelerator abstraction layer should be compatible with the popular programming environments and languages that developers are already familiar with [5].

Therefore, this thesis will initially analyse various software abstraction layers for CNN from the related current works. Then, we will find a suitable method to abstract the CNN model adaptable for the suggested hardware. Then, we will design a CNN model that can be abstracted and implement an optimal abstraction layer to compute the CNN rapidly and efficiently using the heterogeneous hardware accelerator. Furthermore, we will compare the efficiency of the abstraction layer with the direct deployment of the model with native frameworks of the respective hardware based on various performance metrics.

1.1 Background

Volvo cars is an automotive company that manufactures cars that also recently incorporated the autonomous driving feature. These autonomous cars have lidars and cameras to perform various autonomous driving functionality connected to two main computers in the car. The two main computers used in Volvo cars for autonomous driving are Autonomous Drive Primary Module (ADPM) and Vehicle Control Unit (VCU). Apart from these, many other microprocessors are present in the car to perform various applications.

The primary control units are VCU and ADPM, which run the applications for autonomous driving. One is the CNN application which translates pixels from the camera input to driving commands and is deployed in the VCU. The VCU uses Nvidia GPU accelerators to speed up the computing process of CNN. In the future, the company might change the CNN application to be deployed in ADPM, which uses Qualcomm GPU for redundancy. In this case, developing an algorithm for each hardware vendor is not optimal. Thus a single programming interface for diverse heterogeneous systems should be designed to overcome the cross-platform programming barrier.

In this thesis, the scope is limited to Nvidia and Qualcomm boards. The pre-trained YOLOV3 [7] and MobileNetV2 [8] model will be used as suitable CNN models for testing purposes. Thus, DL compilers will be a good choice for abstraction that can generate target-specific programs for CUDA and OpenCL. These suitable frameworks result in a good performance for the corresponding accelerators. Then, we will compare the performance of the CNN model using the DL compiler with direct implementation using suitable frameworks. Some of the suggested performance

metrics that are significant to analyse are throughput, adaptability, GPU usage, memory, execution time, and how well our model adapts to new hardware with ease and expressiveness, that is, how easy it is to express a wide range of CNN models with various hardware vendors.

1.2 Related Work

Deep learning frameworks [9] give users a simple way to express deep learning workloads and deploy them on various hardware backends. Rather than relying on vendor-specific tensor operator libraries to run their workloads like the existing frameworks.

In the present era of machine learning, compiler development for DNNs has received considerable attention. A few prominent frameworks designed to compile deep learning models into minimum deployable modules are Apache TVM [10], Intel's nGraph [11], Nvidia's TensorRT [12] , Google's XLA [13] , and Tensorflow Lite [14]. These frameworks take a computation graph from deep learning frameworks like PyTorch, Caffe2, and Tensorflow and generate highly optimised code for machine learning accelerators.

Deep learning compilers [9] have gained prominence in recent years as a flexible way to optimise and deploy deep learning models. Apache TVM [10], Glow [15] and XLA [13] are three DL compilers that provide optimisation and code creation for different hardware platforms.

A comprehensive survey by Chen Tianqi at al[16] shows that the TVM compiler is suitable for platforms like server-class GPUs, embedded CPUs and GPU, low power FPGA SoCs. Benchmarks were generated using real-world DL inference workloads such as ResNet and MobileNet. Unlike an external operator library, TVM provides end-to-end automatic optimisation and code generation compared to existing DL frameworks.

1.3 Thesis Overview

We begin by providing a brief overview of CNN and the abstraction layer for heterogeneous CNN accelerators, explaining the choice of DL compiler, and discussing relevant studies and our contribution to the present chapter. In the following Chapter 2, we will give a more in-depth CNN description, implementation of CNN on GPU, the introduction of Nvidia and Qualcomm GPUs, and a brief description of how TVM works. Chapter 3 will help the reader understand the tools and DL models designed using specific frameworks and describe the native implementation of CNN and the abstraction layer. Chapter 4 will show the results of both the methods described in the previous section and compare the two using performance

metrics. In the final Chapter 5, we will give the conclusions drawn from our work and suggest some future work.

2
Technical Background

In this chapter, we will explain the abstraction layer, the choice of DL compilers as an abstraction layer and why the Apache-TVM DL compiler is chosen. We will also explain the general structure of CNN, details of the hardware used, the general architecture of the DL and TVM compiler and the GPU architecture of Nvidia and Qualcomm.

2.1 Abstraction Layer

Currently, accelerator devices are employed in high-performance computing (HPC) systems, where this tendency is discernible from desktop computers to conventional supercomputers [17]. Therefore, the software executed on these targets should also be deployed efficiently for an optimised result. To develop solutions to be deployed in these heterogeneous systems, there can be two possible approaches: utilising a single programming model to handle the conceptual and architectural variations among the many computational devices and combining various programming models designed for each computing device [18]. However, the second approach is better than the first approach because, in the first approach, displaying non-complete regular programs with complex communications or synchronisations can be challenging. Also, the final code will not be as optimised as the original code. Furthermore, the second approach requires more in-depth knowledge about various programming models and various strategies used for memory access for different hardware targets for the programmer, as the programmer is responsible for manually handling the transfer of data from between various memory spaces at the opportune time based on the device's memory architecture and choosing the suitable kernel-launching configuration parameters. Though it is a tedious process for manual adjustment, it is more optimal.

In addition, several popular DL frameworks, such as TensorFlow and Darknet, are proposed so that developers can build and deploy DL models quickly. A unified model format, such as ONNX [19], can be used to provide interoperability for the model to be used across different frameworks. Therefore, to bridge productivity-focused deep learning frameworks and the performance-or efficiency-oriented hardware backends, the compiler technology makes the process sophisticated by generating optimised kernel codes based on the hardware target. The compiler will automatically help to manage some of the tasks, such as creating a program based on the programming model such as CUDA and OpenCL to utilise the accelerators'

computing capabilities, an optimisation system, memory management, and abstraction for indexed data structures. Compilers take DL models represented with DL frameworks as input and generate optimised code based on hardware targets as output [20].

Finally, when comparing multiple DL compilers such as TVM, nGraph, TC, Glow, and XLA, TVM is a better choice of compiler based on the reasons such as better performance on CPU for both full-fledged models and lightweight models and also better tuning performance on both CPU and GPU. It also enables sub-graph partitioning, quantisation, and unified optimisation, which means sharing advanced optimisations and supporting new hardware platforms, differentiable programming, privacy protection and training support [9].

2.2 Convolutional Neural Network

CNN is part of the artificial neural networks (ANN) family within DL, widely used in computer vision applications in the automobile and medical industries. It has three layers: convolution, pooling and fully connected. The convolution layer and pooling layer extract features, while the fully connected layer maps those features to the final output [21], as shown in figure 2.1.

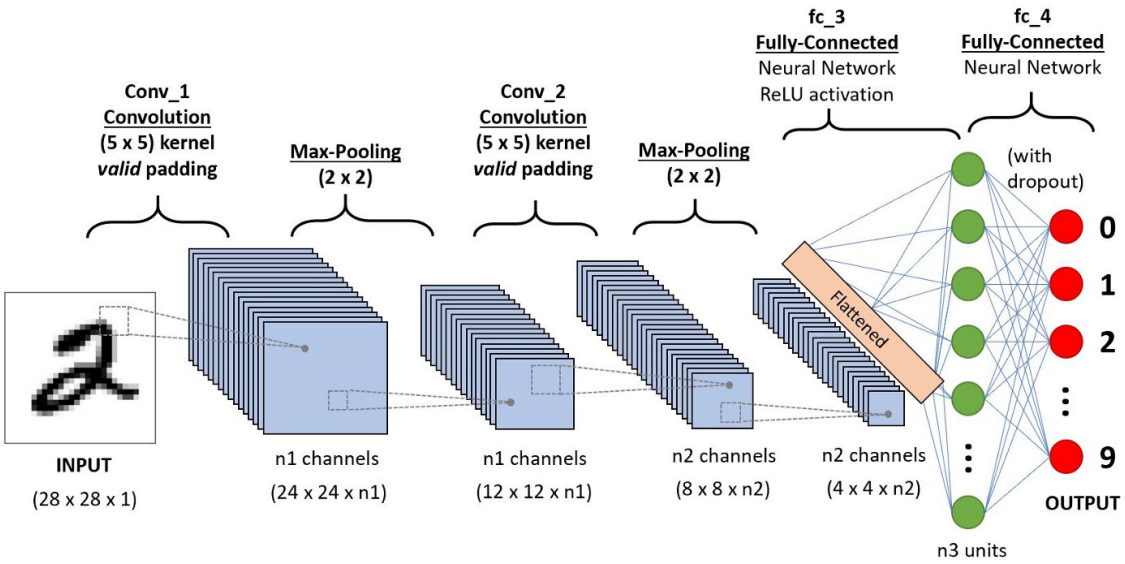

Figure 2.1: Convolutional Neural Networks [22]

The fundamental component of CNN is the convolutional layer. The convolutional layer is usually stacked with multiple layers that perform feature extraction, convolutional operation, and activation function, as shown in figure 2.2. The convolutional

operation is essentially an element-wise multiplication operation between two matrices. One of the matrices is a kernel, a filter or group of weights used to extract the image's feature and always smaller than the size of the input image. The other matrix is a portion of the input image. The input image has three RGB channels, and if it is black and white, it has only one channel, and according to the number of channels in the input image, the number of channels for the kernel also varies. The kernel is passed throughout the image, element-wise multiplication is performed, and the final output matrix is called a feature map.

The convolution method described above prevents the centre of each kernel from

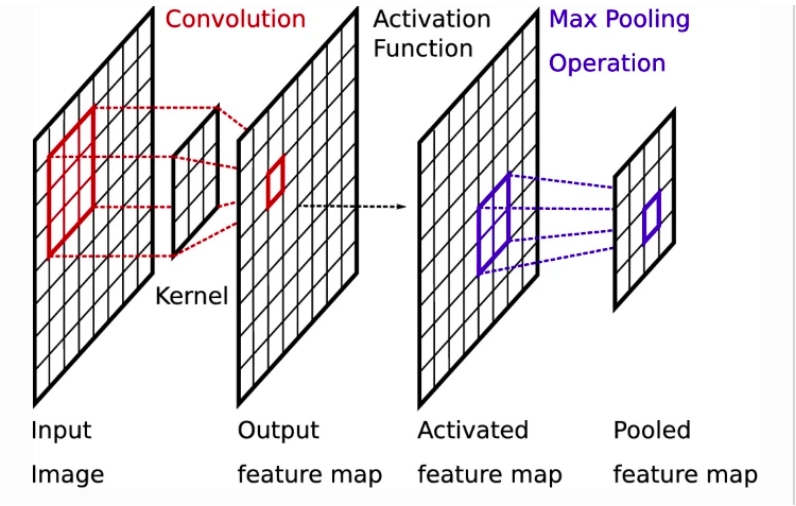

Figure 2.2: A basic CNN block depicting a single layer which applies a kernel on an input filter followed by an activation function and a max pooling operation [23]

overlapping the input of the outermost element of the tensor, reducing the output height and width of the feature map compared to the input tensor. To deal with this problem, zero padding is done as shown in figure 2.3. Zero padding adds additional rows and columns of zeros to all sides of the input tensor so the feature map would not get smaller after the operation. The stride is the distance between two successive kernel points. Usually, the stride is one, but it can also be bigger. The output of the convolution layer is sent to activation function which computes $f(x) = max(0, x)$.

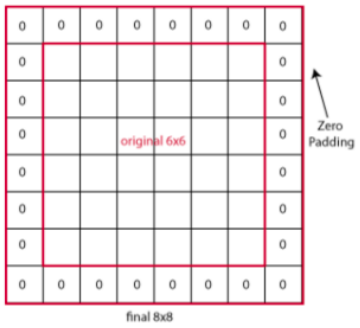

Figure 2.3: Zero Padding
[24]

The next layer is the pooling layer which downsamples the output from the previous layer by compiling a list of features seen in patches. The pooling layers do not have learnable parameters, but hyperparameters like filter size, stride, and padding are present, much like convolution operations. There are two types of pooling: max pooling and average pooling, which are used to downsample the feature map and then passed to the next layer. The final layer is the fully connected layer which uses the output of the pooling layer and converts it into a one-dimensional (1D) numeric array (or vector)as shown in figure 2.4. There are multiple fully connected layers where every input is connected to every output by a learnable weight [25] [26].

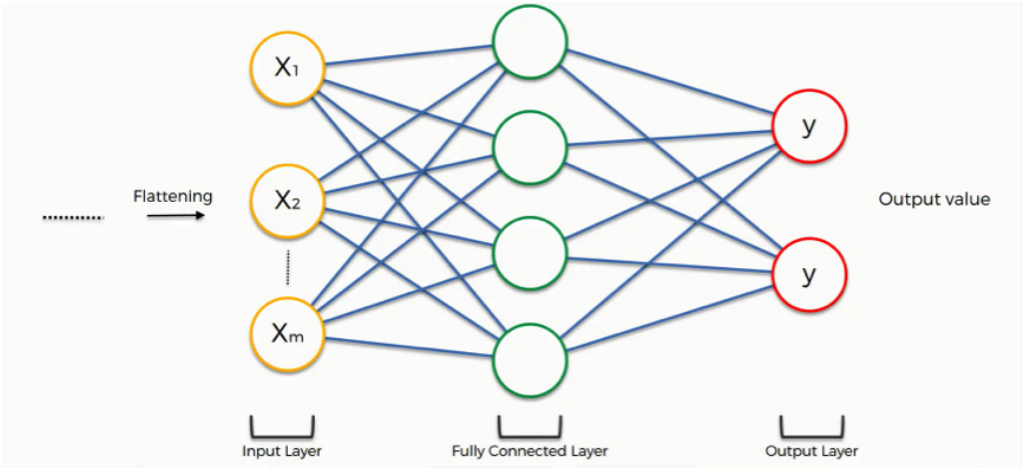

Figure 2.4: A fully connected layer
[27]

2.3 DL Compiler

Various DL programming frameworks, such as PyTorch, MXNet, TensorFlow and Keras, make the application of DL models easier. Interoperability between the frameworks is critical for enabling emerging DL models across existing DL models depending on the tradeoffs in their designs. Therefore, an open-source standard

was proposed - the "ONNX model" to improve interoperability. This model helps convert all the DL models into different DL frameworks easily, as shown in figure 2.5.

Moreover, highly efficient DL chips customised to perform matrix multiplication and improve the computing capability of the DL models were invented, such as Google TPU and Hisilicon NPU, and some of the processor vendors are NVIDIA Turing and Intel NNP. It is critical to efficiently map computation to DL chips to speed up the DL models on various DL devices. Hence, various libraries were released specifically tailored for DL operations, like Basic Linear Algebra Subprograms(BLAS), to increase DL operations' efficiency and faster computation.

The disadvantage of depending on libraries and tools for mapping DL models to various DL chips is that they frequently fall behind the rapid growth of DL models, causing them to be underutilised. DL compilers were utilised to address this limitation. These compilers can take the model specifications provided in DL frameworks as inputs and deliver an efficient, optimised code for the specified DL hardware. Furthermore, existing DL compilers use mature toolchains from general-purpose compilers (e.g., LLVM), resulting in improved portability across a wide range of hardware architectures. The DL compilers use a layered design with a frontend, multi-level intermediate representation (IR), and backend, focusing on optimisation. In this thesis, we have used Apache TVM DL compiler for the following reasons [9] such as supporting most of the common DL frameworks and DL hardware chips, OpenGL to generate codes, the execution of the model in standalone mode using ahead-of-time compilation (AOT) and also quantisation.

Figure 2.5: DL framework view: 1) Currently popular DL frameworks; 2) Historical DL frameworks; 3) ONNX supported frameworks. [9]

Thus, as mentioned in the above section, the Apache TVM compiler is used in this thesis for the reasons explained above. It is an open-source machine learning compiler that distils the largest, most powerful deep learning models into lightweight software that can run on the edge. This allows the acquired model to run inference much faster on various target hardware (CPUs, GPUs, FPGAs accelerators) and save high costs. The architecture of the TVM compiler is explained in more detail in the next section.

2.3.1 Design Architecture of TVM

This section presents the overall architecture of the Apache TVM. It consists of two parts: the frontend and backend of the compiler, as shown in figure 2.6.

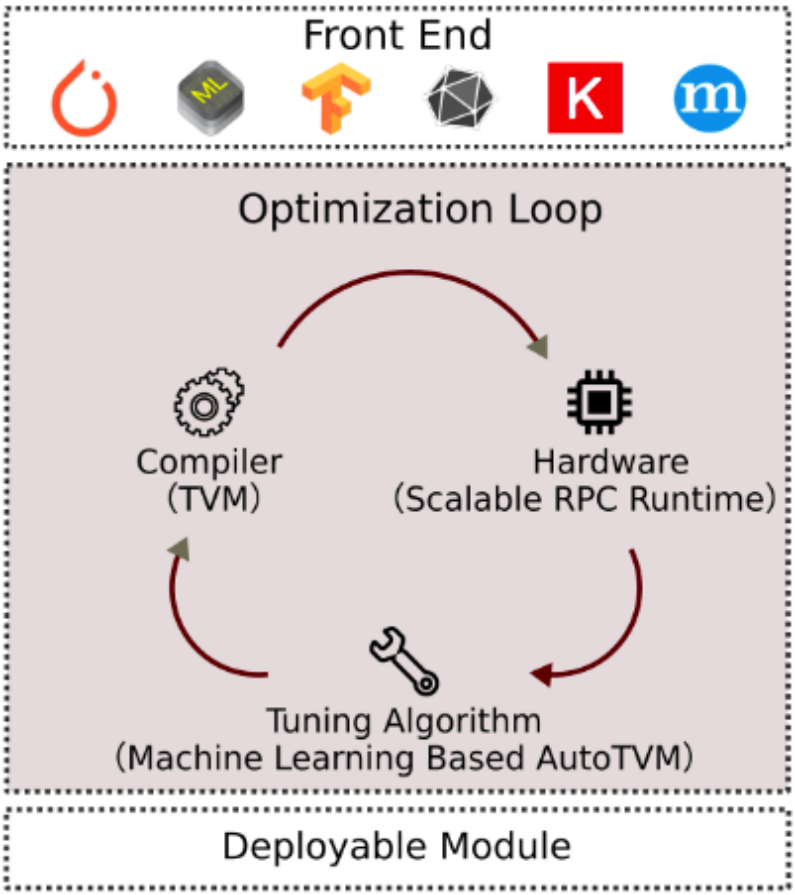

Figure 2.6: TVM Model
[28]

The frontend and backend use the intermediate representation (IR), an abstraction of the program used across the entire stack that consists of a collection of functions used to optimise the program. The next step is the transformation which is carried out for two reasons; firstly, for optimisation, where the same program is transformed to a more optimised version and secondly, for lowering, that is, the program is transformed to a lower level and closer to the target.

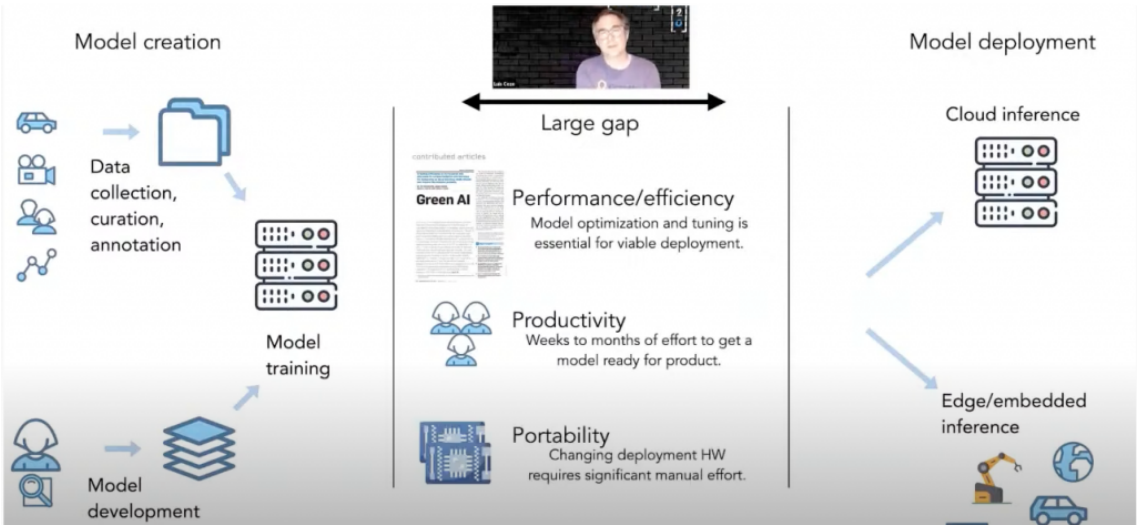

Figure 2.7: TVM Model creation and deployment [29]

The figure 2.7 gives an idea about how the DL model is deployed using TVM. First, the model is ingested through the frontend of the compiler to the IR, which has a collection of functions representing the model concerning the internal representation.Numerous changes are made from one IR module to another IR module that is functionally comparable. This is done in the case of quantising the model for a more compact model representation, but this transformation is done independently of the hardware. The next step is transformation based on the hardware specified where the IR module translates (Codegen) to another IR module. Finally, the outcome is packaged as a runtime module that can be exported, loaded, and executed on the target runtime environment.

2.3.2 Overview of how TVM works

TVM has been built to achieve state-of-the-art results with hardware-specific optimisations on embedded CPU, GPU, embedded GPU and FPGA hardware. TVM supports numerous hardware optimisations; initial graph level optimisations like pruning and fusion are nearly identical for all hardware, but low-level hardware-specific optimisation varies depending on different hardware.

TVM offers many levels of optimisation, as shown figure 2.8. When the model is imported, the graph undergoes first-level optimisation, which includes graph level fusion, layout transformation and memory management. This optimisation happens at the tensor level at the code generation layer. The TVM stack has many layers; firstly, the user interfaces layer. This layer is written in python and supports input modules of various frameworks. In this layer, DL models designed with specific frameworks such as TensorFlow, Caffe, and onnx are converted to TVM-compatible graphs.

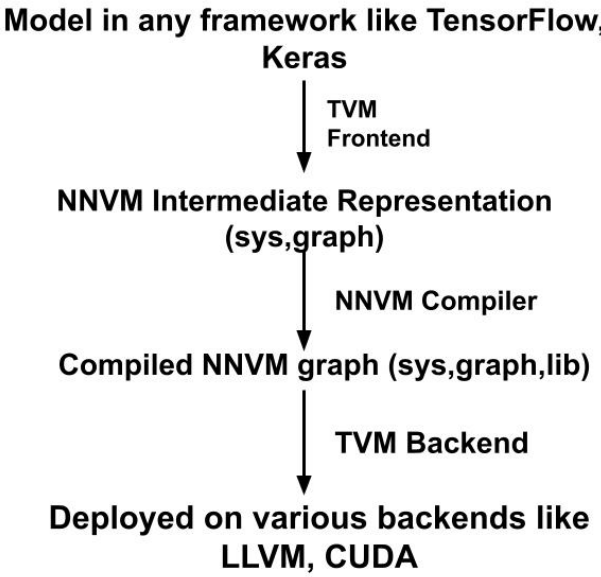

Figure 2.8: TVM workflow

The second layer is the computation graph optimisation layer. The graph representation is optimised by various passes like a pre-compute prune, which prunes the graph nodes that can be computed at compilation time. The layout conversion pass adds the necessary layout conversion operations (or nodes) across layers if there is a layout mismatch between layers. Then, a fusion pass is added to convert the computation of multiple nodes into one based on specific rules. For rapid code optimisations, a breakthrough learning-based cost model method automates the optimisation of low-level programs to hardware characteristics.

The last layer in the stack is the schedule space and optimisations for low-level and hardware-specific optimisations. Here, operators are built using a tensor expression language and provide primitive changes in the program that generate various optimised programs. We will discuss further the tools, code and implementation of the CNN model using TVM in the subsections in detail [29].

2.4 GPU Introduction

A graphics processing unit (GPU) is one of the vital computing units designed to perform parallel processing and is used in various applications such as graphics and video rendering. It has many cores, and each core runs at a much slower clock speed than a CPU. GPUs are designed to speed up the execution throughput of massively parallel programs. For example, the Nvidia GeForce GTX 280 GPU has 240 cores,

each of which is a heavily multi-threaded, in-order, single-instruction issue processor (SIMD single instruction, multiple-data) that shares its control and instruction cache with seven other cores. So far, GPUs have been the best in handling floating point operations per second.

GPUs are designed for data-intensive applications, created for designing 3D rendering and necessitate storing a significant amount of texture and polygon data. The caches cannot handle such vast amounts of data; the only way to improve rendering performance was to raise the bus width and memory clock. The Intel i7, for example, has a memory bus width of 192b and a memory speed of 800MHz, making it the fastest processor currently available. The GTX 285 had a memory clock of 1242 MHz and a bus width of 512b. The GPU has hundreds of cores compared to the 4 or 8 cores in the latest CPUs, as shown in figure 2.9.

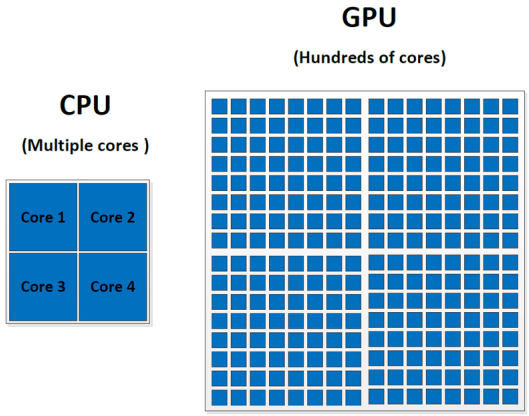

Figure 2.9: CPU-vs-GPU-architecture [30]

2.4.1 GPU Architecture

A CPU handles complex tasks such as time slicing, virtual machine emulation, complex control flows, branching, and security. They are in charge of billions of low-level, repetitive tasks. They were initially meant to produce triangles in 3D graphics and feature hundreds of arithmetic logic units(ALUs) compared to ordinary CPUs, which typically have only 4 or 8. Many scientific algorithms spend most of their time doing what GPUs are designed to do, that is billions of repeated arithmetic operations. The graph in figure 2.10 depicts the improvement in the performance of GPU over time when compared to classic CPU architectures.

A GPU program comprises a host component that runs on the CPU and one or more GPU kernels. The CPU component of the program is typically used to set up the parameters and data for the computation, whereas the kernel portion is responsible for the actual computation. In some circumstances, the CPU part will include a parallel application that uses a message passing interface (MPI) to accomplish

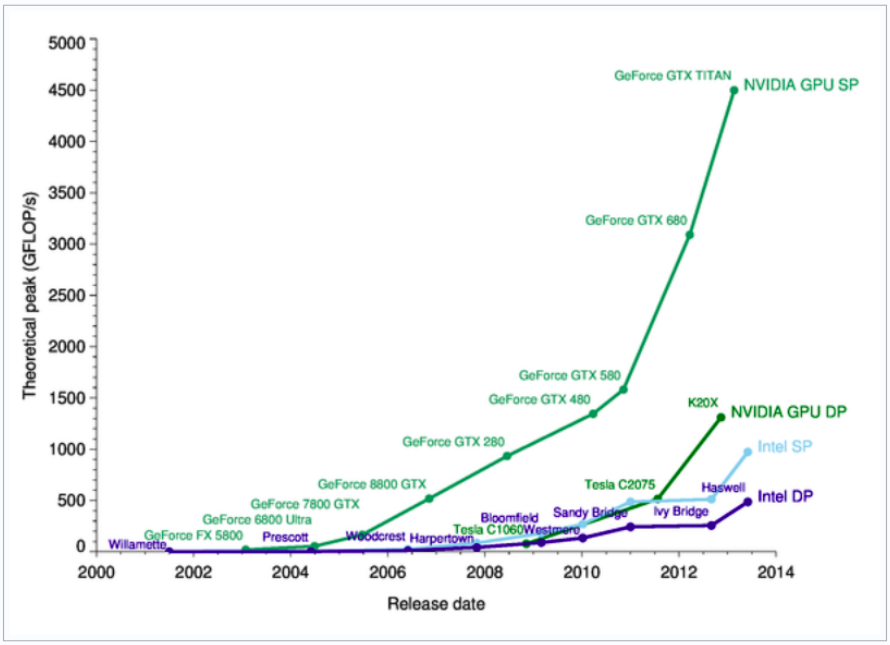

Figure 2.10: Theoretical GFLOP/s at base clock [31]

message passing activities [32].

2.4.2 GPU Implementation of CNN

In high-performance computing (HPC) and real-time systems, the utilisation of GPU platforms is becoming more common. Previously, GPUs were only used in the gaming sector. However, because of their benefits, GPU increasingly being employed by general platforms to meet the needs of many types of users. As a consequence of all this, the general-purpose GPU (GPGPU) evolved. These devices show enormous raw computational capability when compared to traditional CPUs. The architecture of these cores differs based on the manufacturer. However, each core contains pipelined ALU, which implements the essential arithmetic functions. Therefore, these cores have a smaller instruction set when compared to CPUs. For example, the GeForce GTX 560 GPU has 336 CUDA cores, for which 1.95 billion transistors have been used [33].

The implementation of the DL model necessitates a significant amount of computing power. Specifically, the training phase of the DL model requires more resources as it takes a large number of inputs processed with adjustable weights during training and gives a prediction. These operations usually involve matrix multiplications. Thus, an NN will use thousands of parameters to handle this operation, as shown in figure 2.11. It is achievable using CPUs. However, it would consume more time. Therefore, to make this process faster, GPU is a better solution as they parallelly execute the tasks rather than executing one-by-one. One of the features of GPUs is dedicated memory that performs the floating point computations required for

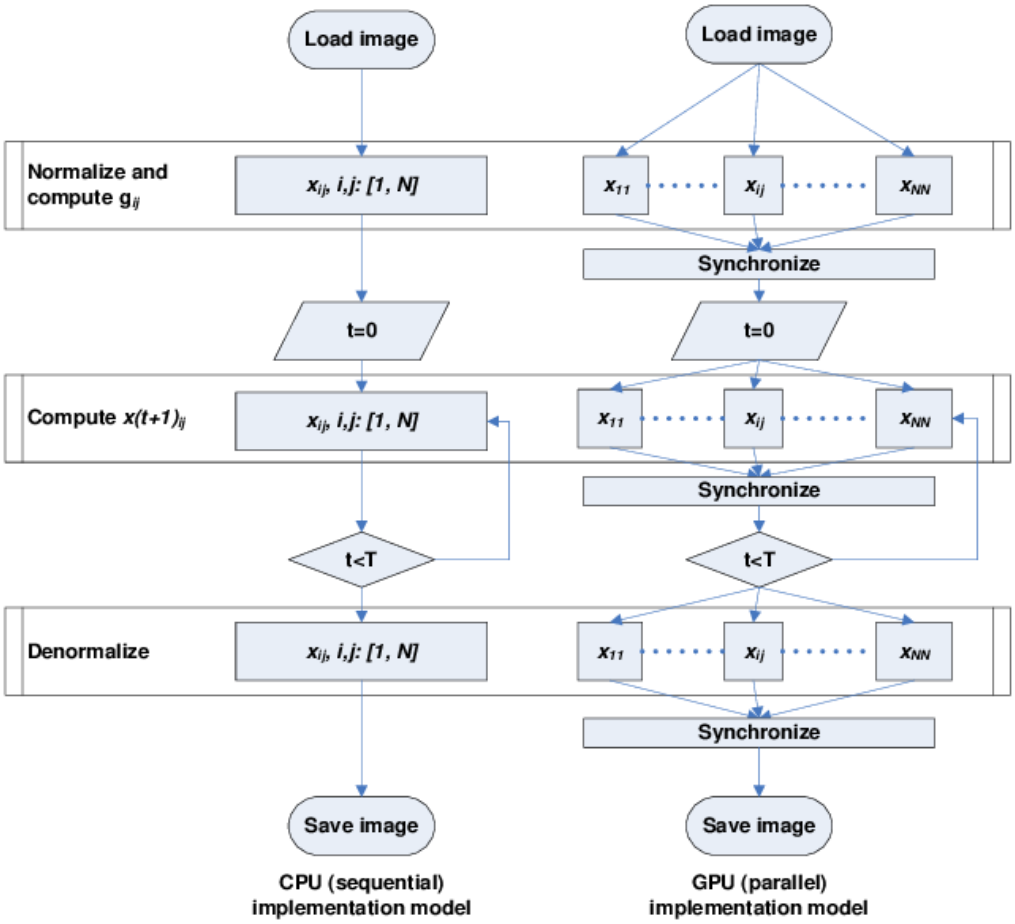

Figure 2.11: CPU sequential, GPU parallel implementation model of CNN [34]

graphics rendering [35].

2.5 Nvidia GPU

In the late 1990s, Nvidia released the first GPU, the GeForce 256. GPUs have become cheaper, more powerful, and smaller since then and have become one of the significant components of deep learning. The GPU has multiple cores with high throughput. Many of the major deep learning frameworks are compatible with this platform, and Nvidia offers tools to use them, such as cuDNN and TensorRT [36].

2.5.1 Compute Unified Architecture(CUDA)

CUDA is a programming model for general-purpose computing on Nvidia GPUs, which allows non-sequential tasks to be executed in parallel with other processes on the GPU [33]. It is a C/C++ programming extension. CUDA is a parallel computing platform and an API model developed by Nvidia. CUDA combines hardware and software technology that enables programmers to write well-optimised code for a GPU architecture typical to all CUDA graphics cards. The CUDA architecture is designed for the immense use of parallel computing applications and those that use a lot of computation power. It assists programmers in writing programs in various languages and providing keywords for extensions for languages such as C, C++, Python, MATLAB and Fortran.

The architecture of CUDA consists of various components such as parallel compute engines, OS-kernel level support for hardware initialisation and configuration, device-level API for developers and PTX instruction set architecture (ISA) for parallel computing kernels and functions.

2.5.2 CUDA - GPU design

The figure 2.12 represents the architecture of a CUDA-capable GPU, which consists of 16 streaming multiprocessors (SMs), each having eight streaming processors (SPs), making a total of 128 SPs. Each SP has a multiply and addition unit and an additional multiply unit. The GT200 has 240 SPs and exceeds 1 Tera floating point operations per second (TFLOP) of processing power. Each SP is massively threaded, and each application can run thousands of threads. For example, the G80 card supports 768 threads. Because each SM has eight SPs, each SP can support 96 threads. The total number of threads that can be run is 128 * 96 = 12,228. This is why CUDA-capable GPU are referred to as "massively parallel" processors [32].
The CUDA parallel programming model has three critical abstractions at its core: a hierarchy of thread groups shared memories and barrier synchronisation. The CUDA abstractions provide fine-grained data parallelism and thread parallelism, which are nested within coarse-grained data parallelism and task parallelism. They show the programmer how to divide the problem into coarse sub-problems that can be addressed independently in parallel by blocks of threads and finer sub-problems that can be tackled cooperatively in parallel by all threads within the block.

A kernel is executed in parallel by an array of threads where all threads run the

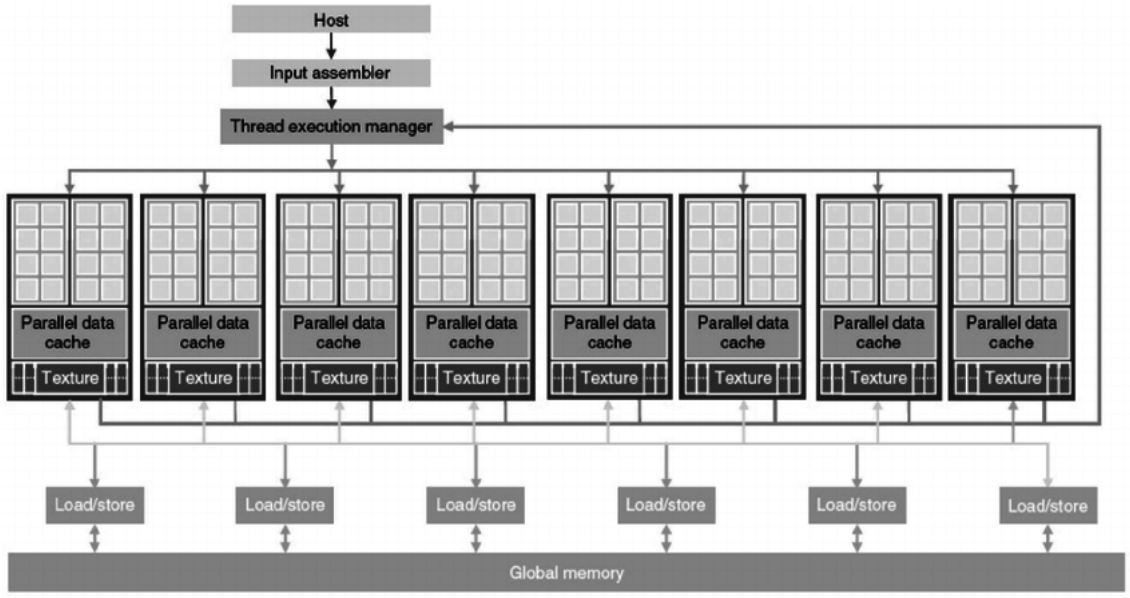

Figure 2.12: Architecture of a CUDA capable GPU
[32]

same code. Each thread uses an ID to compute memory addresses and make control decisions. These threads are arranged as a grid of thread blocks where different kernels can have different grid/block configurations, as shown in figure 2.13, and threads from the same block have access to shared memory, and their execution can be synchronised. It also provides an 8GB/s communication channel with the CPU (4GB/s for uploading to the CPU RAM and 4GB/s for downloading from the CPU RAM).

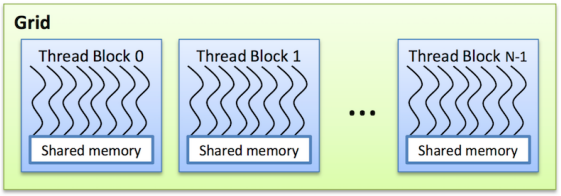

Figure 2.13: Grids and Threads
[32]

The grid of blocks and the thread blocks can be 1, 2, or 3-dimensional as shown in figure 2.14.
One of the main advantages of CUDA over traditional GPGPUs with graphics API is that they have Integrated memory (CUDA 6.0 or later) and Integrated virtual memory (CUDA 4.0 or later). They provide full support for bitwise and integer operations. Moreover, CUDA has shared memory which provides more bandwidth and improves performance [37].

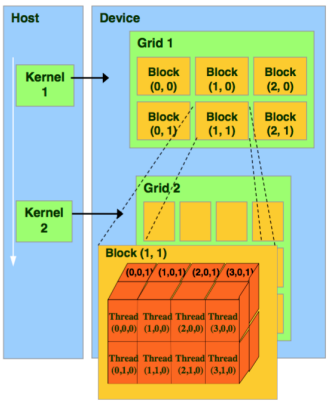

Figure 2.14: Grids and Threads with different dimensions [32]

2.6 Qualcomm GPU

Adreno is a sequence of GPU semiconductor intellectual property cores developed by Qualcomm. These GPUs are used in many of their System-on-Chip (SoCs) [38]. These GPUs accelerate the rendering of complicated geometries, resulting in high-performance graphics and a rich user experience while consuming little power [39].

To speed up the rendering of complicated geometry, the design of the Snapdragon processor incorporates the AdrenoTM GPU. The Adreno GPU is a component of the Snapdragon that produces excellent graphics with smooth animation (up to 144 frames per second) and HDR (High Dynamic Range), which supports over one billion different shades of colour. The Snapdragon platform is depicted in the simple block diagram in figure 2.15.

The GMEM in the block diagram above represents the local memory of the GPU. It is used for fast colour and stencil rendering. The GPU effectively burst writes all the blended pixels from GMEM to the frame buffer in system memory in a single layer. In the Snapdragon, the Adreno is designed to improve GPGPU performance to distribute the workload of the CPU cores to cover a wide range of use cases such as computational rendering, computer vision, image processing, and machine learning.

The Adreno GPU utilises the unified shader architecture, which employs several shader processors and is not tailored for either pixel or vertex processing, but rather is capable of handling both instruction types and completing both tasks at once. The shaders improve the efficiency of vertex and pixel shaders and offer a high degree of latency for multithreading and FIFOs. The advanced architecture ensures that nonvisible or concealed pixels are quickly rejected before texture fetch and processing for better efficiency and performance.

All prominent graphics API standards, such as OpenGL ES 3.2 (Android O or later), OpenCL 2.0 FP-compliant, Google RenderScript, EGL 1.4 plus extensions, C2D 3.0,

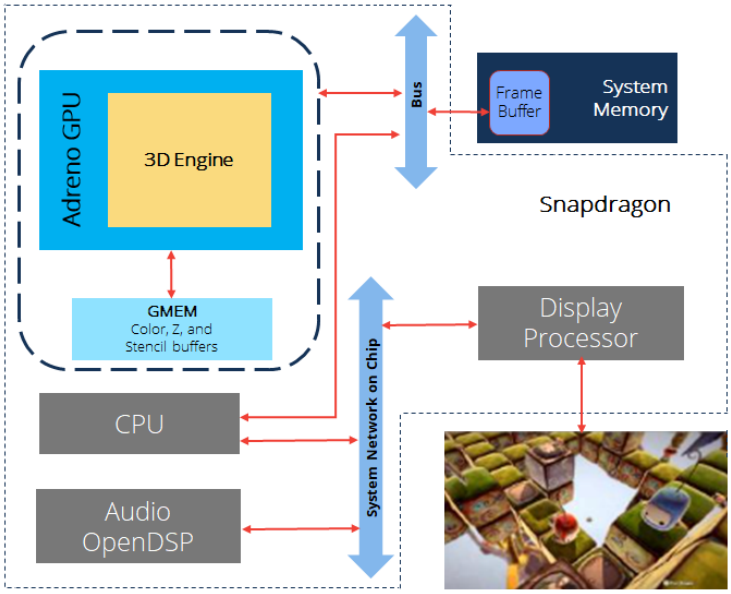

Figure 2.15: A simple block diagram of Snapdragon platform [40]

and Vulkan 1.0, are supported by the Adreno GPU (Android O or later) [40].

The Qualcomm Adreno 640 chip was introduced in early 2019 and was mainly used for high-end Android devices. This GPU is a primary GPU for smartphones and tablets, integrated within the Qualcomm Snapdragon 855 SoC. This GPU supports Vulkan 1.1 API [41].

A 10 nm technology was used to create the GPU. The Adreno 640 graphics processor features 384 ALUs running at 750 MHz as shown in the figure 2.16 and is used in games like Minecraft, Asphalt 9, Madout2, GTA, PUBG, Call of Duty, and others.

In this thesis, we are using an Adreno GPU high-level diagram from the perspective of OpenCL programming.

2.6.1 OpenCL

OpenCL is another programming model that supports various accelerators found in various embedded devices. OpenCL has a controller-device execution model, host process running in the CPU, and task creation and managing is done in GPU. In this case, the host creates the target device's kernel code and submits the task for its execution to the command queue hence the command queue schedules the task on the GPU, sometimes asynchronously. It consists of four components: a work item, a separate thread that executes the kernel software; work group, a group of threads that work together in lockstep; local size, dimension of each workgroup which can be up to three-dimensional arrangements; global size, The overall number of threads that must be executed. The memory hierarchy in OpenCL has four types, private, constant, local and global. OpenCL takes complete control of the hardware and is

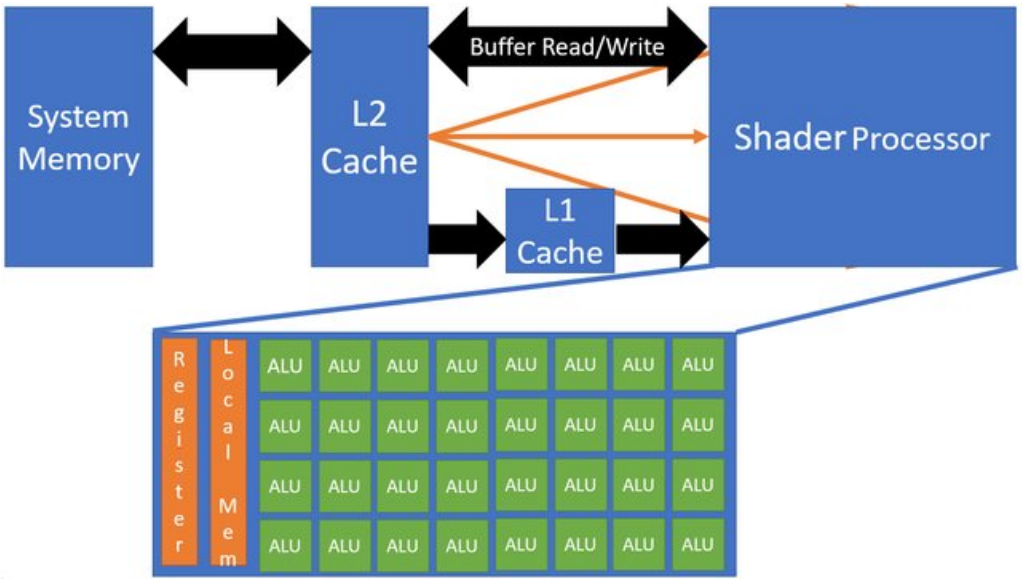

Figure 2.16: Adreno GPU architecture for OpenCL [42]

in charge of the parallelization process [42]. The basis of the OpenCL programming model is the idea of a host device backed by an application API and other devices connected through a bus, where the host API consists of platform and runtime layers. Thus, the Qualcomm hardware uses TensorFlow, an open-source mobile deep learning framework, and its OpenCL backend to compute kernels on GPUs.

3
Methods

In this chapter, we will discuss the design of the DL model using specific DL frameworks, its necessary tool-chain, and the specific library functions used to implement the object detection CNN model and discuss the implementation of the abstraction layer for heterogeneous NN accelerator using Nvidia Jetson Nano and Qualcomm board respectively.

3.1 Need for abstraction layer

The limitation in this thesis is designing a DL programming model compatible with both Nvidia and Qualcomm hardware accelerators. The abstraction layer we design must be able to work on both this platforms irrespective of their internal hardware designs and specifications. It might be challenging to program a shared and optimal neural network with a commonality or intersection between these two hardwares. Since both the hardware platforms support a specific software development kit (SDK), it might be cumbersome to study and test these SDKs with different programs. HW accelerators could have different limitations in the NN layers they support, such as integer and or float operation. Therefore, the abstraction layer should support the intersection of available functionality or the union, where the TVM compiler comes into the picture. TVM intends to close the gap between developing machine learning models and their implementation in production. It streamlines the time-consuming process of adapting the models to different types of backend hardware, such as CPUs, GPUs, and specialised accelerators [43].

3.2 Nvidia Jetson Nano

In this section, we will discuss the features of the Jetson Nano development kit, implementing CNN with native and TVM compilation on Jetson Nano.

3.2.1 Nvidia Jetson Nano Module

We have used the Nvidia Jetson Nano development kit as one of the vendors of GPU-based accelerator and implemented the abstraction layer as shown in figure 3.1.

Figure 3.1: Nvidia Jetson-Nano
[44]

The Nvidia Jetson Nano module is compact of size (69mm x 45mm) and a powerful computer that features a 64-bit quad-core Arm Cortex-A57 CPU running at 1.43GHz, as well as an Nvidia Maxwell GPU with 128 CUDA cores capable of 472 GFLOPs (FP16), 4GB of 64-bit LPDDR4 RAM, and 16GB of eMMC storage. It also runs Linux for Tegra. The 260-pin SODIMM connector on the 70 x 45 mm module separates video, audio, USB, and networking interfaces and can be connected to a compatible carrier board. The above features let us run multiple neural networks in parallel for applications like image classification, object detection, segmentation, and speech processing which delivers 472 GFLOPS of computing performance for running modern AI workloads and is highly power-efficient, consuming as little as 5 watts [45] [46] [47].

The Nvidia Jetson Nano developer kit has a MicroSD card slot for storage. The connections are via Gigabit Ethernet and M.2 Key E and a 2.0 Micro-B port to connect the power source. The OS used is Ubuntu 18.04. The Jetson device is configured with 2 GB reserved swap memory and 4 GB total RAM. The board features peripherals such as USB ports, HDMI and DisplayPort, pin connectors, and an Ethernet connector. The board can be powered using either a 5V/2.5A micro USB or a 5V/4A barrel connector [48].

The Jetson Nano can be used as a standalone computer as it runs on Ubuntu. As a result, it can take advantage of the frameworks, tools, and libraries available for Ubuntu to improve efficiency during development. The Jetson Nano has power options that need to be set to maximum to obtain the performance. The commands sudo nvpmodel -m 0, sets the Jetson Nano to maximum power usage, and jetson clocks, which disables dynamic frequency and voltage scaling (DFVS), can be used to do this [49].

3.3 CNN Model

Firstly, we implemented CNN using a pre-trained Yolov3 (You Only Look Once) object detection model because it is fast and accurate. All Yolo models work by performing regression, i.e. they work by predicting the bounding boxes and class probabilities for each using a single network pass.

The Yolov3 model is a real-time object detection algorithm that identifies specific objects in videos, live feeds, or images. It is an improved version of Yolo and Yolov2. There are significant differences between Yolov3 and previous versions regarding class speed, precision, and specificity. In Yolov3 models, the bounding boxes get predicted at different scales [7].

Along with the Yolov3 model, we have used the Darknet framework to train neural networks. It is open source written in C or CUDA and serves as the basis for Yolo. In addition, it acts as a backbone for the Yolov3 object detection approach.

3.3.1 Implementing CNN with Native Compilation

We implemented CNN with native compilation using the Yolov3 model as the inference. We used Pjreddie's Yolo [50] model and compiled the Darknet using 'make' along with OpenCV for building a deep learning model.

The first step we took was to install all of the package dependencies for the Yolov3 model on Ubuntu. The prerequisites for GPU acceleration using an Nvidia GPU with Cuda cores, including the installation of libraries and versions of Cuda, have been met. When we download the Darknet, a makefile is also installed. In order to run this model on GPU, a few parameters must be modified in the makefile; therefore, we enabled CUDNN and GPU to accelerate the training process threefold over actual training. Furthermore, we have enabled OpenCV in the makefile to accelerate the DNN module's implementation. After all the changes have been made, the makefile is executed to implement the changes. The model is then tested with a single image to detect the object. The pre-trained weights are trained using the COCO dataset and then used in the model. After multiple trials, we set the threshold value as 0.25 as it gives reasonable accuracy. After the model is executed, an output image with bounding boxes and accuracy will be predicted. The time taken to compile the model will be measured. This is one of the critical parameters used for comparing the performance of the native model with the TVM compiler.

3.3.2 Implementing CNN with TVM Compilation

This section will explain the TVM compiler and how it is implemented to build a unified, programmable software stack. We will also explain the installation of TVM and the implementation of the Yolov3 model using the TVM compiler.

3.3.2.1 Installation of TVM compiler

There are multiple ways to install the Apache TVM on a Ubuntu OS, one of which is installing from a source. To begin with, we installed the TVM package from the official website. Firstly, a git folder of the TVM package is cloned, and multiple files and libraries are downloaded to a folder in the system.

The downloaded TVM package contains a makefile, which can be configured and executed. For example, if we want to enable or disable the CUDA backend, we can set $set(USE\_CUDA\ ON)$ or $set(USE\_CUDA\ OFF)$. Likewise, if we want the enable the graph executor and debugging functions, we set them as $set(USE\_GRAPH\_EXECUTOR\ ON)$ and $set(USE\_PROFILER\ ON)$ respectively. Similarly, we can also enable the 'debug with relay IR' parameter. Then, we install the LLVM package for CPU codegen, and suitable changes are made in the TVM makefile file. Finally, the makefile is executed, where all the necessary parameters are set to modify the TVM as required.

The next step is to setup a Conda environment as it is a great tool for obtaining the dependencies that are needed to run TVM. Since, Jetson Nano is 64bit arm cores and does not support Anaconda, we have installed Archiconda, which is a distribution of Conda for 64 bit ARM and then, a Conda environment is built for TVM. Finally, Python and C++ packages and their dependencies are installed.

3.3.2.2 Implementation of CNN using TVM compiler

We started with compiling and optimizing a basic CNN model like Resnet50 on TVM and then tuning it by making use of the auto-tuner which is in-built into the TVM. The Resnet50 is a model with 50 layers that are trained with multiple numbers of images in various classifications. The image size given as input for this model is 224×224.

Firstly, we used TVMC which is a command line driver of TVM that reveals TVM features like auto-tuning compiling, profiling, and execution of models available through the command line interface. The Jetson Nano can be used as a standalone computer as it runs on Ubuntu. As a result, it can take advantage of the frameworks, tools, and libraries available for Ubuntu to improve efficiency during development. The Jetson Nano has power options that need to be set to maximum to obtain full performance. The commands `sudo nvpmodel -m 0`, sets the Jetson Nano to maximum power usage, and Jetson clocks, which disables dynamic frequency and voltage scaling (DFVS), can be used to do this and line interface. It is a Python application that is installed with the TVM python package. The TVM supports a model in various formats such as Keras, ONNX, Tensorflow, etc. For testing purposes, we downloaded the Resnet50 in ONNX format from the web. Here, we are using CPU for compilation and we give LLVM as the target. In case, we need GPU to compile, we must give CUDA as the target. We then compile the model with the `tvmc compile` command. After compilation, a Tape Archive files (TAR) file consisting of three different files is generated. First, a mod.so file is generated. This file is the

model represented as a c++ library that can be directly used in the TVM runtime. Second, a `mod.json` file is generated which contains the computational graph represented in the text format, and then, a `mod.params` file is produced that contains the parameters of the pre-trained model. This compiled model is then loaded in the TVM runtime which is in-built into TVMC along with the input image. The image is resized as required by the model and the ONNX format expects it to be in NCHW (batch N, channels C, height H, width W) format. Therefore, it is converted to NCHW format, and the image is normalised according to Imagenet. Finally, we run the model in TVM runtime using `tvmc run` and `predictions.npz` file is produced as an output. Then, the predicted file is converted to a more human-readable form to produce a proper output.

The next step is to optimise the model to run faster. Therefore, the model is tuned according to the target. The tuning process is not similar to training or fine-tuning because it does not make any changes in the accuracy of the model but rather just improves the runtime performance. XGBoost package is also used for this purpose to get an optimised distributed gradient boosting. This tuned output is again compiled and run. Lastly, the performance of the tuned and untuned models are compared.

As a further step, we used the Yolov3 object detection model with Darknet deep learning framework using OpenCV and CFFI libraries and compiled using TVM. The same weights cfg files and Darknet C++ libraries are used from the previous task. When the frontend is executed, the graphical representation will be generated. This graph is then converted to relay using the functions of TVM. During compilation, the target has to be set to GPU, therefore, it is set to CUDA. After compilation, the input image is specified and the model is executed.

In this process, we did not tune the model due to memory restrictions in Jetson Nano, thus we did not obtain an optimized model and the compilation time is greater than expected. However, according to studies, the obtained compilation in GPU will be 30 times better than the native compilation if the model is tuned.

3.4 Qualcomm Board

In this section, we will discuss the features of the Qualcomm board, implementing CNN with native and TVM compilation on the Qualcomm board.

3.4.1 CNN Model

MobileNetV2 [8] is a mobile-optimized lightweight convolutional neural network design which is a good choice of CNN model to be implemented for the android platform that improves the performance state.

CNNs, such as MobileNet, are designed to be used on embedded and mobile vision applications. They are built using depthwise separable convolutions, which are lightweight DNNs that can have low latency for embedded and mobile devices.

In order to successfully maximise accuracy while taking into account the constrained resources for an on-device or embedded application, the MobileNets family of computer vision models for TensorFlow was designed [51].

3.4.2 Qualcomm Module

We have used Qualcomm SA8155P automotive development platform (ADP), as shown in figure 3.2, as another vendor of GPU-based accelerator and implemented the abstraction layer on Android OS [52].

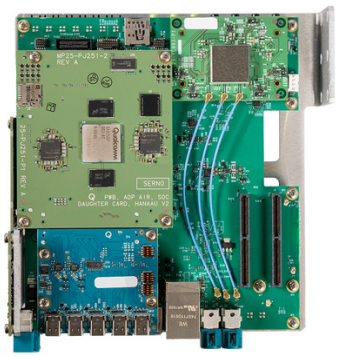

Figure 3.2: Qualcomm SA8155P ADP
[52]

Qualcomm SA8155P is an integrated, next-generation automotive cockpit platform. It has a 7nm system-on-chip (SoC) designed with custom hardware blocks, including eight Kryo CPUs, a Qualcomm Adreno 640 GPU, and a high-performance Hexagon 6 DSP.

This processor provides OEMs and ecosystem partners with access to QTI's high-performance automotive infotainment and advanced driver assists platform for developing, testing, optimizing and showcasing the next-generation in-vehicle infotainment solutions using a 12V DC power supply. In addition, this ADP supports Android 24 OS and provides connectivity for dual-band 802.11ac Wi-Fi and Bluetooth 5.0, an Ethernet port, and 4 Controller Area Networks (CAN).

3.4.3 Native Implementation of CNN

The native implementation of CNN was done using the Mobilenet model using TensorFlowLite (TFLite) on the Qualcomm board as discussed in section 3.4.1.

We built an Android application using Kotlin and Android Studio that could detect the images. The prerequisites required to run this application such as the recent version of the android studio(v 4.2+), installation of Kotlin, and necessary libraries

were installed. Android Studio is the Integrated Development Environment (IDE) for Android app development which provides the easiest tools for creating apps on every type of Android device.

Firstly, we download a pre-trained TFLite object detection model from TensorFlow hub. This downloaded model is a EfficientDet-Lite Object detection model, trained on the COCO 2017 dataset, optimised for TFLite, and designed for performance on mobile CPU, GPU, and EdgeTPU. Then, the pre-trained TFLite model is integrated into our application by using TFLite task library which makes it easy to incorporate mobile-optimised machine learning models into a mobile app. Next, we copy the model we downloaded to the assets folder of our mobile application and then, we update the Gradle file task library dependencies to run the TFLite libraries. Then, we sync our project with all the Gradles in the Android Studio.

Secondly, we load and run on-device object detection on an image by creating three different APIs. First, we write a function `runObjectDetection(bitmap: Bitmap)`in file `MainActivity.kt`. The `runObjectDetection` function receives the input image in decoded Bitmap format. Then, a simple API provided by TFLite to create a TensorImage from Bitmap which is added to the function. Next, we initialise the object detector instance by specifying the TFLite model file name and the configuration options such as the maximum number of objects that the model should detect, how confidence the object detector should be to return a detected object and finally, we feed the image to the detector which returns a list of detection that contains information about the object that the model has found in the image.

Now, the object detection app is built in the Android Studio. This file is extracted as Android Package file (apk) and uploaded to Qualcomm board through android interface. The app is run on the Qualcomm board and the time required by the GPU to detect the image is calculated using Android Debug Bridge (abd) commands.

3.4.4 Implementation of CNN using TVM compiler

The Qualcomm Snapdragon SA8155P processor from Qualcomm Technologies, Inc. (QTI) powers the third-generation Snapdragon Automotive Development Platform (ADP). It provides OEMs access to advanced driver assist and a high-performance automotive infotainment platform. The TVM compiler tool is used for analysing the CPU and GPU on Android devices using Snapdragon processors.

The TVM stack comprises two components: a compiler and a runtime. Since we use a Linux-based embedded system, a Qualcomm Snapdragon processor using the Android operating system, we can only install the runtime API in the Qualcomm module, and the compiler stack can be installed in the remote device. The TVM runtime library will be cross-compiled, and the library can be linked to the target device. The runtime can be compiled natively or cross-compiled in the local machine, but cross-compiling a runtime for a specific hardware architecture with high-performance processors is a fast process. The relevant toolchain should be

installed. The CMake is done for the specific toolchain in the local machine with additional CMake arguments. The TVM4J core is installed in the local Maven repository, using the built runtime library for aarch64 architecture, which contains all java interfaces. Then using the java dependencies, compiled the JNI using Gradle and built an apk file linking OpenCL, where the shared library for OpenCL is pulled from the android device to the local machine; that is, an android application called TVM RPC with tvm4j is generated.

The TVM RPC application launches an RPC server on the android device, which is then connected to the python script in which the TVM model is developed. Firstly, the apk file that is generated in the local host is installed on the android device using ADB commands, and the app is launched. Then, to set up the compiler stack locally, we need to build a standalone shared library (.so file) for the android device. Then using the android NDK, a standalone toolchain is generated. It can then be used to compile C++ source code and produce shared libraries for Android devices running on the arm64 architecture.

The TVM setup environment in the local machine is run using a Docker image, where we also configure the port that we will use to connect to the RPC server. The RPC tracker is then started in the host, and the RPC server installed in the android is configured with the IP address and port of the RPC tracker and turned on parallel. The device is first paired to test the pairing and communication, and we run a test script provided. That compiles a graph module on the CPU with OpenCL and Vulkan by enabling the respective target.

As mentioned earlier, we will use the pre-trained MobileNetV2 classification model with the Keras framework. A python script was developed where we downloaded the Python and its weights and loaded it to the model. We first downloaded an image to test the model and transformed its format into letters. Finally, in the script, we added the compile instructions like the target, its architecture, and the model to be compiled.
Now, the object detection app is built in the Android Studio. This file is extracted as an Android Package file (apk) and uploaded to the Qualcomm board through the android interface. Next, the app is run on the Qualcomm board, and the time required by the GPU to detect the image is calculated using Android Debug Bridge (abd) commands and build the relay. We will get three return values after 'relay.build': graph, library, and the new parameter; because we are doing some optimization that will modify the parameters while keeping the model's outcome the same, then that library is saved in the local repository. Then a remote runtime module is created using the library and executed in the TVM runtime installed in the remote machine, which is the android device.

4
Results

In this chapter, we will explain our findings and compare them to the current state of the art. We have tested and compared the performance of the abstraction layer designed for the object detection model on the Jetson Nano and Qualcomm with native compilation and TVM compiler, respectively.

4.1 Nvidia Jetson Nano

This section will discuss the results obtained while compiling Jetson Nano with native compilation and TVM compiler using the Yolov3 model with the Darknet framework. In native compilation, we have used OpenCV and compiled the model using CUDA. In TVM compilation, we have used the TVM compiler that acts as an abstraction for all DL frameworks to be executed without hardware vendor-specific programming. We have used CUDA as the parallel computing platform in both cases.

We used performance metrics like accuracy, compilation time, execution time, GPU usage, power consumed and temperature to compare both compilations.

- **Accuracy:**
  Accuracy is the percentage of correctly predicted data points among all the data points, i.e. how frequently the ML model is correct overall. The accuracy of the prediction varies a little with native and TVM compilation, as shown in figures 4.1 and 4.2 respectively. However, as per the research, TVM does not affect the model's accuracy.

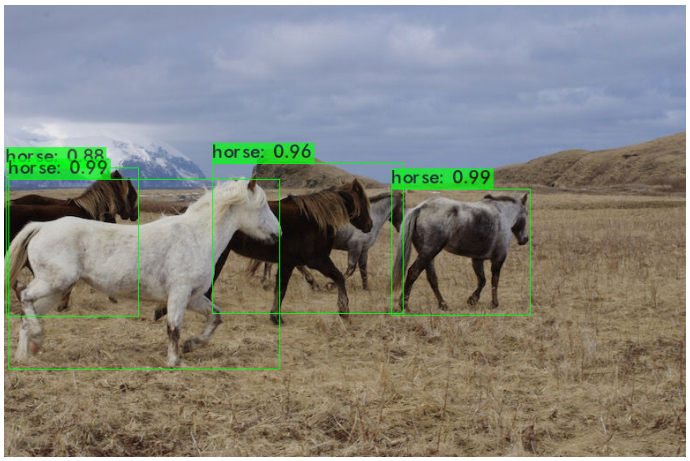

Figure 4.1: Prediction from native compilation

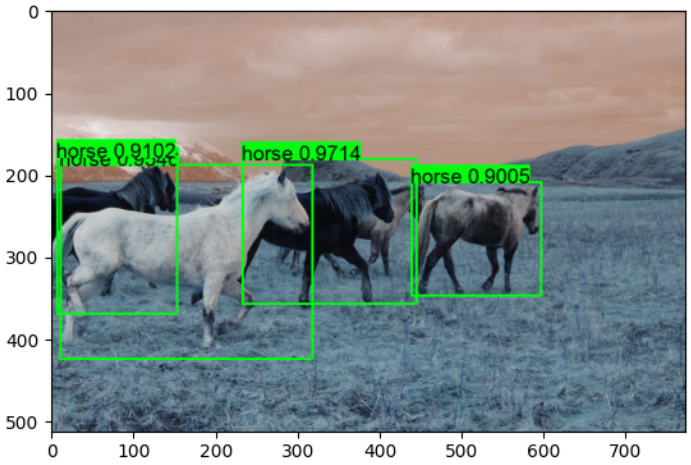

Figure 4.2: Prediction from TVM compilation

- **Compilation and Execution time:**
  Compilation time is the period when the programming code (i.e. Python) is converted to the machine code (i.e. binary code), and execution time is the amount of time required by the task to complete its execution.

  The compilation time is 1884.9860 milliseconds with an accuracy of approximately 99% for the model using the TVM compiler. Moreover, the compilation time is 13.686 milliseconds for the model using the native compiler.

- **GPU usage:**
  The GPU usage during the execution of the model with the TVM compiler is 100% when compared with native compilation as shown in figures 4.3 and 4.4 respectively. These figures represent a simple graph of GPU activity for the Nvidia Jetson Nano Developer Kit, allowing visualisation of GPU utilisation. The `Matplotlib` command from Python implements the graph as an animated graph. The Y-axis is plotted for GPU usage data, and the line is filled using the MatLab function that fills selected areas in plots. The DL model efficiently utilises the GPU and executes the tasks in parallel, resulting in a better execution time for the DL model. A 100 per cent GPU load indicates nothing in the system that might cause the graphics card to bottleneck.

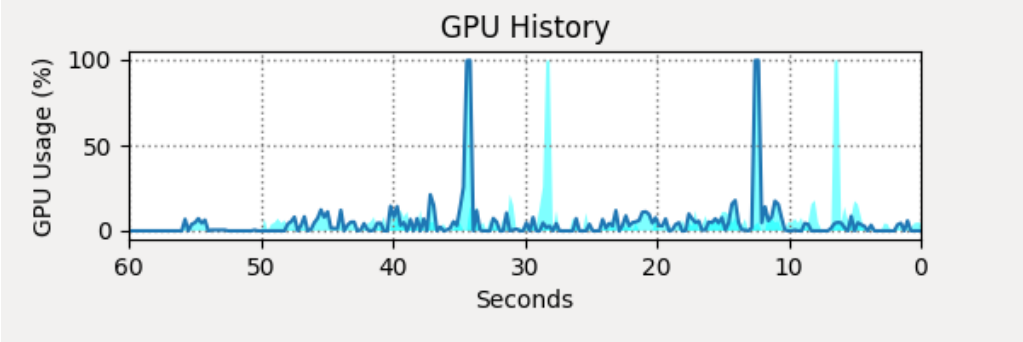

Figure 4.3: GPU usage during native compilation

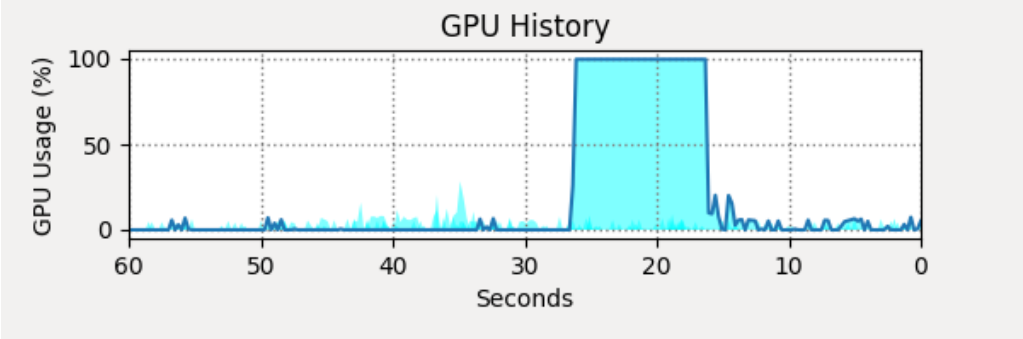

Figure 4.4: GPU usage during TVM compilation

- **Memory usage and temperature:**
  The average memory consumed by GPU by both the compilation methods is 99%, and the temperature is between 41 - 44° C during the execution of the model, as shown in figure 4.5 and 4.4 respectively.

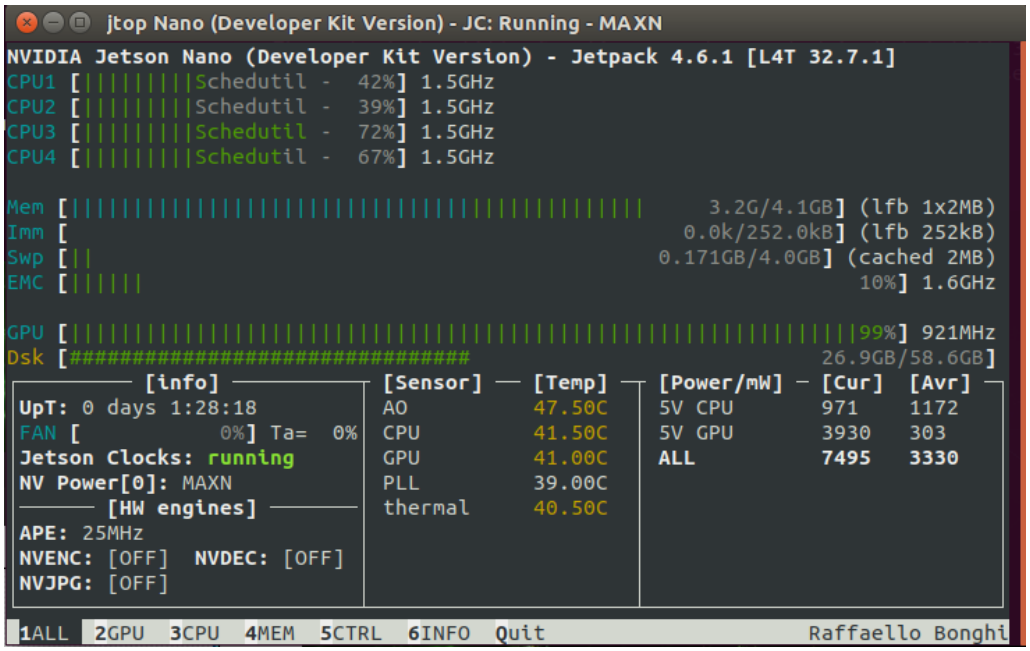

Figure 4.5: Memory usage and temperature during native compilation

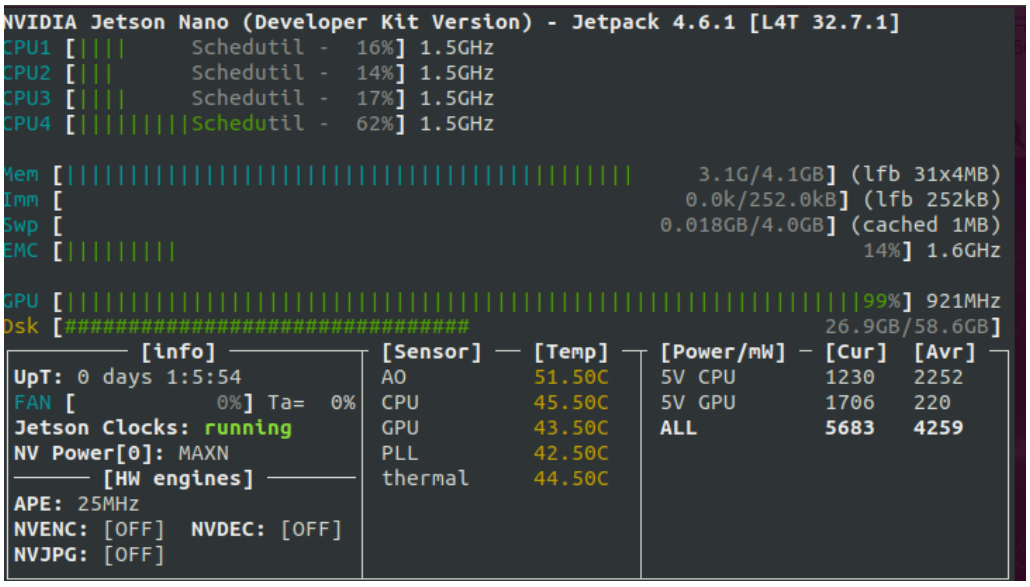

Figure 4.6: Memory usage and temperature during TVM compilation

- **Power consumption:**
  The readings represent the power consumed using the command `sudo tegrastats`. The command `tegrastats` show current and average power consumption (in x/y format, x is current power consumption, and y is average value). In addition, it displays the total GPU and CPU power consumption in milliWatts (mW).

  This result shows the power consumption while using the native compiler.
  $POM\_5V\_IN2976/3009POM\_5V\_GPU244/260POM\_5V\_CPU691/946$
  The above values show that the 2976 mW of 3009 mW total power is consumed. Similarly, 244 mW of 260 mW of GPU power and 691 mW of 946 mW of CPU power is consumed.

  This result shows the power consumption while using the TVM compiler.
  $POM\_5V\_IN2215/3761POM\_5V\_GPU164/247POM\_5V\_CPU410/1728$
  The above values show that the 2215 mW of 3761 mW total power is consumed. Similarly, 164 mW of 247 mW of GPU power and 410 mW of 1728 mW of CPU power is consumed.
  Where,
  - $POM\_5V\_IN - totalinmW$
  - $POM\_5V\_GPU - GPUinmW$
  - $POM\_5V\_CPU - CPUinmW$

  While comparing these two results shows that the TVM compiler consumes less power than the native compiler.

Table 4.1 summarises different performance metrics used to compare native and TVM compilation using Nvidia Jetson Nano. As we can see, the accuracy remains nearly the same, as the TVM compiler does not affect the model's accuracy. In comparison with the native model, the TVM compiler takes much less time to execute. The TVM compiler uses a high percentage of GPU resources compared with the native model. In terms of GPU board temperatures, both native and TVM compilers produce nearly similar results. In addition, we observed that the power consumed when the model is compiled using TVM is about 700 mW less than when it is compiled using the native compiler. Therefore, we conclude from the above results that TVM compilers provide better results than native compilers.

Performance metrics	Native	TVM compiler
Accuracy	0.99	0.97
Execution time	1884.986000ms	13.686281ms
GPU Usage	GPU Usage during execution time is less	GPU Usage during execution time is comparatively high
Temperature	40C	44C
Power consumption	POM 5V IN 2976/3009 total in mW	POM 5V IN 2215/3761 total in mW

Table 4.1: Comparison of different performance metrics for Nvidia Jetson Nano.

4.2 Qualcomm

This section will discuss the results obtained while compiling Qualcomm GPU with native compilation and TVM compiler using the MobileNetV2 model. In native compilation, we have used the MobilenetV2 model with TFLite using Android Studio. For DL compilation, the Apache TVM tool compiles and executes the MobileNetV2 model according to Qualcomm Adreno 640 GPU and CPU using OpenCL, which is done by integrating the TVM runtime API on the target hardware so that it can be accessed remotely. The android RPC app is launched on Qualcomm hardware which acts as an RPC server. The RPC tracker, which is installed in the local host, is started in a specific port, and the android device will also be configured to the same port in the android application so that it connects to the android tracker. The connection was successfully established, and that is verified as shown in figure 4.7

```
root@tvm:/workspace# python3 -m tvm.exec.query_rpc_tracker --port=3389
Tracker address 127.0.0.1:3389

Server List
------------------------------
server-address           key
------------------------------
    10.228.7.199:5001    server:android
------------------------------

Queue Status
------------------------------
key       total  free  pending
------------------------------
android   1      1     0
------------------------------
```

Figure 4.7: Tracker query test

Then, we ran a CPU, GPU test on the target device and the results are as shown in table 4.2.

which shows that the TVM IR is compiled to shared libraries without any problem and the vector addition is run on the target android device correctly.

CPU test	GPU test
104.41 secs/ops	136.203 secs/ops

Table 4.2: CPU, GPU test on the target Qualcomm device with OpenCL

mean	median	max	min	std
134.3509 ms	132.840 ms	146.616 ms	128.838 ms	5.554 ms

Table 4.3: Execution time summary for MobileNetV2 output according to Qualcomm Adreno 640 GPU and CPU using OpenCL

The MobilenetV2 model is then executed using the Python script we created, which enables the model to operate on GPU and OpenCL in the Android target as shown in table 4.3 Running the model using GPU is comparatively slower than running the model using CPU, thus we have to optimise the schedule according to the GPU architecture.

Table 4.4 summarises different performance metrics used to compare native and TVM compilation using Qualcomm GPU. Due to time limitations, we could not test different performance metrics and perform optimisation for the model on Qualcomm platform. Similar to the results we obtained in Nvidia, the accuracy of model when tested on the both the compilers remains nearly the same. However, we observed that there is no much difference in the execution time when the model is executed on both the compilers. This is because we did not tune the model because of time constraints.

Performance metrics	Native	TVM-compiler
Accuracy	0.99	0.97
Execution time	130ms	134.3509ms

Table 4.4: Comparison of different performance metrics for Qualcomm GPU.

4.3 Summary and Key Outcomes

The above results show that executing a DL model with the TVM compiler gives us better results and proves a good choice. This DL compiler abstracts the hardware irrespective of the programming language and delivers better performance when compared with the native implementation of the model. In the TVM compiler, the programming model is developed in the compiler's backend, which can also be tuned and optimised. It is also efficient as it reduces the programmer's work as he/she does not have to design the model for every DL framework and hardware accelerator chip. Using the DL compiler as an abstraction layer is an efficient, easy, and less time-consuming way to implement a model in the hardware. By effectively using its resources, the TVM-DL compiler, as an abstraction method seems to deliver higher outcomes on both Nvidia and Qualcomm platforms. Good usage of GPUs by Nvidia and Qualcomm during TVM compilation, less power consumption when implementing a model, and better execution time, which has been well shown in the above results.

5
Conclusion

In this thesis, we conducted a thorough analysis of the best approach for abstracting the DNN application that needs to be used with DL accelerators so that the application need not be altered every time based on the architecture and details of the hardware accelerators. First, we analysed different types of abstractions, for instance, using frameworks like OpenCL and Vulkan to abstract the DL application, but that method had drawbacks. For instance, a programming model such as CUDA produces better results for Nvidia accelerators than OpenCL. Then, we explored further the standard architecture used by the current DL compilers, including the multi-level IR, the frontend and the backend. Where it supported multiple frameworks in the frontend and various hardware targets and programming models in the backend, this served the purpose of our problem and was an excellent, versatile alternative. Finally, after examining several compilers, we selected one appropriate for the hardware provided in the problem statement. Then, we conducted the test by first developing the DL application and implementing it using the TVM compiler on both hardware platforms. Next, we tested the native implementation of the model. Finally, we evaluated the outcomes based on performance metrics such as execution time, GPU usage, power consumption, and accuracy, and we also monitored the temperature. As a result, we concluded that utilising DL compilers is an excellent technique to abstract and has no discernible effect on the DL model's performance; in fact, it performed better than native implementation in execution time and GPU usage.

5.1 Future work

There can be limitations in using DL compilers in some cases where if none of the DL compilers supports a particular framework or hardware target, in that case, an alternative solution has to be researched. However, due to time constraints, we could not conduct additional testing on various performance metrics for the Qualcomm accelerator. As a result, additional experiments can be conducted in the future to assess the effectiveness of using TVM as a DL compiler for the Qualcomm accelerator.

Bibliography

[1] Intel, "Accelerate data from edge to cloud." [Online]. Available: https://www.intel.com/content/www/us/en/products/details/fpga/platforms/pac.html

[2] "Accelerator cards." [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo.html

[3] Run:AI, "Nvidia CUDA in deep learning." [Online]. Available: https://www.run.ai/guides/nvidia-cuda-basics-and-best-practices

[4] K. H. et al., "Exploration of OpenCL heterogeneous programming for porting solidification modeling to CPU-GPU platforms," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 4, 2018 [Online].

[5] I. McCallum, "Intel® quickassist technology accelerator abstraction layer (aal)," Intel Corporation, 2007 [Online]. [Online]. Available: https://blog-assets.oss-cn-shanghai.aliyuncs.com/18951/6103fadf4dd3a0dfbd0d637308a94b8e99e799d2.pdf

[6] K. Karimi, N. Dickson, and F. Hamze, "A performance comparison of cuda and opencl," *Computing Research Repository - CORR*, vol. arXiv:1005.2581, 05 2010.

[7] V. Meel, "Yolov3: Real-time object detection algorithm (what's new?)." [Online]. Available: "https://viso.ai/deep-learning/yolov3-overview"

[8] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[9] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 708–727, mar 2021. [Online]. Available: https://doi.org/10.1109%2Ftpds.2020.3030548

[10] I. OpenGenus, "Tvm: A deep learning compiler stack." [Online]. Available: https://iq.opengenus.org/tvm-deep-learning-compiler-stack/

[11] "ngraph." [Online]. Available: https://www.intel.com/content/www/us/en/artificial-intelligence/ngraph.html

[12] "Nvidia tensorrt." [Online]. Available: https://developer.nvidia.com/tensorrt

[13] TensorFlow, "Xla: Optimizing compiler for machine learning." [Online]. Available: https://www.tensorflow.org/xla

[14] R. Khandelwal, "A basic introduction to tensorflow lite," Jun 2020. [Online]. Available: "https://towardsdatascience.com/a-basic-introduction-to-tensorflow-lite-59e480c57292"

[15] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhabarov, J. Hegeman, R. Levenstein, B. Maher, N. Satish, J. Olesen, J. Park, A. Rakhov, and M. Smelyanskiy, "Glow: Graph lowering compiler techniques for neural networks," *CoRR*, vol. abs/1805.00907, 2018. [Online]. Available: http://arxiv.org/abs/1805.00907

[16] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: end-to-end optimization stack for deep learning," *CoRR*, vol. abs/1802.04799, 2018. [Online]. Available: http://arxiv.org/abs/1802.04799

[17] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories." Association for Computing Machinery, 2008. [Online]. Available: https://doi.org/10.1145/1345206.1345210

[18] L. Li, "Programming abstractions and optimization techniques for gpu-based heterogeneous systems," Ph.D. dissertation, Linköping University Electronic Press, 2018. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-145304

[19] "Onnx." [Online]. Available: https://onnx.ai/

[20] A. Moreton–Fernandez, H. Ortega–Arranz, and A. Gonzalez–Escribano, "Controllers: An abstraction to ease the use of hardware accelerators," *The International Journal of High Performance Computing Applications*, vol. 32, no. 6, pp. 838–853, 2018. [Online]. Available: "https://doi.org/10.1177/1094342017702962

[21] S. Sumit, "A comprehensive guide to convolutional neural networks — the eli5 way," Dec 2018. [Online]. Available: "https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53"

[22] G. Verma, Y. Gupta, A. M. Malik, and B. Chapman, "Performance evaluation of deep learning compilers for edge inference," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021, pp. 858–865.

[23] T. Georgiou, Y. Liu, W. Chen, and M. S. Lew, "A survey of traditional and deep learning-based feature descriptors for high dimensional data in computer vision," *International Journal of Multimedia Information Retrieval*, vol. 9, pp. 135 – 170, 2019.

[24] "Convolutional networks." [Online]. Available: "http://ethen8181.github.io/machine-learning/deep_learning/cnn_image_tensorflow.html"

[25] R. Yamashita, M. Nishio, R. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," *Insights into Imaging*, vol. 9, 06 2018.

[26] M. Mishra, "Convolutional neural networks, explained," August 2020. [Online]. Available: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939

[27] P.-C. Shih, C.-C. Hsu, and F.-C. Tien, "Automatic reclaimed wafer classification using deep learning neural networks," *Symmetry*, vol. 12, no. 5, 2020. [Online]. Available: https://www.mdpi.com/2073-8994/12/5/705

[28] T. C. Lianmin Zheng, Eddie Yan, “Automatic kernel optimization for deep learning on all hardware platforms,” Oct 2018. [Online]. Available: "https://tvm.apache.org/2018/10/03/auto-opt-all"

[29] S. Hall, “Apache tvm: Portable machine learning across backends,” 28 Dec 2020. [Online]. Available: "https://thenewstack.io/apache-tvm-portable-machine-learning-across-backends"

[30] M. A. Meselhi, S. M. Elsayed, D. L. Essam, and R. A. Sarker, “Fast differential evolution for big optimization,” in *2017 11th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)*, 2017, pp. 1–6.

[31] G. Watson, “Introduction to gpus,” 2017. [Online]. Available: https://nyu-cds.github.io/python-gpu/01-introduction/

[32] tutorialspoint, “CUDA - introduction to the GPU.” [Online]. Available: https://www.tutorialspoint.com/cuda/cuda_introduction_to_the_gpu.htm

[33] E. László, P. Szolgay, and Z. Nagy, “Analysis of a gpu based cnn implementation,” in *2012 13th International Workshop on Cellular Nanoscale Networks and their Applications*, 2012, pp. 1–5.

[34] G. V. Stoica, R. Dogaru, and E. C. Stoica, “High performance cuda based cnn image processor,” Ph.D. dissertation, Linköping University Electronic Press, 2015. [Online]. Available: https://www.academia.edu/73884400/High_performance_CUDA_based_CNN_image_processor

[35] J. Dsouza, “What is a gpu and do you need one in deep learning?” Apr 2020. [Online]. Available: "https://towardsdatascience.com/what-is-a-gpu-and-do-you-need-one-in-deep-learning-718b9597aa0d"

[36] nvidia, “What’s the difference between a cpu and a gpu?” [Online]. Available: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

[37] GeeksForGeeks, “Introduction to cuda programming.” [Online]. Available: https://www.geeksforgeeks.org/introduction-to-cuda-programming/

[38] Wikipedia, “Adreno.” [Online]. Available: https://en.wikipedia.org/wiki/Adreno

[39] Q. developer network, “Adreno™ graphics processing units.” [Online]. Available: https://developer.qualcomm.com/software/adreno-gpu-sdk/gpu

[40] P. Solutions, “The qualcomm snapdragon graphics pipeline explained,” March 2020. [Online]. Available: "https://www.penguinsolutions.com/about-us/newsroom/qualcomm-snapdragon-graphics-pipeline"

[41] Notebookcheck, “Qualcomm adreno 640.” [Online]. Available: https://www.notebookcheck.net/Qualcomm-Adreno-640-Graphics-Card.374761.0.html

[42] A. Das, Y. D. Kwon, J. Chauhan, and C. Mascolo, “Enabling on-device smartphone gpu based training: Lessons learned,” in *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2022, pp. 533–538.

[43] M. A. Meselhi, S. M. Elsayed, D. L. Essam, and R. A. Sarker, “Fast differential evolution for big optimization,” in *2017 11th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)*, 2017, pp. 1–6.

[44] M. Assumma, “Nvidia jetson nano: What is it, and what can it do?” June 2019. [Online]. Available: https://www.newegg.com/insider/nvidia-jetson-nano-dev-kit-makers-overview-what-can-it-do/

[45] nvidia developer, “Jetson nano developer kit.” [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[46] nvidia newsroom, “Nvidia announces jetson nano: 99 tiny, yet mighty nvidia cuda-x ai computer that runs all ai models,” Monday, March 18, 2019. [Online]. Available: https://nvidianews.nvidia.com/news/nvidia-announces-jetson-nano-99-tiny-yet-mighty-nvidia-cuda-x-ai/ /-computer-that-runs-all-ai-models/

[47] A. Allan, “Introducing the nvidia jetson nano,” 2020. [Online]. Available: https://aallan.medium.com/introducing-the-nvidia-jetson-nano-aaa9738ef3ff

[48] S. Valladares, M. Toscano, R. Tufiño, P. Morillo, and D. Vallejo-Huanga, “Performance evaluation of the nvidia jetson nano through a real-time machine learning application,” in *Intelligent Human Systems Integration 2021*, D. Russo, T. Ahram, W. Karwowski, G. Di Bucchianico, and R. Taiar, Eds. Cham: Springer International Publishing, 2021, pp. 343–349.

[49] L. Pettersson, “Convolutional neural networks on fpga and gpu on the edge: A comparison,” Ph.D. dissertation, Uppsala University, Signals and Systems, 2020.

[50] “pjreddie/darknet.” [Online]. Available: https://github.com/pjreddie/darknet

[51] kdnuggets, “Comparing mobilenet models in tensorflow,” March 2019. [Online]. Available: "https://www.kdnuggets.com/2019/03/comparing-mobilenet-models-tensorflow.html"

[52] Lantronix, “Sa8155p automotive development platform.” [Online]. Available: https://www.lantronix.com/products/sa8155p-automotive-development-platform/