Deep Learning on a Low Power GPU

Perry Gibson

MInf Project (Part 1) Report

Master of Informatics
School of Informatics
University of Edinburgh

2018

Abstract

Deep Learning applications have surged in popularity in research and industry in recent years, as dataset size and quality has grown, and computational capability has improved. They have proved themselves effective in the areas of image and audio recognition, and their ubiquity is on the increase. However, the high computational and storage costs of these systems mean that resource constrained environments (such as embedded systems and mobile phones) require the investigation of optimisation techniques at both the network architecture layer, and the system layer. This report explores the variety of optimisation techniques used to improve the execution time of deep neural network inference, with a focus on low-power Mali-T628 GPU. We present an implementation of the MobileNet architecture with support for parallelised operations using OpenCL, and related libraries clBLAS and CLBlast. Our best improvement over a serialised version is 2.95X, though there is scope for further improvement

Acknowledgements

I would like to thank Professor Michael "Mike" O'Boyle for his support and guidance.

Additionally, my gratitude to Jose Cano for his expertise and motivation throughout the span of the investigation.

And finally to my family and friends for their ongoing support and companionship.

Table of Contents

Chapter 1

Introduction

The pervasiveness of heterogeneous computing platforms, such as consumer mobile devices, coupled with the growing demands for production grade Deep Neural Network applications has seen a plethora of research and tools focused on optimising the performance of the former on the latter.

Current real world deployment of such systems have included real-time audio transcription and translation, vision tasks such as industrial quality control, medical picture analysis, and robotic sensing.

Modern production systems are often trained for long periods of time, on the order of days and weeks on large scale computing infrastructure. Once trained, their usage is much less time consuming, typically on the order of microseconds to minutes to classify an image. However, when they are deployed they are often required to be in much more constrained environments, or suffer the latency cost of sending data to a remote process. In some situations (e.g. bandwidth limited environments such as oil rigs or disaster zones), this may not be possible.

Therefore, the need to explore and find techniques for running these systems in the heterogeneous environments that they are intended for is essential, as many of these applications are used to preserve and save lives, as well as more indirect benefits such as reducing operational costs for organisations which utilise these tools.

This project explores the state of neural network optimisation efforts, with a focus on the ARM Mali GPU, a low power GPU often found in mobile devices. The network we use is "MobileNet", an image recognition convolutional neural network with an architecture that balances accuracy with computational cost.

1.1 Structure of this report

Chapter 2 presents relevant background information, including the MobileNet architecture, the deep learning optimisation techniques, the OpenCL paradigm, and the Mali-T628 GPU on the ODROID-XU4 board.

Chapter 3 explores some of the related work relevant to our investigation.

Chapter 4 details the methods we investigated to answer our research questions, and the implementation details of our final system. It also presents the results of our experiments.

Chapter 5 discusses some avenues for future work, given the space mapped out in this project.

1.2 Key Achievements

- Investigated the OpenCL paradigm, and its behaviour on the Mali-T628 GPU
- Altered the "Darknet" deep learning framework to support OpenCL operations
- Adapted Darknet GNU Makefile and CMake build infrastructures to support the new OpenCL backend
- Researched the space of deep learning optimisation techniques

Chapter 2

Background

2.1 Neural networks

Artificial neural networks are a class of machine learning models. That is to say, computer programs which when given examples of some data, can extract information about the structure of that data, and use it to make inference when shown new data. Inspired by the structure of animal brains, these systems are comprised of 'neurons' which take multiple inputs, combine them using some function, and output a value. The outputs of these neurons are generally fed as input to other neurons, and by layering and other architectural arrangements of neurons, useful representations of the data can be created.

For inference to occur, the data is given as input to the network, a vector or matrix. Transformations, combinations which weight some values higher than others, and finally an output layer which gives some label given the input.

There are a variety of training techniques which allow these models to be created, however these are not relevant in this work. Put simply, the untrained network is initialised according to some scheme, and shown some training data. Its output is measured, and its weights are altered to bring the output closer to the a correct output. Then the process is repeated until the accuracy of the output given the input is deemed acceptable.

The metaphor of a neural network is useful for conceptualising how a system works. We typically split the network into "layers" of neurons that are fed the same input, but compute on it in a slightly different way. However, when implemented these systems for the most part boil down to matrix multiplication and data reshaping. For example, the system in Figure 2.1.

2.1.1 Convnets

Convolutional layers are class of layers which are commonly utilised in neural networks. A basic layer consists of a set of learnable filters, also known as kernels. Typ-

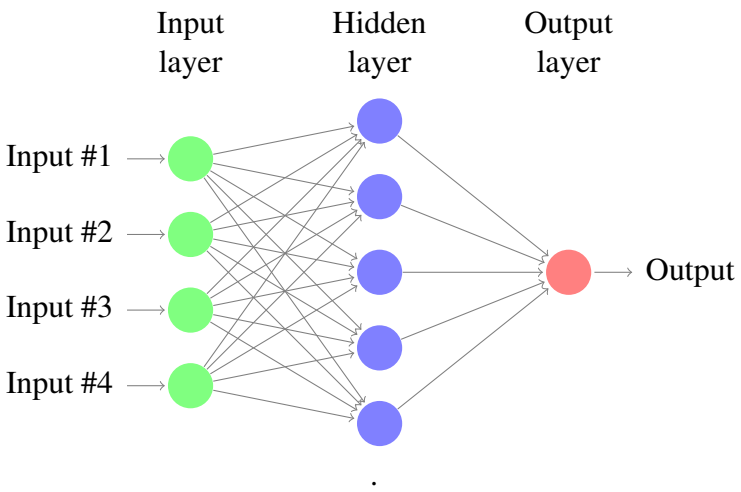

Figure 2.1: Two layer neural network

ically these filters will be small, although they extend through the full depth of input volume.

A convolutional neuron will apply its filter to the input, sometimes a subset of the input if striding is used. One of the motivations for this is that it reduces the number of free parameters, when compared to a typical fully-connected layer, which means that training is faster, and trained models are smaller.

Additionally convolutional layers are often paired with 'pooling layers', which combines the outputs of multiple neurons into a single output. Although this process loses information, it is useful at further reducing the complexity of a model.

2.1.2 Separable Depthwise convolutional layers

A type of convolutional layer utilised in this work is a separable depthwise convolutional layer, first introduced in [27]. This performs a spatial convolution, keeping channels separate, and then performs a depthwise convolution.

2.1.3 Datasets

The Imagenet dataset [14] is a standard in machine learning problems, where over 14 million images of diverse scenes are hand-annotated to describe what is pictured. Systems are generally judged on their 5 best guesses.

The CIFAR-10 dataset [13] similarly, is a standard labelled image dataset, comprised of 60,000 32x32 colour images, across 10 classes.

2.2 MobileNet

MobileNets, first introduced in [7], are a class of deep convolutional neural networks, with a focus on computational efficiency. The seminal paper presented systems trained on the ImageNet dataset (see 2.1.3) The key technique that make MobileNets efficient is the usage of depthwise separable convolutions, which were found to produce reasonable classification accuracy while keeping models lightweight.

Pre-trained MobileNet models are distributed through official TensorFlow sources [30].

2.3 Parallel computing and GPGPU

Parallel computation is a type of computation where the execution of processes is carried out simultaneously.

One approach to parallelism is SIMD (single instruction, multiple data). This is where multiple processing elements perform the same operation on multiple pieces of data simultaneously.

Parallelism is not possible for all operations, since some components of a system may need to data from another component to be processed first before they can be executed. The data dependency problem places limits on much a system can be parallelised, and identifying elements which are not dependent is often one of the first steps to implementing parallelisation.

2.3.1 GPGPU

Graphics cards, or GPUs (graphics processing units) are pieces of computer hardware that are most commonly used for running computations specific to fast image rendering. Their architecture typically features several hundred "shader cores", similar to CPU cores, though slower. They may also feature their own pool of RAM.

Near the dawn of the 21st century, the research community began to look at their potential as general purpose devices. This approach is commonly referred to as GPGPU (General-purpose computing on graphics processing units). One of the first attempts in this area was matrix multiplication [16]. As the performance of GPUs improved, and the accessibility of programming interfaces to these devices improved, GPGPU became more mainstream.

One of the first direct usages of GPGPU in the area of machine learning was [28]. Today, thanks to the orders of magnitude speedup that GPGPU can provide, a sizeable portion of deep learning research utilises GPGPU.

2.4 OpenCL

The Open Computing Language (OpenCL) provides a common programming interface for interacting with heterogeneous pieces of computationally capable hardware, such as GPUs, CPUs, and DSPs. Programs which use OpenCL consist of two parts: host-code and device-code.

Device-code is executed on an OpenCL device (e.g. GPU), and describes the operation we want to parallelise (e.g. matrix multiplication). The functions which are executed on devices are known as "kernels".

Host-code is executed on the CPU. It is used to manage the OpenCL environment to be used, send data to be used with the device-code, and load and send device-code to the device. It can be written in any language which has OpenCL bindings [1]. One could think of it as the infrastructure which enables device-code.

The standard is defined by the Khronos Group, a consortium of diverse organisations. These include, but are not limited to: Advanced Micro Devices, Inc., Apple Inc. (who originally developed it, and hold trademark rights), Codeplay Software Limited, and Qualcomm [31].

2.4.1 Terminology

We shall define here relevant terms used in OpenCL terminology.

- **Devices**: A *device* is typically corresponds to a GPU, CPU, or some other processor. A *device* will possess one or more *compute units*.
- **Compute Units**: A *device* is a collection of *compute units*.
- **Kernels**: functions which are executed on *devices*. Equivalent to a C functions, take arguments, no return value.
- **Work-items**: a single *work-item* is one instantiation/execution of a *kernel* on a *device*.
- **Work-groups**: a collection of related *work-items* that are executed on the same *compute unit*. *Work-items* in the same *work-group* execute the same kernel, and share *local memory*.
- **Global memory**: Stores data for the entire *device*. Could be VRAM for a GPU, though for the Mali on the ODROID-XU4 GPU memory and board memory are unified.
- **Local memory**: Stores data for *work items* in a *work group*. Generally *work items* can access *local memory* much faster than *global memory*.
- **Private memory**: Stores data for an individual *work item*. Generally the fastest access times, though with a limited address space.

[1] Bindings for OpenCL have been written for languages such as Python, Haskell, and C++

- **Command queue**: *Devices* receive *kernels* through a *command queue*. The a queue can contain multiple kernels, which the *device* will execute in sequence.
- **Context**: The *context* encapsulates *devices* and *command queues* during a particular runtime, essentially defining the environment in which an OpenCL program runs in. It permits data transfer between *devices*.

2.4.2 OpenCL Paradigm

OpenCL defines standard hardware abstractions for the stuff it uses. Functions are called kernels. Kernels are written in a language called OpenCL C, based on C99.

Performance is not guarnteed for a given kernel for different device types.

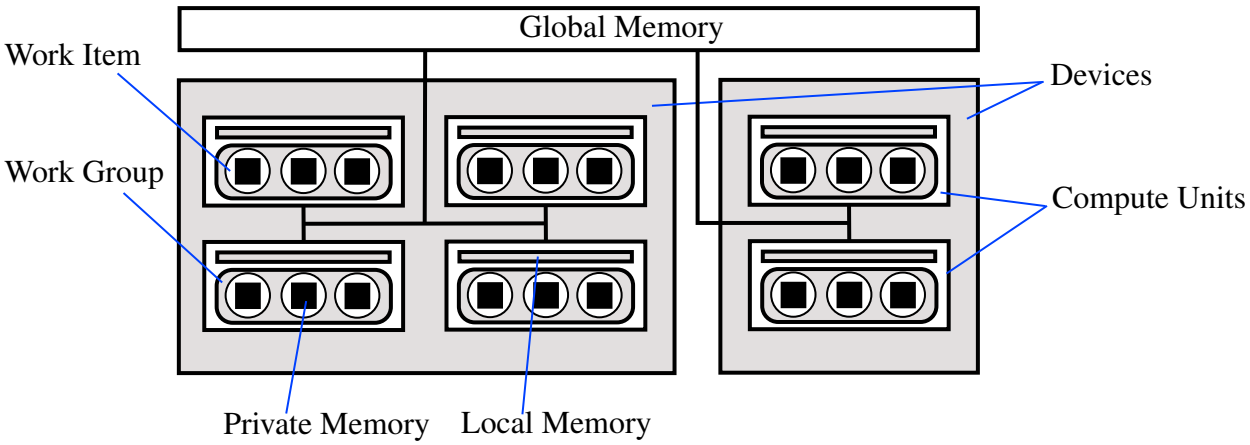

Figure 2.2: The OpenCL device model

OpenCL's API represents CPUs, GPUs, and other computing hardware as **devices**. Each device contains one or more **compute units**, which generally represents a CPU or shader core. A **compute unit** is assigned a **work group**, which is a collection of **work items**, with shared **local memory**. Each compute unit contains one or more SIMD lanes, which means that **work items** in a group may be executed sequentially or concurrently.

2.5 Other GPGPU APIs

A popular alternative to, and precurser of OpenCL is CUDA: a proprietary GPGPU API from nVidia. However it is designed and optimised for nVidia GPUs, and the specification is not freely available to port to other architectures. Thus alternatives such as OpenCL are needed for non-nVidia hardware. This has not stopped CUDA being one of the most popular ways researchers use GPGPU to train their models in recent years.

RenderScript is a component of the Android OS, which provides an API for 3D graphics rendering, as well of general computing. It can be used for both CPUs and GPUs.

2.6 ASICs

It is possible that further performance and energy efficiency can be achieved by running deep learning applications on specialised hardware: application specific integrated circuits (ASICs). By focusing on optimising matrix multiplication in chip design, significant improvements can be made for systems which rely on such operations.

Examples of these include Google LLC's "Tensor Processing Unit" (TPU) [2], and Bitmain's "Tensor Computing Processor" [24]. Compared to a Nvidia K80 GPU, Google's TPU claims to be 15X-30X faster, with 30X-80X improvement in TOPS/Watt (energy efficiency) [12].

One perfomance enhancing is a technique called "quantisation", where large 32 or 16 bit floating point numbers (such as weights) are represented by a shorter bit string, say 8 bits. This sacrifices precision for performance. This technique does not work well for training, as maximum precision is preferred, but can be effective for the forward pass, which is more noise tolerant [8]. The Google TPU has 65,536 8 bit integer multipliers "The popular GPUs used widely on the cloud environment contains a few thousands of 32-bit floating-point multipliers. As long as you can meet the accuracy requirements of your application with 8-bits, that can be up to 25X or more multipliers. " Although there may be some operations in machine learning that do not lend themselves well to an ASIC design, matrix multiplication remains one of the key workhorses of such systems. Although still in their infancy, these ASICs could become a cornerstone of future research. Initially these will be used in data centres and specialist labs, however there may be a market for tensor processors in embedded and mobile devices in the future. There is also scope for the use of FPGAs in deep learning applications, as discussed in [15].

2.7 The Mali-T628 GPU on the ODROID-XU4 board

The Mali line of GPUs are designed by ARM, who give licences for others to use. It offers API support for OpenCL (1.1), OpenGL, DirectX, and Google RenderScript.

The ODROID-XU4 board [22] is a single board computer produced by Hardkernel Co. Ltd, a South Korean designer/manufacturer. It features an ARM Cortex-A15 and Cortex-A7 big.LITTLE CPU, ARM Mali-T628 GPU, and 2GB of shared LPDDR3 RAM.

2.7.1 Mali Architecture

The Mali architecture differs from desktop GPUs in a number of ways. For example, Mali has no distinction between global and local memory (i.e. the GPU has the same memory pool as the rest of the device, and must share with the OS, and other programs). Desktop GPUs generally have their own pool of memory, seperate from the rest of the host machine. This means many algorithms which optimise data transfer between global and local memory will not be useful in the context of the Mali-T628 GPU.

The Mali-T628 architecture, shown in Figure 2.3, has support for 1 to 8 shader cores. The one found on the ODROID-XU4 has 6 shader cores, split over 2 OpenCL devices. Desktop GPUs typically have on the order of several hundred shader cores.

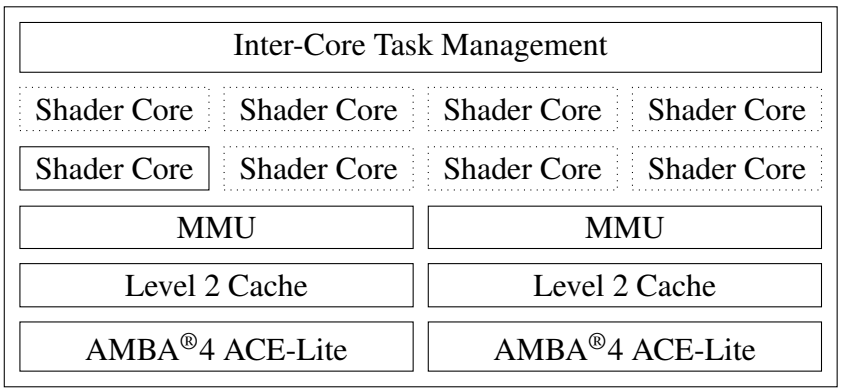

Figure 2.3: The Mali-T628 GPU architecture

The Mali GPU is exposed to OpenCL, with attributes shown in Table 2.1.

	Device 1	Device 2
Computation		
Compute Units	4	2
Clock Speed	600MHz	600MHz
Max Work Group Size	256	256
Max Work Item Size	256,256,256	256,256,256
Memory		
Local Memory (bytes)	3067863656	3067863656

Table 2.1: Specifications of the Mali-T628 GPU on the ODROID-XU4

2.8 Deep Learning Frameworks

Most deep learning applications are developed in frameworks which provide efficient implementations of common operations, such as data handling, training algorithms, and network construction. There are dozens of such frameworks available, such as TensorFlow, Caffe, Torch, and Theano. Programming language used, scalability, and

specialised functionality are some of the reasons that one might choose one framework over another.

2.8.1 TensorFlow

TensorFlow [1] is an Apache licensed framework written in C++ and Python developed by Google. It is designed to be efficient, adaptable, and scalable, and is used in many modern research and production systems. Bindings exist for a growing number of programming languages, such as R, Haskell, C++, Python, Java, and Go.

2.8.2 Darknet

Darknet [25], is a Free and Open Source neural network framework written in C, with optional CUDA support. Natively, it has support for a variety of architectures such as CNNs and DNNs. Network architecture is defined in a raw text file which describes each layer. Weights are stored in a separate file as a long string of floats. Networks are trained by passing arguments such as the architecture and the location of the training data to the Darknet executable.

The core development of Darknet is not particularly active, given that it is maintained by a single person. However it has a moderately lively community, who have published various forks which add or alter the functionality of Darknet in some way. One of these is Darknet-Mobilenet, which adds support for depthwise separable convolutional layers, essential for the MobileNet architecture.

2.9 Deep Learning Optimisation techniques

Typical constraints on embedded systems include power consumption, memory and disk space, and compute power. In this work we focus on how to best utilise compute power, as the size of the model is within the capabilities of the device, and we do not examine power consumption.

2.9.1 Parallelisation

For neural networks, the main computational cost comes from linear algebra operations such as matrix multiplication. There are several BLAS libraries which can be used, which utilise parallelism to improve performance.

1. **OpenBLAS** A BLAS library specifically for CPUs, with efficient implementations of most functions, for a variety of architectures. It also supports multi-threading.

2. **clBLAS** A BLAS library, which provides efficient OpenCL parallelised implementations of common linear algebra routines.
3. **CLBlast** is an OpenCL BLAS library, introduced in [21]. Its distinguishing feature from other libraries such as clBLAS is that it makes the process of tuning for specific environments more accessible, and comes with recommended settings for a large variety of devices, including Mali GPUs.

2.9.2 Vectorisation

Another way to achieve parallelism is the use of vectorisation. Programs which utilise vector operations are able to run multiple operations on a single instruction, instead of re-instantiating the instruction for every operation. Hardware which supports this "Single Instruction, Multiple Data" (SIMD) technique can see algorithms with significant performance improvement.

An basic example of this would be an element-wise addition of two arrays `A` and `B`, of length 1024.

```
// scalar approach (non-vectorised)
for (i = 0; i < 1024; i++) {
  C[i] = A[i]+B[i];
}

// vectorised approach
for (i = 0; i < 1024; i+=4) {
  C[i:i+3] = A[i:i+3]+B[i:i+3];
}
```

Notice that in the scalar approach the loop runs 1024 times, whereas the vectorised version runs 256 times. With the simplifying assumption that one SIMD operation takes the same amount of time as a scalar one, this is a 4X speedup.

Though device dependent, OpenCL has support for vector data sizes of 2, 4, 8, and 16 elements, across most built-in datatypes.

To take best advantage of vectorisation (i.e. minimising the the number of memory reads, and complexity of pointer arithmetic), data should be stored in memory in a sensible manner. Data should be stored sequentially with respect to its need in computation. Data which goes through many transformations (such as that of multi-layer neural networks) will need to be reshaped regularly to match the requirements of the next operation. Thus there is a trade-off to be considered with the cost of reshaping data versus the speedup afforded by a more efficient vectorised version, though this cost is typically relatively low.

In memory, matrices are generally stored in a row-major layout, as shown in Figure 2.4.

However, this layout does not necessarily reflect the sequence of when elements are required during computation. Thus there are a number of novel memory layouts that

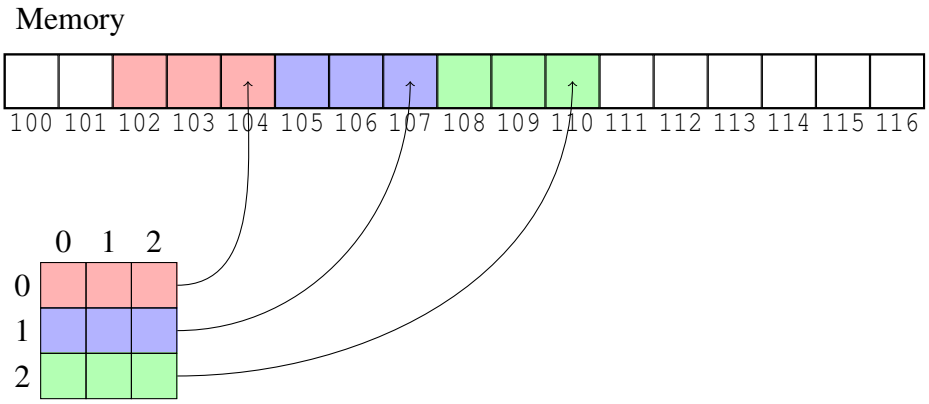

Figure 2.4: Row-major matrix memory layout

can be used. This can be as simple as switching from row-major to column-major (where elements in the same column of the matrix are stored sequentially in memory, see Figure 2.5).

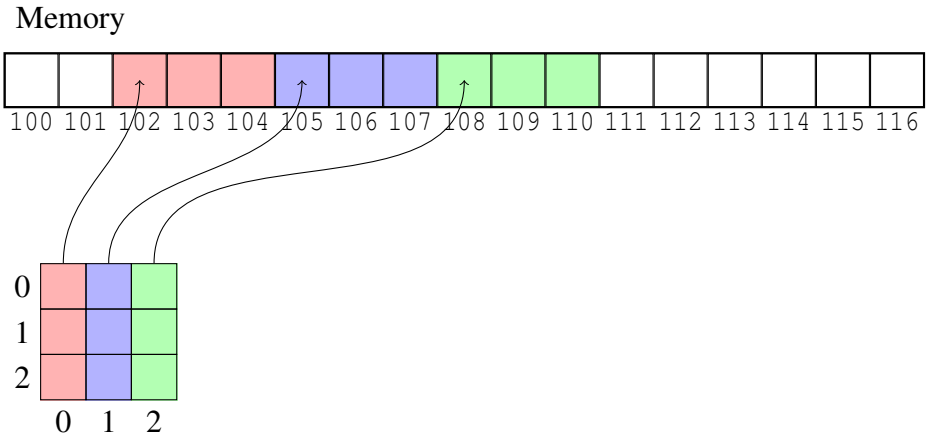

Figure 2.5: Col-major matrix memory layout

For example, in the multiplication $AB = C$, one takes the dot product of every row of A with every column of B. Thus a row-major layout for A and a column-major layout for B means that data will be stored in the order that it is needed.

2.9.3 Quantisation

Quantisation is a technique for reducing the memory footprint and computational cost of neural networks. Networks are trained using floating point number weights, for higher training effiency. However, the weights can be rescaled to smaller weights, such as 8-bit integer weights, or sometimes even less. This results in a loss of accuracy, however some network architectures do not see significant accuracy drops, since quantisation has a similar effect to adding more noise to input data.

2.9.4 Matrix sparsity

A sparse matrix is a matrix in which most elements are zero. There are compact datastructures which can represent sparse matrices in memory, saving space by indirectly representing zeros. Specialised algorithms which perform matrix operations on sparse matrices more efficiently than algorithms designed for dense matrices.

If one has a matrix of a very high rank, this saves a lot of space. One can use a sparse matrix data structure.

Chapter 3

Related work

In this chapter we present some of the key pieces of work related to our research topic.

3.1 Deep Learning frameworks with accelerators

GPGPU has become a key part of deep learning research, due to the vast computational cost associated with training complex models. CUDA with nVidia GPUs remains one the most popular ways to do so. The majority of frameworks have some level of integration with CUDA, so we will focus on frameworks which use more open standards such as OpenCL.

Caffe touts an experimental OpenCL utilising branch, adapted from work presented in [32]. Compared to an Intel CPU without OpenCL, using AlexNet [14], they were able to produce inference speedup on the order of CUDA (17x vs 22x) on a GTX 980 GPU.

3.1.1 SYCL and TensorFlow

TensorFlow has optional support to use SYCL [1] as part of its math backend (Eigen [6]). It performs the same acceleration tasks as CUDA, which is one of the popular tools that researchers use to train networks (on GPUs only). SYCL, another standard defined by the Khronos Group, is a native C++ API which enables interaction with OpenCL devices, without the need to use external code for kernels. The advantages of this is that it can simplify the development process, by allowing better access to C++ features such templates and classes, and reducing the amount of host side code used in setting OpenCL runtime parameters.

[1] pronounced "sickle"

3.2 Optimising Deep Learning Execution

There are a variety of techniques used to improve neural network execution. Using accelerators such as OpenCL is one way. However one can also alter the nature of the network itself.

3.2.1 Network level

Changing aspects of the network's architecture can improve efficiency, sometimes at the cost of accuracy.

For example, the paper "A Quantization-Friendly Separable Convolution for MobileNets" [26] explores the problem of applying quantisation to MobileNets. They found that the separable convolutions which are core to reducing network footprint, are also vulnerable to significant performance loss when quantisation is used. Their initial results, shown in Table 3.1 show that architectures such as InceptionV3 were found to typically loose around 2% Top-1 accuracy when switched to an 8-bit pipeline, whereas MobileNets lost almost 69%.

Table 3.1: Top-1 accuracy on ImageNet2012 validation dataset

Networks	Float Pipeline	8-bit Pipeline	Comments
InceptionV3	78.00%	76.92%	Only standard convolution
MobileNetV1	70.50%	1.80%	Mainly separable convolution

They identified that batch normalisation and ReLU6 as the root cause of the problem, and experimented with altered depthwise convolutional layers. Their 3 proposed designs achieved comparable float pipeline accuracy to the original architecture, and on the order of 2 to 8% accuracy loss with a quantised pipeline. A significant improvement.

Also in this space, [11] proposed a quantised training framework, which though training with floating point weights, simulates the rounding errors introduced by quantisation to improve network accuracy when "real" quantisation is applied after training.

3.2.2 System level

Concerted research effort for efficient neural network inference on mobile devices is still in its early stages, as it is only in the past few years that hardware has become powerful enough, and models useful enough. As a result, many tools to facilitate this endeavour are still in their infancy. An investigation in [29] 2017 found that for simpler models on a SoC, both performance and development time were better if the inference system was built from scratch, rather than battle with the compatibility issues

of existing frameworks. However, this may not continue to be the case as the ecosystem matures.

The Android Neural Networks API [20] provides implementations of standard machine learning operations, that can be run efficiently on mobile devices tuned which support it. This can be utilised by higher level frameworks (such as TensorFlow Lite [10] and Caffe2 [4]). This means that despite the API being constrained to the Android OS, networks trained on other systems will still be easily compatible.

Similarly, the ARM Computer Vision and Machine Learning Library ("ComputeLibrary", "ACL") [18] provides a set of optimised functions for ARM CPUs and GPUs, with support for Linux and Android. [29] used the library to build a Squeezenet [9] inference engine from scratch, which ran on a quad-core ARM Cortex-A7. This outperformed a TensorFlow version without ACL functions.

3.3 The Mali Architecture

Systems such as [17] have explored deep learning on the Mali architecture. With an implementation of the VGG-16 network, they found compared to a serialised baseline, an optimised OpenCL model that used vector data types and SIMD gave them a speedup of 9.4X. Another model, using the CLBlast library, with tunings optimised for the Mali-T628 GPU saw a speedup of 6.2X.

The deep learning framework CNNDroid [23] for the Android mobile operating system, utilises the RenderScript API to accelerate the execution of CNNs. The inaugural paper reports a speed-up of over 60X for AlexNet on the Samsung Galaxy Note 7 with Mali-T760 GPU compared to CPU only inference.

Chapter 4

Evaluation of MobileNet

Our investigation looked at how to run a forward pass of MobileNet on the ODROID-XU4 board, and then exploit the particulars of the environment architecture to best improve performance.

Initially we explored building TensorFlow for the ODROID-XU4, making alterations to the math backend to add OpenCL support, and if possible compare to the SYCL implementation. This endeavour proved to be unsuccessful, as discussed in 4.1.

We used a modified version of a fork of Darknet, Darknet-MobileNet, which added definitions for depthwise separable convolutional layers, which are a key feature of the MobileNet architecture. Our system adds support to build with either raw OpenCL kernels, the clBLAS library, the CLBlast library, or the original CPU BLAS routines. We also explored vectorised kernels, which in their most basic form would multiply two matrices with Row-Major and Column-Major layouts respectively. However due to time constraints we were unable to fully realise this functionality.

For the sake of valid baselines, we attempted to convert the original MobileNet model into Darknet's format. In 4.1 we discuss our approach in attempting to do this. Our final system used a model which has a architecture MobileNet, but with weights produced by training using Darknet. Additionally, due to Darknet using 64 bit datatypes, our alterations to permit execution on the 32 bit ODROID mean that the system does not produce meaningful labels. Given the focus of this work is improving inference time, rather than accuracy, this is not a problem.

The scope for potential improvements is large, and our intent was to proceed in the following manner:

- Add scalar matrix multiplication routines for Naive OpenCL (basic OpenCL kernels without optimisations), clBLAS, and CLBlast.
- Add vectorised matrix multiplication routines for Naive OpenCL, clBLAS, and CLBlast.
- Explore alternative memory layouts for vectorised matrix multiplication.
- Vary work group sizes.

- Reshape the stored weight files to match the best memory layout.
- Parallelise other network operations such as activation functions, and im2col.

4.1 Difficulties encountered

TensorFlow binaries are distributed for 64 bit systems, and the design of TensorFlow is focused on such. Compiling from source a 32 bit version is possible in theory, however we found this process to be non-trivial. TensorFlow build-scripts are intended for use with Bazel [3], a tool similar to Make [5], and we found that altering multiple core aspects of this, as well of some other TensorFlow dependencies would be necessary for a 32 bit build.

As discussed in 2.2, the original implementation of MobileNet was created using TensorFlow. Models produced by most deep learning frameworks, including TensorFlow, are stored on disk in a format specific to said framework. A sensible approach would be to either alter TensorFlow to utilise OpenCL devices, or utilise a tool which parses TensorFlow models into a format that can be understood by another system. The former approach might be expedited by using SYCL (as discussed in 3.1.1), or libraries such as the ARM ComputeLibrary (discussed in 3.2.2).

4.1.1 Model translation

Having found the TensorFlow approach to be infeasible, we looked at building our system using a simpler framework. We settled on Darknet, however to consider our accuracy scores to be valid, we desired to convert the MobileNet TensorFlow model into a Darknet one. There is currently no tool which allows the direct conversion between the two. However, we found a tool which could convert from PyTorch to Darknet. Coupled with another tool which converted from TensorFlow to PyTorch, we hoped to produce a valid model to run with our experiments. However, despite successfully converting from TensorFlow to PyTorch, the second step of the process failed.

With many deep learning frameworks still in their infancy, tools to convert between model formats exist; however, they are not widespread or fully developed. One cross-framework solution we explored was MMdnn, an MIT licensed tool-suite produced by Microsoft Corporation [19], which describes an intermediate representation (IR) that models can be translated into and from.

The use of an IR makes adding a new format to the tools suite an $O(1)$ problem rather than an $O(N^2)$ one (where N is the number of model formats), which would be defining a direct translation between each format.

4.1.2 Darknet

When implementing Darknet on the board, it was found that due to the ARM CPUs using a 32 bit instruction set, the vanilla implementation of Darknet would not work due it using multiple 64bit variable types such as `double`. Changing these to `floats`, we still encountered issues. We found that when Darknet loads a model, first it loads the structure, then the weights. While loading the structure, it sets some temporary weights using an initialisation scheme. This was not memory safe in a 32 bit system, and since we were not training, we removed this feature. However, a segmentation fault still occurred when loading the weights for a later convolutional layer. We changed all variables to suitable types, and found inference was possible. However, on the board the outputs from the network are incorrect, leading to no discernible classification of the input. Therefore, though we can compare the speedup for different operations in Darknet, we cannot make any observations of accuracy trade-offs.

4.2 Experimental setup

Our system gives the option of 4 GEMM routines for the convolutional layers: single thread CPU, a minimal OpenCL kernel, the clBLAS `clblasSgemm` operation, and the CLBlast `CLBlastSgemm` operation. We were unable to use the same GEMM routines for the depthwise separable convolutional layers, as the system would invariably throw a segmentation fault, though our investigation was unable to determine why this would be the case.

The configuration was set using CMake variables at compile time, with corresponding `#ifdef` flags in source ensuring only selected functions were compiled. A listing of the main CMake scipt can be found in **??**

To compare our systems, we investigated inference time for a single image, across individual layers and the network as a whole. We would expect the execution time to be deterministic for each system, regardless of input. Since our environment is a full Linux distribution with a myriad of other processes vying for resources, we repeated experiments to reduce noise. For our experiments, we selected 100 random CIFAR images, and performed inference on each separately using each of the 4 systems. We then took the average time to use in our evaluation. We found the variance to be negligible between runs of the same system, and thus do not include it.

4.3 Initial Results

The total inference time for a single image using a single CPU thread was around 10sec. Table 4.1 shows the relative speedup using each method.

Given our system only varies the functioning of convolutional layers, in the below Figure 4.1 only the relative inference time for these layers is included.

	Linear CPU	Naive OpenCL	clBLAS	CLBlast
Speedup	1X	2.95X	1.46X	1.37X

Table 4.1: Total inference time speedup on MobileNet

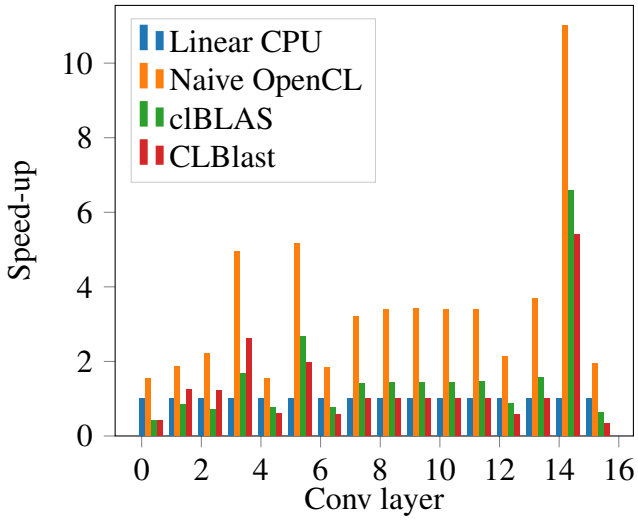

Figure 4.1: MobileNet relative inference time speedup by layer (convolutional layers only)

Note that our Naive OpenCL implementation consistently performed significantly better than all other approaches. Additionally, for some layers both clBLAS and CLBlast perform worse than the Linear CPU version. This result is surprising, since we would expect the optimisations of clBLAS to be better than our basic kernel, and for a tuned CLBlast to outperform both clBLAS and NaiveCL.

4.4 Further Investigation

The poor performance of CLBlast relative to our Naive OpenCL approach could not be reasonably explained. Thus to better explore the problem we isolated the GEMM components of our system into a minimum working example (MWE) without the complexity of the rest of the inference system. We verified that the results of each operation were correct, and timed inference on square matrices of varying size, with randomised entries. The runtimes are shown in Table 4.2.

These results seem to match our those of our full system, with our Naive OpenCL kernel performing significantly better than all other approaches, once the order of the test matrices is sufficiently large (over 200). The speedups of clBLAS and CLBlast become evident on matrices of order larger than 400.

A sensible explanation why CLBlast performs worse than expected, despite coming with well tuned settings for the Mali-T628 GPU, is that the library is not actually using these tuned values. However, we repeatedly ensured that the library was built with

dim	Linear CPU	Naive OpenCL	clBLAS	CLBlast
10	0.000007	0.065508	0.031259	1.026603
50	0.000278	0.020033	0.104300	0.665272
100	0.001897	0.020312	0.102974	0.665380
200	0.014461	0.023221	0.154625	0.268885
300	0.048645	0.028167	0.109440	0.292907
400	0.115614	0.034357	0.172429	0.300971
500	0.236099	0.042376	0.216086	0.310650
1000	2.616251	0.111629	0.330880	0.371635
1500	6.542013	0.206589	1.050376	0.551784

Table 4.2: Inference time (sec) using different GEMM routines, of square matrices of size dim

its tuning components enabled. It is possible that tuners did not recognise our device correctly, however we were unable to verify this.

Chapter 5

Conclusion

It is clear there is a high barrier of entry to developing for heterogeneous systems. Despite the ongoing efforts of hardware manufacturers, developers, and protocol standardisation bodies, having one or two basic assumptions about a runtime environment being violated can cause common software to malfunction or fail to compile. It is unrealistic to expect all but the most community rich and large scale projects to attempt to support all quirks, however this makes working with heterogeneous systems for deep learning models challenging.

We found an unreasonable amount of our investigation time was spent on trivial matters such as finding appropriate drivers for the Mali GPU, which would often stop working for esoteric reasons. Once we established our build system, complete with CMake scripts to find the correct libraries, this became less of an issue, though occasionally libraries would stop working for unknown reasons, and had to be rebuilt. Additionally, on when tested on other machines, the CMake scripts would not always find libraries which were known to be present. Though this was later found to be assumptions implicit in the scripts about where libraries and headers were likely to be.

Finding a deep learning framework to suit our needs caused us issues. TensorFlow made sense, from the perspective of it being the native environment for MobileNet, having extensive documentation (including on its math backend - Eigen), though coming with many features extraneous to our needs. However, the build process came with too many caveats, and with our time being finite we had to find something else. Darknet seemed to be appropriate, given that it was written in C, and was bereft of a great deal of unnecessary bloat that something like TensorFlow might have. However, Darknet is poorly documented, and many of its implementation details are unintuitive. Additionally, though ideal for efficiency potential, C lacks many useful language abstractions which speed up development time. Losing the ability to get meaningful inference results arguable defeats the purpose of the exercise, and in future work, we will look to move to languages such as C++ and Python, likely abandoning the Darknet framework.

Our results demonstrated the expect outcome that utilising OpenCL in linear algebra operations can speed up the inference time of a neural network. However, our best

result was an improvement of 2.95X, using a minimal OpenCL kernel. Optimised libraries such as CLBlast and clBLAS performed worse than this approach, which was surprising. Similar research such as [17] found that CLBlast provided much better performance improvements on the Mali-T628 than our experiments were able to achieve.

5.1 Future Work

The chief issue of developing for heterogeneous systems is the variety of the particulars of their given environments, which often violates some of the design assumptions of common pieces of software.

What approaches are there, specifically for Deep Learning? One promising avenue is a greater number of model conversion tools. If a deep learning framework that a model was trained on does not work in the environment it is to see deployment on, one might be able to convert it into a framework which is supported by the environment. For pre-trained inference-only systems, it may be desirable to strip the model down to its bear essentials. Automating this process - that is translating a model from its complex training environment into a efficient standalone inference system could be an area of future research. Perhaps the overhead of efficiency focused systems such as TensorFlow Lite is small enough as to be negligible, however this is yet to be adequately demonstrated.

The growing suite of libraries focused on providing environment-tuned implementations of common operations, from the lower level of CLBlast, to those higher up the abstraction hierarchy such as ComputeLibrary or Android Neural Networks API are also a promising area for future research. By proving a consistent API for Deep Learning framework developers, and putting the onus of optimisation on those specialised in such, the portability of frameworks could improve dramatically.

The current system is unable to perform valid inference on input images, though by using basic OpenCL kernels has demonstrated an at best speedup of 2.95X for total inference time. However we believe that the scope for improvement is significant.

Taking what we have learned about navigating the OpenCL stack. We believe that though necessary given our initial time-constraints and environment issues, the technical debt introduced by using Darknet, and the reduced convenience of C mean that we will be rebuilding our system from the ground-up using C++, with OpenCL. This will allow us to move beyond matrix multiplication, and better implement improvements for costly convolutional layer operations.

We will seek to explain the unexpected performance issues of clBLAS and CLBlast, as well as introduce high level machine learning operation libraries such as the ARM ComputeLibrary.

If possible, we will also investigate TensorFlow Lite, TensorFlow, and TensorFlow with SYCL support to benchmark how large the overheads of a full deep learning framework is, even with accelerators.

Both OpenCL and Linux come with a large suite of profiling tools, which could be used to further investigate performance bottlenecks.

Bibliography

[1] Martín Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: (Mar. 14, 2016). arXiv: 1603.04467 [cs]. URL: http://arxiv.org/abs/1603.04467 (visited on 03/07/2018).

[2] *An In-Depth Look at Google's First Tensor Processing Unit (TPU) | Google Cloud Big Data and Machine Learning Blog*. URL: https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu (visited on 03/08/2018).

[3] *Bazel - a Fast, Scalable, Multi-Language and Extensible Build System*. URL: https://bazel.build/ (visited on 03/08/2018).

[4] *Caffe2*. URL: http://caffe2.ai/ (visited on 04/06/2018).

[5] *Gnu.Org*. URL: https://www.gnu.org/software/make/ (visited on 03/08/2018).

[6] Gaël Guennebaud, Benoît Jacob, and others. "Eigen V3". In: (2010). URL: http://eigen.tuxfamily.org.

[7] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: (Apr. 16, 2017). arXiv: 1704.04861 [cs]. URL: http://arxiv.org/abs/1704.04861 (visited on 03/07/2018).

[8] Itay Hubara et al. "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: (Sept. 22, 2016). arXiv: 1609.07061 [cs]. URL: http://arxiv.org/abs/1609.07061 (visited on 03/09/2018).

[9] Forrest N. Iandola et al. "SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and <0.5MB Model Size". In: (Feb. 23, 2016). arXiv: 1602.07360 [cs]. URL: http://arxiv.org/abs/1602.07360 (visited on 04/07/2018).

[10] *Introduction to TensorFlow Lite*. URL: https://www.tensorflow.org/mobile/tflite/ (visited on 04/06/2018).

[11] Benoit Jacob et al. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: (Dec. 15, 2017). arXiv: 1712.05877 [cs, stat]. URL: http://arxiv.org/abs/1712.05877 (visited on 04/16/2018).

[12] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246. URL: http://doi.acm.org/10.1145/3079856.3080246 (visited on 03/08/2018).

[13] Alex Krizhevsky. "Learning Multiple Layers of Features from Tiny Images". In: (), p. 60.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Communications of the ACM* 60.6 (2012), pp. 84–90. ISSN: 00010782. DOI: 10.1145/3065386. URL: http://dl.acm.org/citation.cfm?doid=3098997.3065386 (visited on 04/06/2018).

[15] Griffin Lacey, Graham W. Taylor, and Shawki Areibi. "Deep Learning on FPGAs: Past, Present, and Future". In: (Feb. 12, 2016). arXiv: 1602.04283 [cs, stat]. URL: http://arxiv.org/abs/1602.04283 (visited on 04/06/2018).

[16] E. Scott Larsen and David McAllister. "Fast Matrix Multiplies Using Graphics Hardware". In: *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. SC '01. New York, NY, USA: ACM, 2001, pp. 55–55. ISBN: 978-1-58113-293-9. DOI: 10.1145/582034.582089. URL: http://doi.acm.org/10.1145/582034.582089 (visited on 03/08/2018).

[17] Manolis Loukadakis. "Accelerating Deep Neural Networks on Low Power Heterogeneous Architectures". In: (), p. 14.

[18] Arm Ltd. *Technologies | Compute Library*. URL: https://developer.arm.com/technologies/compute-library (visited on 04/07/2018).

[19] *MMdnn Is a Set of Tools to Help Users Inter-Operate among Different Deep Learning Frameworks. E.g. Model Conversion and Visualization. Convert Models between Caffe, Keras, MXNet, Tensorflow, CNTK, ..* URL: https://github.com/Microsoft/MMdnn (visited on 03/08/2018).

[20] *Neural Networks API | Android Developers*. URL: https://developer.android.com/ndk/guides/neuralnetworks/index.html (visited on 04/06/2018).

[21] Cedric Nugteren. "CLBlast: A Tuned OpenCL BLAS Library". In: (May 12, 2017). arXiv: 1705.05249 [cs]. URL: http://arxiv.org/abs/1705.05249 (visited on 03/31/2018).

[22] *ODROID | Hardkernel*. URL: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825 (visited on 03/07/2018).

[23] Seyyed Salar Latifi Oskouei et al. "CNNdroid: GPU-Accelerated Execution of Trained Deep Convolutional Neural Networks on Android". In: (2016), pp. 1201–1205. DOI: 10.1145/2964284.2973801. arXiv: 1511.07376 [cs]. URL: http://arxiv.org/abs/1511.07376 (visited on 04/06/2018).

[24] *Processor*. URL: https://www.sophon.ai/product/sc1.html#processor (visited on 03/08/2018).

[25] Joseph Redmon. *Darknet: Open Source Neural Networks in C*. http://pjreddie.com/darknet/. 2013–2016.

[26] Tao Sheng et al. "A Quantization-Friendly Separable Convolution for MobileNets". In: (Mar. 22, 2018). arXiv: 1803.08607 [cs]. URL: http://arxiv.org/abs/1803.08607 (visited on 04/16/2018).

[27] Laurent Sifre. "Rigid-Motion Scattering for Image Classification". Ecole Polytechnique, CMAP, 2014. URL: http://www.cmapx.polytechnique.fr/~sifre/research/phd_sifre.pdf.

[28] P. Y. Simard, D. Steinkrau, and I. Buck. "Using GPUs for Machine Learning Algorithms". In: *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)(ICDAR)*. Aug. 2005, pp. 1115–1119. ISBN: 978-0-

7695-2420-7. DOI: 10.1109/ICDAR.2005.251. URL: doi.ieeecomputersociety.org/10.1109/ICDAR.2005.251 (visited on 03/10/2018).

[29] Dawei Sun, Shaoshan Liu, and Jean-Luc Gaudiot. "Enabling Embedded Inference Engine with ARM Compute Library: A Case Study". In: (Apr. 12, 2017). arXiv: 1704.03751 [cs]. URL: http://arxiv.org/abs/1704.03751 (visited on 04/07/2018).

[30] *Tensorflow: Computation Using Data Flow Graphs for Scalable Machine Learning*. URL: https://github.com/tensorflow/tensorflow (visited on 03/08/2018).

[31] *The Khronos Group*. URL: https://www.khronos.org/members/list (visited on 03/07/2018).

[32] Fabian Tschopp. "Efficient Convolutional Neural Networks for Pixelwise Classification on Heterogeneous Hardware Systems". In: (Sept. 10, 2015). arXiv: 1509.03371 [cs]. URL: http://arxiv.org/abs/1509.03371 (visited on 04/07/2018).

Appendices

Appendix A

Glossary

- **BLAS**: Basic Linear Algebra Subprograms. Includes efficient implementations of operations such as matrix multiplication.
- **GEMM**: General Matrix Multiplication
- **CNN**: Convolutional Neural Network. A class of neural network which includes convolutional layers, which have proved effective in the area of image recognition.
- **Forward pass**: The operation in a neural network where one gives the system input data and computes on it an output. An example of this would be giving a label to an unlabelled image.

Appendix B

CMake script

code:cmake CMake scipt to build Darknet with OpenCL support, if desired. Finds and links libraries automatically.

```
cmake_minimum_required (VERSION 3.5)
project (darknet_kernels)

set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Ofast")

# Select directories of code and headers
include_directories(include)
include_directories(src)
include_directories(examples)
set (CMAKE_MODULE_PATH
  "${CMAKE_CURRENT_SOURCE_DIR}/cmake")

file(GLOB SRC_SOURCES "src/*.c")
file(GLOB BENCHMARK_SOURCES "benchmark/*.c")
file(GLOB EXAMPLES_SOURCES "examples/*.c")

add_executable(darknet
  ${BENCHMARK_SOURCES}
  ${SRC_SOURCES}
  ${EXAMPLES_SOURCES})

# Find and link libraries
target_link_libraries(darknet m pthread)

if (CLBLAS)
  find_package(clBLAS)
  if (CLBLAS_FOUND)
    message(STATUS "clBLAS library found!")
    target_link_libraries(darknet ${CLBLAS_LIBRARIES})
```

```
    include_directories(${CLBLAS_INCLUDE_DIRS})
    add_definitions(-DCLBLAS)
  else()
    message(SEND_ERROR "clBLAS library not found")
  endif()
endif()

if (CLBLAST)
  find_package(CLBlast)
  if (CLBLAST_FOUND)
    message(STATUS "clBLAST LIBRARY FOUND!")
    target_link_libraries(darknet ${CLBLAST_LIBRARIES})
    include_directories(${CLBLAST_INCLUDE_DIRS})
    add_definitions(-DCLBLAST)
  else()
    message(SEND_ERROR "clBLAST library not found")
  endif()
endif()

if (NAIVE_CL OR CLBLAS OR CLBLAST)
  find_package(OpenCL)
  if (OpenCL_FOUND)
    message(STATUS "OpenCL library found!")
    target_link_libraries(darknet ${OpenCL_LIBRARY})
    include_directories(${OpenCL_INCLUDE_DIRS})
    add_definitions(-DOPENCL)
    if (NAIVE_CL)
      add_definitions(-DNAIVE_CL)
      # Configure OpenCL kernels' location for runtime
      file(COPY "src/"
        DESTINATION "${CMAKE_BINARY_DIR}/kernels"
        FILES_MATCHING PATTERN "*.cl")
    endif()
  else()
    message(SEND_ERROR "OpenCL library not found")
  endif()
endif()

if (BENCHMARK)
  file(MAKE_DIRECTORY "${CMAKE_BINARY_DIR}/logs")
  # Add definitions
  add_definitions(-DBENCHMARK)
endif()

install(TARGETS darknet DESTINATION bin)
```