Evaluating Neural Networks on Low Powered GPUs.

Simon Rovder

Master of Informatics

School of Informatics
University of Edinburgh

2017

Abstract

This report presents optimisation techniques for improving neural network execution performance on low-powered GPUs. Specifically targeting the Odroid-XU3 board, the report demonstrates the performance of variously optimised OpenCL kernels for forward propagating inputs over a fully-connected neural network using the Mali T-628 GPU present on the Odroid board.

We write a framework for testing general-purpose OpenCL kernels on GPUs, on top of which the neural network execution system is subsequently built.

Considered optimisation techniques include altering memory layouts, managing work-group sizes, vectorisation, and cache blocking. We reproduce the results of the best performing matrix multiplication kernel published by ARM for this GPU, and improve its performance by 12% using Hybrid Morton Order memory layouts.

The final system is benchmarked against TensorFlow on a desktop GPU as well as Numpy on the CPU of the Odroid Board. In both cases the system demonstrates promissing results, matching TensorFlow in performance on small networks and greatly outperforming Numpy for any network.

Acknowledgements

I would like to thank Michael O'Boyle for his support and guidance over the course of this project.

Further thanks goes to James Renwick for always gladly providing insight into the intricacies of C++.

I would also like to thank Constance Crowe and Paul Sinclair for proofreading my final report draft.

Table of Contents

Chapter 1

Introduction

Neural networks have repeatedly proved their use in the field of Machine Learning. Being adaptable to a large variety of problems, neural networks have yielded state of the art results in pattern recognition and classification tasks such as image recognition [1] and speech processing [2]. For certain problems, neural networks have even been trained to outperform professional human predictions [3].

Until recently, the computational complexity of training and executing neural networks posed a significant problem to their wide-spread use. However, with the progressing performance improvements of Graphics Processing Units (GPUs), technology has caught up with the computational demands of neural networks and we are beginning to be able to effectively harness their potential.

In recent years GPUs made their way into the hand-held devices market of spartphones and tablets [1]. These low powered GPUs open doors for making use of neural networks on these devices.

1.1 Goal

The goal of this work is to create a simple system for executing neural networks on low powered GPUs. The target device of this research is the Mali T-628 GPU on the Odroid XU-3 board. We shall write an OpenCL implementation of the basic essential operations needed to execute fully-connected neural networks and we tailor these kernels to exploit the architectural hardware optimisations present on this particular device.

The most computationally expensive task we tackle is matrix multiplication. Our goal is to reproduce the performance of the best performing Mali T-628 matrix multiplication kernel published by ARM and improve upon it. This goal is successfully accomplished in Chapter 8, where we use Hybrid Morton Order memory layouts to outperform the ARM implementation performance by 12%.

[1] A concise list of smartphone and tablet GPUs has been compiled by Klaus Hinum[4]

1.2 Motivation

As mentioned previously, low powered GPUs have become popular in the field of hand-held devices. There is, however, a notable lack of research in the area of optimising neural network execution on these devices. Most research today focuses on the issue of efficient training of neural networks, which is a much more expensive endeavor, and as such regular desktop GPUs are the preferred tool for this task.

One of the forefront frameworks for executing neural networks of GPUs is TensorFlow. TensorFlow, however, works on the CUDA architecture, meaning it cannot execute on the Mali T-628 GPU. Given the popularity of TensorFlow, it is understandable that a similar demand exists for OpenCL platforms. The lack of OpenCL support is a long standing issue with TensorFlow [5], as well as other similar frameworks further discussed in Section 3.2.

This work hence aims to fill this gap in the field of GPU machine learning research.

Chapter 2

Background

2.1 Neural Networks

The concept of today's neural networks dates back to the 1950s and the invention of the perceptron. This section gives a brief introduction to the topic and describes the principles behind a simple fully-connected neural network.

2.1.1 The Perceptron

The fundamental building block of neural networks is the perceptron. The perceptron is a computational representation of a multidimensional binary classifier that mimics the functioning of a biological neuron. It takes a set of values as inputs (much like a neuron does via dendrites) and outputs a single value (much like a neuron does via the axon).

Suppose we had a perceptron that processes N dimensional input vectors with entries $\alpha_1 ... \alpha_N$. In addition to this, the perceptron will add an $a_0 = 1$ entry to this vector (also known as the *bias* term). For each a_i, the perceptron will have a *weight* term w_i. The perceptron will then calculate its output r as seen in Equation 2.1. A visualisation of a perceptron can be seen in Figure 2.1.

$$r = f\Big(\sum_{i=0}^{N} w_i a_i\Big)$$

where

$$f(x) = \begin{cases} 1 & x \geq 0 \\ 0 & otherwise \end{cases}$$

Equation 2.1: Perceptron Computation

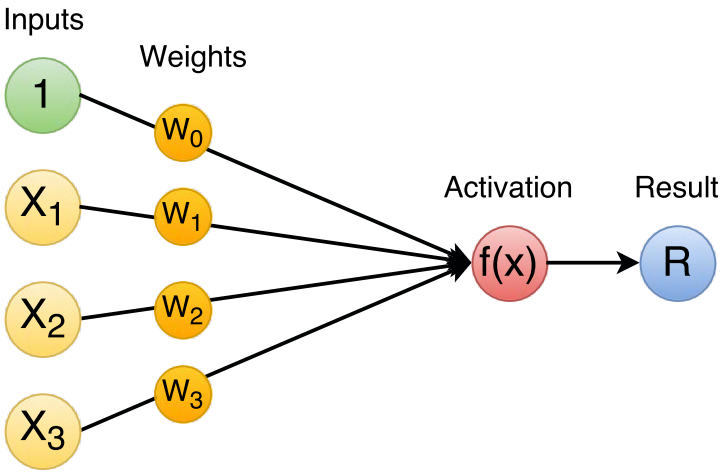

Figure 2.1: Perceptron Diagram - Capable of classifying linearly separable data.

The resulting binary classification achieved by a perceptron splits the input space between the two classes. An example of such a split of a two dimensional input space can be seen in Figure 2.2. Note that this boundary is linear. A perceptron is only capable of splitting the input space into two regions linearly, where the boundary is decided by the weights.

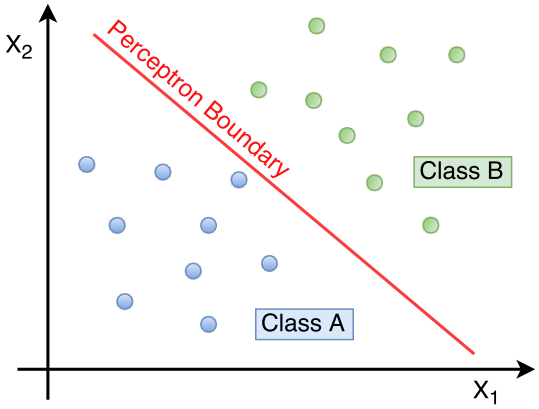

Figure 2.2: Perceptron Classification

2.1.2 Fully-Connected Layers

The output of a single perceptron conveys a single piece of information about the input feature vector. Complex classification problems often require more information about the input vector, which is why perceptron are rarely used individually. The common approach to extracting more information from an input vector is to pass it to multiple perceptron, collecting the outputs into an output vector. Suppose we had an n-dimensional input vector V and an m dimensional vector R. The individual values R_i are computed as follows:

$$R_i = f\Big(\sum_{k=0}^{n}\big(V_k w_{ik}\big)\Big)$$

In this formula, w_{ij} is the weight connecting V_i to R_j, which means we are effectively linearly transforming V by computing $W \times V$ for a matrix W where $W_{ij} = w_{ji}$. In addition, the function f may be replaced with any other activation function (popular choices including Sigmoid and ReLU functions), to introduce non-linearity.

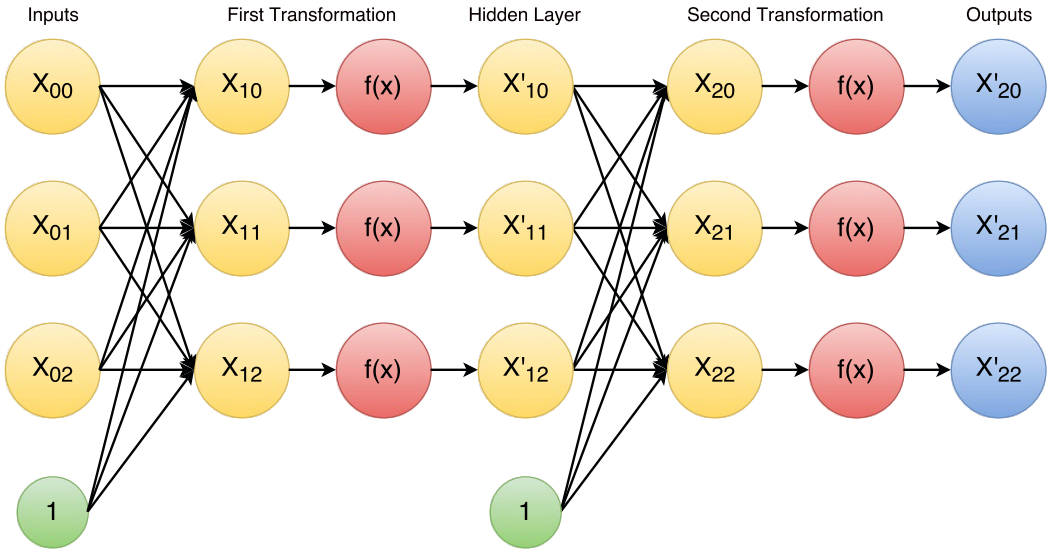

Figure 2.3: Diagram of a deep neural network - Capable of classifying non-linearly separable data.

2.1.3 Deep Neural Networks

The issue with the perceptron layer approach from Section 2.1.2 is that, despite the model having greater classification power than a single perceptron, it still cannot classify non-linearly separable classes. This is solved by chaining multiple perceptron layers sequentially into what we call **deep neural networks**, which enables the classifier to separate data which is not necessarily linearly separable. This process is called **feature extraction**, as the first layer of perceptrons extract **features** from the input vector, such that these higher-level features allow linear separability of classes, or convey less featurally entangled classes (closer to linearly separable).

Due to this ability to perform feature extraction and learn which features to extract, a deep neural network is superior to a single layer network, allowing it to be applied to more complex problems. A visual representation of a deep neural network can be seen in Figure 2.3.

Given propagating an input vector over a single layer requires matrix multiplication, and multiple layers connected sequentially, it is understandable that executing neural networks is a highly computationally demanding task. This is especially true for high dimensional data that may have dimensionality in the order of thousands (for example for images).

It is for this reason that GPUs are often utilized to tackle the most computationally intensive operations of this process.

2.2 Graphic Processing Units (GPUs)

Primarily motivated by the growth of the video game industry, GPUs have steadily improved and became integral parts of many audio-visual devices, which, in recent years, also includes hand-held devices like smartphones. Although GPUs were primarily designed with the purpose of graphics processing, their computational power can be employed for any data-based parallelisable task.

There are two primary types of parallelism: **task-based** parallelism and **data-based** parallelism. Task-based parallelism is the kind we are most familiar with from interacting with computers in every day life: having multiple different programs running at the concurrently, each working on achieving its own goal. Data-based parallelism is the type of parallelism used by GPUs to process data: having a single program run many times with each instance processing different data (or different sections of the same data), for the most part with the intention of eventually combining their partial results into one final result.

Data-based parallelism is highly effective at executing tasks which require a large amount of number crunching, which is why GPUs are a great candidate device for executing neural networks.

2.3 OpenCL

OpenCL [6] is a framework for writing and executing code in parallel across multiple heterogeneous platforms and we shall make use of it in this work to execute programs on the Mali GPU. The OpenCL programming paradigm focuses on splitting a program into **host-code** and **device-code**.

Host-code is the part of the program intended to run on the CPU, and usually comprises of the sequential infrstructural code. It can be written in any language, provided OpenCL bindings are available for this language. [1]

Device-code is the part of the program intended to run in parallel on the GPU. OpenCL defines its own programming language for device-code (which is closely based on C-99) and is compiled by OpenCL at runtime for individual devices. This OpenCL code can be stored in strings or in separate files.

[1] Bindings have been written for many programming languages, most popular of which include Java [7], Python [8] and Haskell [9]. In our work we use C++ bindings [10].

2.3.1 Terminology

This section introduces all terminology related to OpenCL.

OpenCL Device – A hardware component capable of executing OpenCL code. An example is the Mali T-628 GPU itself. Other examples are regular graphics cards in computers (such as NVIDIA or RADEON cards), integrated graphics devices, coprocessors, even CPUs themselves are also usually compliant with OpenCL. An OpenCL Device will contain one or more *Compute Units*.

Compute Unit (or *Core*) – A piece of hardware capable of executing a *Kernel* on a *Workgroup*. An OpenCL Device will usually have one or more of these.

Kernel – A single function from the device-code. Host-code will run the Kernel in parallel on the OpenCL device.

Workgroup – A set of *Work Items*, small enough to execute on a single compute unit.

Work Item – A particular instantiation of a kernel. When executing, every kernel has its own set of *ids* (this is what we mean by its *instantiation*). The kernel accesses these ids via function calls and once it knows their values, the kernel knows what it should do.

NDRange – The term for the range of ids a kernel is mapped onto. (Stands for N-Dimensional Range).

2.3.2 OpenCL Paradigm

In order to understand how and when OpenCL can be used, one must first understand the principle behind its functionality. What OpenCL achieves is best demonstrated on an example.

Suppose we wanted to add two 2-dimensional matrices together. In C++ we could write a program for this task as seen in Figure 2.4.

```
1 void addMatrices(Matrix& a, Matrix& b, Matrix& r){
2   for(size_t row = 0; row < a.rows; row++) {
3     for(size_t col = 0; col < a.cols; col++) {
4       r[row][col] = a[row][col] + b[row][col];
5     }
6   }
7 }
```

Figure 2.4: Example unparallelised C++ matrix addition algorithm

The code in Figure 2.4 will successfully execute the task, however, it will do so sequentially in a single thread on the CPU (computing one sum at a time). However, if we think about this code in terms of utility, we observe that the only line in this implementation directly contributing to the computation of our desired result is line 4. The `for` loops are there only as boilerplate; they ensure that line 4 gets called for all combinations of `row` and `col`.

Hypothetically, if one was able to execute line 4 from Figure 2.4 in parallel for *all* combinations of `row` and `col` at the same time, the same result would be achieved. In other words we want something resembling code in Figure 2.5.

```
1 parallel void addMatrices(Matrix& a, Matrix& b, Matrix& r){
2   size_t row = whichRow();
3   size_t col = whichCol();
4   r[row][col] = a[row][col] + b[row][col];
5 }
```

Figure 2.5: Pseudocode representation of a parallel matrix addition program

The code in Figure 2.5 is conceptually very close to what OpenCL does. OpenCL allows us to map a function onto a space of **id**s [2], which are then accessible from the code via function calls. For our example these ids would be the combinations of rows and columns. A full OpenCL implementation of this algorithm (for reference and contrasting) can be seen in Figure 2.6

```
1  __kernel void addMatrices(
2        size_t cols,
3        const float * a,
4        const float * b,
5        float * r )
6  {
7      size_t row = get_global_id(0);
8      size_t col = get_global_id(1);
9      r[row*cols + col] =
10       a[row*cols + col] + b[row*cols + col];
11 }
```

Figure 2.6: OpenCL implementation of a parallel matrix addition program

[2] The space of ids can be one, two or three dimensional. In the case of matrix addition, we would choose a two-dimensional space.

	Device 1	**Device 2**
Memory (bytes)		
Global	2 091 724 800	2 091 724 800
Cache size	131 072	131 072
Cacheline size	64	64
Cachelines	2048	2048
Local	32768	32768
Vector Sizes		
Char	16	16
Short	8	8
Int	4	4
Long	2	2
Float	4	4
Double	2	2
Computation		
Clock	600 Mhz	600 Mhz
Compute units	4	2
Max work group size	256	256
Work item sizes	256, 256, 256	256, 256, 256

Table 2.1: Specifications of the two OpenCL devices available on the MALI T-628 GPU.

2.4 The Mali-T628 GPU on the Odroid XU-3 board

In this project we will be optimising neural network execution for one specific device - the **Odroid-XU3** board. This board contains one physical Mali T-628 GPU, although its compute units are accessible via two separate devices.

To properly optimise algorithms for the GPU, we need to know what hardware restrictions it imposes. Since the specifications may vary between the two accessible devices, in order to have reliable knowledge of the device specifications, we need to investigate both devices ourselves. The C++ bindings for OpenCL provides us with a simple interface for querying the devices for their hardware capabilities with the `getInfo` method. We have queried both the device for their specifications. These can be seen in Table 2.1.

As you can see in Table 2.1, the only difference between the two devices is the amount of compute units. The first device has twice as many compute units as the second one. This means that the first device can process four workgroups at a time, while the second one can only process two and we may need to take this into consideration when optimising for each individually.

The final relevant metric to the Mali T-628 GPU is the **GFLOPS** (amount of billions of floating point operations performed per second) limit, which is 17 GFLOPS per Compute Unit [11].

2.5 Mali GPUs compared to Desktop GPUs

Every GPU has its own internal architecture, yet they usually adhere to certain common standards, which software engineers expect and rely on when writing device code. The Mali GPU is a low-powered GPU and it architecturally differs from common desktop GPUs, meaning that many techniques used to optimise device code for common GPUs may not work (or may even have an explicitly negative impact) when used in Mali device code. Similarly, certain uncommon optimisation techniques may have a much more pronounced positive impact on the Mali than on a desktop GPU. In this section we present the key differences between the Mali and other more common desktop GPUs.

2.5.1 Local Memory

The main task of GPUs is to do a large amount of data processing and number crunching concurrently. On a GPU, prior to processing, this data will be placed into the device's **global memory**, from which the cores of the GPU can access it. This global memory may be understood as something resembling Random Access Memory (RAM) in the context of a CPU. As mentioned in Secton 2.3.1, a device may have multiple cores, each capable of executing many concurrent work items, and often parts of data used by one work item may also be required by another work item.

Desktop GPUs exploit this data reusability using *local memory*. Local memory is a hardware component for storing data and it is located directly on the chip, one within each core. Thus one may set up workgroups such that the individual work items' data requirement overlaps are maximised. This allows us to write kernels which, before performing any processing, collectively copy the required data from global memory to local memory, making only a single access to global memory per datum. Then, the kernels may proceed to process the data, accessing it in the faster local memory, mitigating the requests to global memory (and allowing other cores to access it).

The Mali GPUs do not have any on-chip local memory. One may still use local memory, however, it is mapped to the same hardware as global memory. This means local memory is not suitable for optimisations purely for improving data access speed and any such optimisations will only have a negative overhead effect. On the other hand, Mali GPUs make heavy use of caching and optimising for cache size and cache line size is desired [12].

2.5.2 Thread Divergence

In GPUs, massive parallelism is achieved by having small groups of threads run on the same program counter. This, of course, introduces problems when execution reaches an inconsistent branch instruction, where GPUs will cause threads to stall (as program counter is no longer identical). This is called *thread divergence*.

Desktop GPUs rely on the programmer to write code to avoid thread divergence. In turn, architectures like CUDA rely on such implementation and add hardware support for optimising coalesced memory accesses, which is in line with the idea of CUDA threads effectively performing vector operations on memory [13] in what we call *wavefronts* or *warps*.

The Mali GPUs do not experience any stalling effect caused by thread divergence. This is because in Mali GPUs, the smallest group of threads with the same program counter is exactly one [14]. As such, any optimisation for preventing thread divergence or for utilising coalesced memory access pattern are unnecessary and may have a negative impact on performance due to strided accesses within kernels.

2.5.3 Vector Operations

Many desktop GPUs operate on scalar types. Mali GPUs, however, have 128-bit registers and support vector operations. Making use of vector operations better utilises the Mali hardware and comes at no cost to the number of concurrent threads [3]. The vector capabilities of the Mali GPU can be seen in Table 2.1.

[3]This is of course limited by the amount of registers a kernel requires. If a kernel requires more registers than the device can provide, the amount of concurrently executing work items will be lowered as hardware gets grouped together.

Chapter 3

Related Work

In this chapter we present a series of works related to our research topic. We present the key observations made from these works and present potential issues that were noticed.

Focusing on implementing a neural network execution library means focusing on the underlying mathematical operations such as dense matrix-matrix multiplication, vector operations, convolution, pooling, normalisation, etc... Since there is no such library as the one we are implementing, most of the work is related to these underlying mathematical operations and their performance on the Mali GPUs as well as other low-powered GPUs.

3.1 Machine Learning and GPUs

As we approach the physical limits of electronics, we are observing noticeable plateauing of sequential computation performance on CPUs. Parallel platforms like GPUs, however, are consistently increasing in performance and the field of machine learning has been slowly moving towards harnessing the potential of GPUs. Individual studies into this effort for various types of machine learning models have been published, covering many of the essential classification methods such as Naive Bayes [15], K-Means Clustering [16], Decision Trees [17] and Random Forests [18].

In all these works, the performance gains from utilising GPU parallelism have been experimentally demonstrated and their measured results were very significant, motivating further research. However, despite a significant presence of research into machine learning on GPUs, there is a noticeable lack of low-power GPU implementations. In fact, all the research mentioned in this section focused on the CUDA architecture, which is radically different from that on the Mali T-628 GPU (Further discussed in Section 2.5)

3.2 Neural Network Frameworks

The idea of having a universal neural network framework is not novel and there are numerous implementations of such frameworks. TensorFlow [19], Caffe [20] or Torch [21] are notable examples, yet there are also frameworks dedicated purely to research in machine learning and deep training, such as Theano [22].

While these frameworks are very popular due to their ease of use and abstraction from the low level nature of GPU device code, they are, much like the research mentioned in Section 3.1, all primarily focused on the CUDA architecture. There is virtually no OpenCL support, despite the overwhelming demand for it from the community [5], and while there may be community-driven incentives for developing OpenCL support independently for some of them (like OpenCL Caffe [23]), these attempts are largely incomplete and (as in the case of OpenCL Caffe), experimental.

3.3 Optimising GPU kernels

Memory access patterns are the main deciding factor of how fast a GPU kernel will be, and as such, have been a target of many studies [13] [24] [25]. It is so crucial to optimise memory patterns, that APIs were designed to assist programmers in designing kernels with optimal memory access patterns [26]. Again, however, as noted in Section 3.2, most of this research focuses on CUDA architectures and is not applicable to OpenCL low-powered GPUs like the Mali T-628.

It is worth noting, however, that improvements in kernel performance in these studies have been gained by remapping elements in memory, in order to have a memory access pattern better suited for the particular hardware optimisations. In our work we shall make use of the same general optimisation techniques, yet we shall target them directly at the hardware optimisations of the Mali T-628 GPU.

Research by Anthony E. Nocentino and Philip J. Rhodes [24] into using Z-Morton memory layouts to provide faster access to individual regions of 2-dimensional data has provided the inspiration for using variants of these layouts for optimising matrix multiplication in Chapter 8 of this work.

3.4 Optimization Techniques for Mali GPUs

The performance of dense matrix multiplication and two-dimensional convolution (both crucial to neural network evaluation) on a MALI GPU were benchmarked in the 2014 pby Ivan Grasso et. al. [27]. The research entailed implementing a series of conventional algorithms for both the Cortex-A15 (sequential CPU) and the Mali T-604 (parallel GPU) and comparing the relative difference in speed and energy consumption for the implementations. The paper reports a GPU speed-up of up to a factor of 25.5 and 24 for dense matrix multiplication and two-dimensional convolution respectively.

While the results this paper presents are extremely promising for our research, the experimental setup and OpenCL code is not included, meaning the results cannot be reproduced. Furthermore, the performance comparison was done on a Mali T-604 GPU while our research focuses on the later T-628 model. These two models are from the same series and are closely related, so we may expect similar results to be achievable on the Mali T-628 GPU as well.

3.5 Mali T-600 Series Matrix Multiplication

Published by ARM, the *Optimising OpenCL kernels for the ARM Mali-T600 GPUs* [12] paper shows the improvements attainable on the single precision general matrix-matrix multiplication algorithm using a variety of kernel optimisations. These include altering memory layouts [1], vectorization, blocking and cache blocking. The paper demonstrates a 6-fold speed increase between unoptimised and optimised matrix-matrix multiplication kernels and demonstrates the significance of memory barrier overheads.

This paper is one of the few available papers stating Mali board performance metrics in GFLOPS. The research also includes the OpenCL code used to achieve the presented metrics, making the results reproducible (we reproduce the paper's final implementation in Section 7.4). The paper does not, however, specify the workgroup sizes, only presenting the best results seen out of all tried workgroup sizes (we manage to reproduce the best results by inferring optimal workgroup sizes using memory access patterns and workgroup allocation order in Section 7.2).

The single drawback of this paper is that all algorithms and statistics presented in it work with square matrices only, which does not translate well to neural networks, as our matrices may drastically vary in sizes. Once adapted to general matrix sizes, the algorithm presented in Listing 1.12 of the paper proved highly effective for simple memory layouts and is evaluated in Section 7.4 of our research.

3.6 CNNdroid

A similar system to that which we are constructing and assessing in this research was released in late 2016 under the name **CNNdroid** [28]. CNNdroid is a neural network execution open source library written in RenderScript [29] and designed for Android phones. It contains implementations for most major neural network operations, such as the dense matrix-matrix multiplication, convolution and max-pooling. As it is written for use on smartphones, this library also includes optimisations for the low-power GPUs (such as vectorisation, see Section 2.5), making it the closest work to this research.

[1] This particular paper used transposition to redefine the matrix cross product. This method is, however, exactly equivalent to altering memory layouts.

The primary issue of CNNdroid is that, as it is designed for Android phones and written in RenderScript, it requires Android and the Java virtual machine to run on the device. This may be an undesirable factor when wanting to deploy a neural network execution library to an embedded system with an attached GPU (our system shall be implemented in C++ in order to avoid this kind of overhead).

Furthermore, the optimisation steps taken in CNNdroid do not go beyond the usage of vector types. This is a problem when working with the Mali GPU, since, as we shall demonstrate in this work, techniques such as cache blocking, optimising work-group sizes, and optimising memory layouts can have a significant positive impact on performance as well, and CNNdroid does not exploit either of these.

Chapter 4

Infrastructure

This chapter aims at bridging the gap between the low-level of OpenCL with the higher level of C++, and it primarily focuses on the technical side of this project. We present the infrastructure that we built on top of the Khronos C++ OpenCL bindings.

4.1 Motivation

During this project we needed to perform an extensive amount of experimentation, benchmarking and statistics gathering for different variations of our kernels. OpenCL compiles device-code for OpenCL devices at runtime and it forces us to manage memory coherency between the host and device in our code. This poses certain complications to our experimentation process, which we shall overcome in this chapter. In order to ensure smooth functioning, our implementation has to ensure the requirements on List 4.1 are met.

The first step we took was to abstract away from the low-level programming of C, for which OpenCL headers are provided. As mentioned in Section 2.3, there exist numerous OpenCL bindings for higher level languages, so for this project, we chose to use the C++ bindings provided by the Khronos Group [10]. This brings the functionality of OpenCL into the context of object oriented programming.

While using C++ bindings for OpenCL brings us into the context of object-oriented programming, we are no closer to satisfying the requirements in List 4.1. Given all these requirements, it became apparent that we require an overarching framework around the C++ bindings, which would satisfy the above requirements.

Such a framework would allow us to focus on the optimisation alone. Writing this framework will hence be the first step we take in this work, and it will subsequently become the foundation for our neural network evaluation system. The framework was named **DynamicCL** and describing its components will be the focus of this chapter.

1. Making any changes to the device-code files requires that we re-read and re-compile the up to-date-code.

2. Changing the OpenCL device requires all device-code to be recompiled for the new device and all data to be re-synchronised with this new device.

3. Any change in the local copy of data requires these changes to be applied to the device memory as well to preserve coherency.

4. For every task we need to perform on the device, we may have multiple kernels with different implementations (for benchmarking purposes). Each of these variations may have certain requirements on the input parameters, related to the implementation. Thus changing which implementation we use may require the input argument values to be altered accordingly and synchronised with the device before the kernel is called.

List 4.1: The requirements for the DynamicCL framework

4.2 The `CLStruct`

In order to perform operations on an OpenCL device, we needs a set of objects from the C++ bindings, which wrap the lower level functions of OpenCL. These are the *Platform*, *Context*, list of *Devices* and a *Command Queue*. Within the `DynamicCL` framework we keep all this information in one structure called the `CLStruct`, as shown in Source 4.1.

```
1 typedef struct {
2   std::vector<cl::Platform> platforms;
3   std::vector<cl::Device> devices;
4   cl::Context context;
5   cl::CommandQueue queue;
6 } CLStruct;
```

Source 4.1: The CLStruct

For traversing the list of platforms and devices to find the appropriate device, we have a function called `PrepareCL`. This function finds the appropriate device and platform, creates an appropriate context and command queue for them and saves them into the `CLStruct`.

Once the `CLStruct` is created and filled, it can be used to compile device-code, allocate memory on the device, read and write to device memory, and execute kernels. In other words, it can do everything our requirements state, except, though not on its own. For this purpose we have the **CLManager**, which is described in the next section.

4.3 The CLManager

The CLManager class was designed to fulfill requirements 1, 2 and 3 from List 4.1, using an instance of CLStruct. The main public interface methods of this class are in Appendix 12.1.

The methods readLibrary and compileLibrary are related to the first requirement in List 4.1. The CLManager class keeps track of all device-code source files, programs and kernels, ensuring they are always compiled for the appropriate device. Further information on where these kernels are read from is in Section 4.4.1.

The methods createBuffer, deleteBuffer and verify allow us to manage memory, and work similarly to the C functions malloc and free, except they manage GPU memory. They are related to Requirements 3 and 4 from List 4.1. We make further use of these functions in the Mirrorable abstract class, which will be explained in Section 4.5. The CLManager class keeps track of all memory buffers created with the createBuffer method, deleting them when the deleteBuffer method is called. Whether buffers are valid can be checked via the verify method, to ensure no kernels are run on invalid GPU memory buffers.

Finally, we have the setDevice method, which allows us to select an OpenCL device by the number of cores it has. This is related to requirement 2 from List 4.1. For the purposes of our research, this is a sufficient filtration criterion, as both the OpenCL devices visible on the Odroid-XU3 board have different numbers of cores. Once a device is selected, the setDevice method invalidates all programs and buffers (if it had any) and recompiles the source files for the new device via the compileLibrary method.

4.4 The DynamicCL::Kernel wrapper

As mentioned in Section 2.3.1, kernels are compiled entry point functions, which can be mapped onto a space of ids. Kernels are wrapped in the cl::Kernel class in the Khronos C++ OpenCL bindings. We, however, wrap the cl::Kernel in a higher class, DynamicCL::Kernel, which provides us with additional simplicity in passing arguments to the kernels. To pass arguments to the kernels, we use the two functions with signatures in Source 4.2. The implementation of these functions takes care of all type conversions, argument counting, and handling of any potential errors.

```
1 // Pass a cl_uint argument to the underlying cl::Kernel
2 Kernel & nextArg(unsigned int longArg);
3
4 // Pass a cl::Buffer to the underlying cl::Kernel
5 Kernel & nextArg(cl::Buffer & buffer);
```

Source 4.2: Primary methods in the DynamicCL::Kernel class

```
{
  "sources": [
    {
      "restrictions": { },
      "function": "mat3",
      "codename": "mat3",
      "source": "lib.cl"
    },
    {
      "codename": "sigmoid1",
      "function": "sigmoid1",
      "source": "lib.cl",
      "restrictions": { },
    }
  ]
}
```

Figure 4.1: Sample content of a Kernel Library JSON file

`DynamicCL::Kernel` instances can be fetched from the `CLManager` instance via the `getKernel` method (see Appendix 12.1). We have, however, not yet explained what the parameter `codename` is. The `codename` parameter is a consequence of our kernel organisational structure explained in the next section.

4.4.1 Kernel Library JSON File

As stated in the requirements List 4.1, we need to be able to work with more variations of kernels performing the same function. For example, we may have three different variations of a matrix multiplication kernel, each implementation using different optimisations. Thus it is clear that we need to differentiate between *function name* and *functionality*, as many functions can have the same functionality, and for each functionality, we always only use one of them at a time. It is also clear, that if not handled correctly, this could lead to accidental invocation of incorrect kernels, leading to unexpected behaviour.

This is where the Kernel Library JSON File comes into play. When we call the `readLibrary` method of the `CLManager` (see Appendix 12.1), we do not pass it a path to the device-code source file. Instead, we pass it the path to the Kernel Library JSON File, which contains a list of all the kernels our program needs. For each kernel, this JSON file contains the name of the device-code function, the name of the file in which the function is located, and a *codename*, with which we can identify the particular kernel in the rest of our codebase. This codename is used to request `DynamicCL::Kernel` instances from the `CLManager` via the `getKernel` method.

A sample set of entries from a Kernel Library JSON File can be seen in Figure 4.1

Note that for every entry in the JSON file in Figure 4.1 there is a key *restrictions*, which

is, for this example, simply an empty JSON. When a DynamicCL::Kernel instance is requested from the CLManager, the returned instance also contains a reference to this restrictions JSON. The DynamicCL framework does not use this restrictions JSON for anything, it is there for use by the higher level method, which is going to pass arguments to the kernel. This feature will be crucial in Section 8.3.

4.5 Mirrorable

This section is aimed at presenting the Mirrorable abstract class. This class bridges the gap between host memory and device memory, by providing us with a simple push/pull interface. The Mirrorable abstract class is designed to wrap itself around a pointer to an array and ensure that a cl:Buffer with device-side memory of adequate size has been allocated for this array on the OpenCL device. The main interface methods provided by the Mirrorable class can be seen in Source 4.3.

```
1  template<typename T>
2  class Mirrorable {
3  public:
4    // Device memory management methods
5    Mirrorable& mirrorTo(const CLManager& manager);
6    int unMirror();
7    bool isMirrored();
8
9    // Synchronisation functions
10   Mirrorable& push();
11   Mirrorable& pull();
12
13   // Returns the cl::Buffer pointing to the device-side memory
14   cl::Buffer & getBuffer();
15
16   // Virtual size method. used internally by the Mirrorable.
17   virtual size_t getSize() = 0;
18
19   // The data itself
20   T * data;
21 }
```

Source 4.3: Main interface provided by the Mirrorable abstract class

The Mirrorable class ensures generalisability to other data types via the template T, as well as a great level of automatic failure detection and avoidance. Whenever the mirrorTo method is called, the Mirrorable remembers the reference to the CLManager passed to it as an argument. Thus when the Mirrorable's cl::Buffer is requested via the getBuffer method, the Mirrorable can check that the last cl::Buffer it

fetched is still valid via the `CLManager.verify` method. If at any point is the buffer *not* valid [1], the `Mirrorable` will automatically mirror itself again and push its content to the device, before returning the (now new and valid) `cl::Buffer`.

The `virtual size_t getSize` method is meant to be implementated in child classes. This is due to the fact that the data structure stored in `T * data` may be more abstract and its spacial restrictions cannot be generalised. For example, a matrix would have a certain amount of rows and columns, so the `getSize` method would return a product of these two values. Leaving the `getSize` method for child class implementation is more general and allows the child class to convey more complex data structures.

4.6 Summary

The purpose of this chapter was to give a brief introduction to the `DynamicCL` framework we wrote. It was written to ensure the automatic management of requirements in List 4.1 and with this framework written we may proceed to build the neural network execution system on top of it.

[1] This can happen as a consequence of changing the device in the `CLManager` instance or calling `unMirror` on the current `Mirrorable`

Chapter 5

Neural Network Execution System

In this section we use the `DynamicCL` framework from Section 4 to construct a system for executing neural networks with OpenCL. Since the goal of this work is to create a system for executing neural networks, not training them, we can limit our implementation to forward propagation only.

As stated in Section 2.1, a neural network is simply a series of affine transformations interleaved with nonlinearities. Therefore, a system capable of forward propagating inputs through a neural network must have all the functionality in List 5.1.

1. **Affine Transformation**

    - **Matrix cross product** - The basis of an affine transformation is a linear transformation using the standard matrix cross product.

    - **Adding biases** - An affine transformation involves a translation of data. In neural networks we call this the *bias*.

2. **Activation Functions**

    - **Sigmoid function** - Nonlinearity mapping inputs to the range [0;1]

    - **ReLU function** - Nonlinearity mapping inputs to the range [0;∞]

List 5.1: List of required functionality of our neural network evaluation framework

5.1 The `CLMatrix`

The first step in implementing the neural network evaluation system is to use the `Mirrorable` abstract class from Section 4.5 to represent a two-dimensional matrix.

For this purpose we implemented the `CLMatrix` class. This class stores the size of the matrix in rows and columns. Thanks to the `Mirrorable` abstract class, the definition for our `CLMatrix` class is extremely short and can be seen in Source 5.1. This is the basic `CLMatrix` class and will be further expanded with different optimisations we try in later sections.

```
1 using HonDataType = float;
2 class CLMatrix : public DynamicCL::Mirrorable<HonDataType> {
3 public:
4   size_t cols;
5   size_t rows;
6   size_t getSize() override {
7     return this->rows * this->cols;
8   };
9 }
```

Source 5.1: `CLMatrix` definition

5.1.1 Loading a Matrix into Memory

Since we shall be evaluating pre-trained neural networks, we need a means to load the matrices of these networks into the `CLMatrix` class. For this we created the Matrix Definition JSON files [1]. These files contain the dimensions of the matrix, the file containing it and information on the file format. An example of such a JSON file can be seen in Figure 5.1.

To load this matrix into the `CLMatrix` class, we pass the path to the JSON file to a matrix read method. Once the matrix is loaded, we may use the methods provided by the `Mirrorable` abstract class to mirror the matrix with the GPU and pass it as an argument to kernel functions.

5.2 Performers and `Operations`

In List 4.1 we mentioned that we will have may have multiple kernels with the same functionality, yet different implementations. In this section we introduce a mechanism for switching these implementations in and out of the system, namely the `Operations` struct and the Performer functions.

[1] We parse JSON files using Niels Lohmann's C++ JSON library [30]

```
{
  "rows": 50,
  "data_type": "csv",
  "file": "sample.csv",
  "cols": 100
}
```

Figure 5.1: Matrix definition JSON file example. This JSON file defines a 50-by-100 matrix stored in csv format in the file *sample.csv*

The Operations struct will allow us to globally change implementations across the entire system and the performer functions ensure that the kernels are fed correct arguments depending on which functionality we expect they have. Both are further explained in this section.

5.2.1 Operations

As mentioned in List 5.1, we need to be able to perform four different operations on the GPU: matrix multiplication, bias addition, sigmoid and ReLU. We may have multiple implementations for each of these, yet we need to be consistent in which one is used at any given moment by the system. We can do this using codenames. Recall from Section 4.3 how individual kernels are requested from the CLManager by their codename. To keep track of which implementations are used at any given time, we implement a struct, which holds the codenames of currently used implementations for each operation (Source 5.2).

```
1 struct Operations {
2   std::string sigmoidCodename = "sigmoid1";
3   std::string reluCodename = "relu1";
4   std::string matmulCodeName  = "cross1";
5   std::string biasAddCodename = "colAddRM";
6 };
```

Source 5.2: Operations struct. Used to hold the codenames of current implementations of each functionality.

It is crucial that there is only one instance of this Operations struct (or that multiple instances are handled with caution). Any classes and structures we create in subsequent sections of this chapter will hold C++ references to the std::string fields in this struct (or references to the struct itself). This allows us to globally change the implementation of any operation by simply changing the corresponding field in the Operations struct.

5.2.2 Performer Methods

Performers are the final step of abstraction from the low levels of OpenCL. They wrap within themselves the calls to our `CLManager` and the passing of arguments to our `DynamicCL::Kernel` instances. They allow us to perform high-level operations (like matrix multiplication) on high-level classes (the `CLMatrix`).

The idea is simple: no matter how many implementations of each operation we have, they are still going to have similar (or in many cases identical) method signatures. Thus for each operation we define a static method, which contains the process of passing arguments to the kernels and launching kernels corresponding to the particular operation. This static method then gets called from other places in the code whenever the operation is required (may be called from within Layer objects in the neural network).

A further task the performer methods do is related to the *restrictions* key in every kernel definition of the Kernel Definition JSON File from Section 4.4.1. We will further explain how these restrictions are useful in Section 8.3, as we have not yet introduced the particular situations in which their purpose is utilised.

5.3 The Neural Network

In this section we present the way in which we represent neural networks in terms of `CLMatrix` objects. The networks contain two kinds of layers, namely Affine Layers and Activation Layers, and out of a Model object, which chains these layers together. We also present a mechanism for loading neural networks from files using the Neural Network Definition JSON File.

We shall have three layer types: AffineLayer, SigmoidLayer and ReLULayer. For each layer type we implement an object, which extends the base `Layer` class seen in Source 5.2. As such, every layer will have an `fprop` method, which can be used to forward propagate values over the network.

```
1 class Layer {
2   public:
3   virtual CLMatrix& fprop(
4     DynamicCL::CLManager& manager,
5     CLMatrix& inputs, int& time
6   ) = 0;
7   virtual void display() = 0;
8 };
```

Figure 5.2: Interface of every `Layer` object.

5.3.1 Neural Network Definition JSON File

In Section 5.1.1 we presented a way of loading matrices into `CLMatrix` objects. In this section we present a similar way of defining neural networks in JSON files. A sample file can be seen in Figure 5.3. The Neural Network Definition JSON File contains a list of JSONs, each of which contains a *layer* key describing the layer type, as well as additional keys, specific to the particular layer type.

Being activation functions, neither the sigmoid Layer nor the ReLU layer require any parameters for functioning. The affine layer needs to know what weights and biases it is to be using. Hence the affine layer definitions contain the keys *weights* and *biases*, which contain the relative path to their respective Matrix Definition JSON files we mentioned in Section 5.1.1.

With this configuration, our program may now load arbitrary fully-connected feed-forward neural networks into `Model` objects and forward propagate values over it by passing the input values to its `Model.fprop` method.

```
{
  "layers": [
    {
      "layer": "AffineLayer",
      "weights": "1_w.json",
      "biases": "1_b.json"
    },
    {
      "layer": "SigmoidLayer"
    },
    {
      "layer": "AffineLayer",
      "weights": "2_w.json",
      "biases": "2_b.json"
    },
    {
      "layer": "SigmoidLayer"
    },
    {
      "layer": "AffineLayer",
      "weights": "3_w.json",
      "biases": "3_b.json"
    }
  ],
  "size": 5
}
```

Figure 5.3: Sample neural network definition JSON file

5.3.2 Affine Layer Execution

Due to the non-commutativity of matrix multiplication, it is important to clarify how exactly the affine layer forward propagates values. Suppose we have n input vectors $\vec{v}$ of dimensionality d and we want to feed them into an affine layer, which transforms them into h dimensional vectors. The input to an affine layer is will be a matrix I with d rows and n columns. The weight matrix W has h rows and d columns, and the bias matrix b has h rows and one column.

The input matrix is:

$$I = \left[\vec{v}_1, \vec{v}_2, \vec{v}_3 \cdots \vec{v}_n\right]$$

The resulting matrix R is computed by the affine layer as:

$$R = W \times I + b$$

This should clear any ambiguity in the implementation of kernels in future sections.

5.4 Summary

The goal of this chapter was to introduce our neural network execution system. The system enables loading neural network models from files, mirroring them to GPU memory and executing the appropriate OpenCL kernels for individual mathematical operations needed to forward propagate inputs over the loaded network. With this system completed, we may now address the writing and optimisation of the individual kernels to maximise forward propagation performance on the Mali T-628 GPU.

Chapter 6

Baseline

As a baseline, we are going to implement very elementary, unoptimised versions for the matrix multiplication, bias addition and the sigmoid activation function. We shall then use this setup to forward propagate values through a sample neural network and attempt to use the measured results to identify bottlenecks.

6.1 Matrix Multiplication

We shall begin with the matrix multiplication. As we know, the product of two matrices A and B of dimensions $a \times b$ and $b \times c$ respectively is a matrix C of dimensions $a \times c$ with each element C_{ij} calculated as:

$$C_{ij} = \sum_{k=1}^{b} \left(A_{ik} \times B_{kj} \right)$$

The stronger of the two Mali T-628 GPU OpenCL devices has 4 compute units, each of which can run 256 work items at the same time (as seen in Table 2.1). In our baseline we are going to have one work item compute one element of the resulting matrix. We are thus going to be mapping our kernel to a two-dimensional space of ids equivalent to:

$$\{0 \ldots (a-1)\} \times \{0 \ldots (c-1)\}$$

We shall not be specifying the workgroup size, as this is the focus of investigation in future sections. OpenCL has an automatic system for grouping the work items into workgroups, albeit it is not always the most effective. For our baseline, however, it will be sufficient. The kernel we implemented can be found in Source 6.1.

```
1  __kernel void cross1( uint l, uint w,
2      global const HON_DATA_TYPE* a,
3      global const HON_DATA_TYPE* b,
4      global HON_DATA_TYPE* c)
5    {
6
7      size_t row = get_global_id(0);
8      size_t col = get_global_id(1);
9      HON_DATA_TYPE cumulative = 0;
10     for (int i = 0; i < l; i++){
11       cumulative += a[row*l + i] * b[w*i + col];
12     }
13     c[row*w + col] = cumulative;
14 }
```

Source 6.1: Matrix multiplication baseline kernel. Parameter `l` stands for the length of the conjoining dimension of the matrices, while `w` stands for the width of the matrix

6.2 Sigmoid Function

The sigmoid activation function transforms the elements of the matrix with the logistic sigmoid function. The dimensions of the matrix remain the same. All elements of the matrix are computed as:

$$B_{ij} = \frac{1}{1 + e^{-A_{ij}}}$$

We shall once again compute each element in the resulting matrix in a single work item, much like we have done in the previous section for matrix multiplication. In contrast to the previous section, however, we are going to perform the transformation in place, that is, we shall replace the initial values with the transformed ones without moving them to a new matrix elsewhere in memory.

The kernel for this transformation can be seen in Source 6.2.

```
1 kernel void sigmoid1(global HON_DATA_TYPE * mat) {
2   size_t index = get_global_id(0);
3   mat[index] = 1.0 / (1 + exp(-mat[index]));
4 }
```

Source 6.2: Sigmoid activation baseline kernel

6.3 Bias Addition

Bias addition is the process of translating the individual features of each input vector by a certain quantity. In our research, inputs are represented as columns in forward propagating matrices. As such, the biases must also be a column matrix of size $d \times 1$, where d is the dimensionality of the input feature vectors.

Translating a matrix A by a bias vector b into the resulting matrix C is computed as follows:

$$C_{ij} = A_{ij} + b_i$$

We shall once again compute each element in the resulting matrix in a separate work item, mapping the bias addition kernel to the same two dimensional range as in Section 6.1. We are also going to perform this operation in place, storing the result in the corresponding location in the origin matrix.

The baseline bias addition kernel can be seen in Source 6.3

```
1 __kernel void colAddRM(uint rows, uint cols,
2     global HON_DATA_TYPE * mat,
3     global HON_DATA_TYPE * column)
4 {
5   size_t row = get_global_id(0);
6   size_t col = get_global_id(1);
7   mat[col + row*cols] += column[row];
8 }
```

Source 6.3: Bias addition baseline kernel

6.4 Evaluation

In order to evaluate the performance of our baseline system, we decided to train a small neural network on classifying the MNIST [31] picture set and use our system to classify 1000 of these images. The MNIST picture set is a dataset of 28-by-28 pixel hand written digits, equating to 784 features per input vector.

We trained a three layer neural network with three layers of widths 100, 100 and 10 neurons respectively, each but the last followed by a sigmoid layer. The neural network definition JSON file for this network is the one in Figure 5.3.

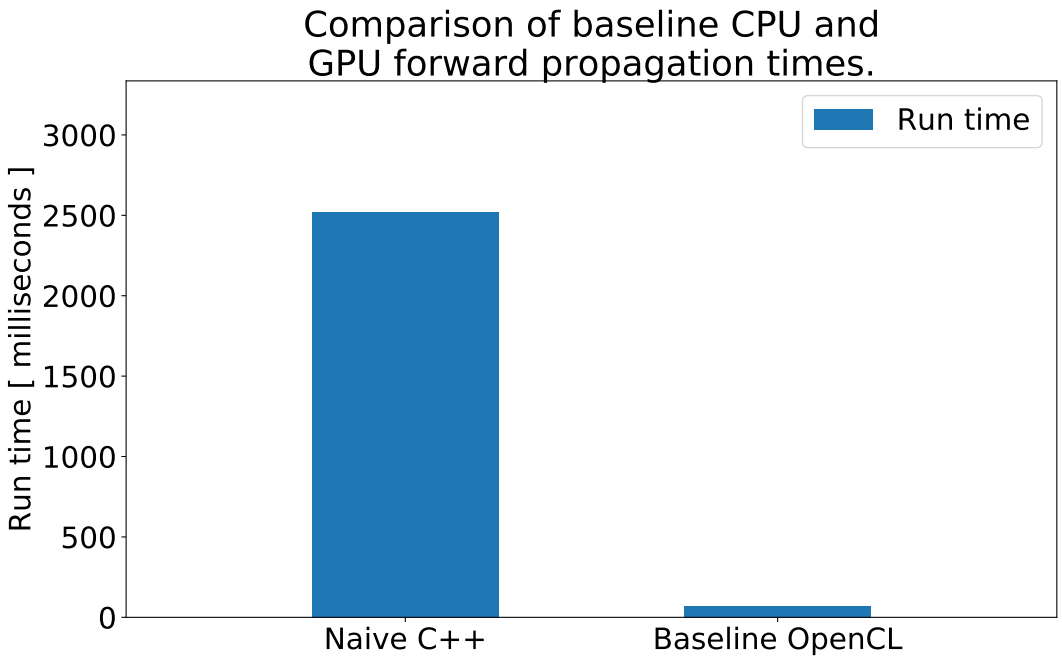

Figure 6.1: Time taken to forward propagate our baseline model. Run times are compared between time taken on a CPU (left) and time taken on a GPU with our implementation (right).

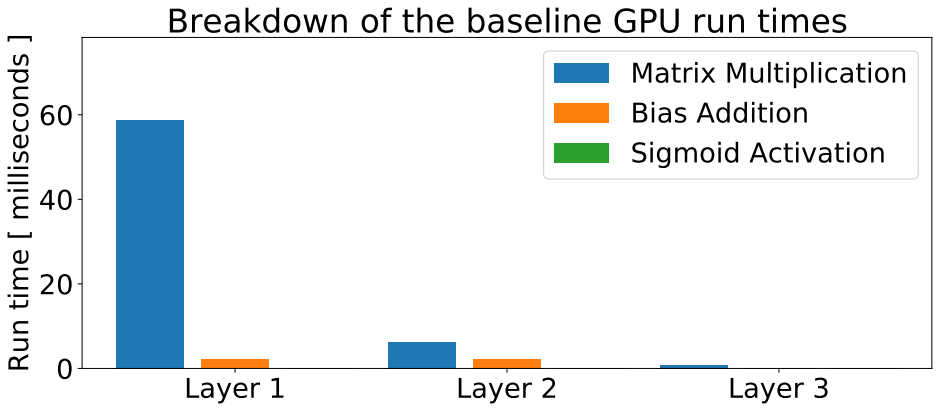

Figure 6.2: The breakdown of operation kernel run time for every operation in the baseline neural network. Plotted values are the average of ten measurements with negligible standard error (invisible on graph). The run time of activation layers was also negligibly small and invisible.

6.5 Summary

It is clear from Figure 6.1 that there is a significant gain in classification speed when using the Mali-T628 GPU on the Odroid board as opposed to simply using a sequential CPU implementation.

We can now further investigate the performance of our OpenCL baseline by breaking the runtime down into the individual kernels in order to find the main bottleneck. You can see this breakdown in Figure 6.2.

It is quite clear that the matrix multiplication is the primary bottleneck of our system. This was expected, as matrix multiplication is the most complex operation we use. The algorithm we use to compute the product of two matrices A and B with dimensions $a \times b$ and $b \times c$ respectively has complexity of $O(abc)$. It is also clear that matrix multiplication should be expected to take more time, as it is the only operation the kernel of which has a loop inside it, yet for every operation we run the same amount of work items.

Chapter 7

Optimizing the Matrix Multiplication

In our baseline we have observed that matrix multiplication is the major bottleneck of our current implementation. In this chapter we optimise matrix multiplication using a variety of techniques including altering memory layouts, optimising workgroup sizes, vectorising the operation and blocking.

Our baseline implementation for this operation is the standard matrix multiplication algorithm which, for square matrices of dimensions $n \times n$, has a complexity of $O(n^3)$. It is worth mentioning that there exist better methods with lower complexities, such as Strassen's algorithm, with complexity of $O(n^{log_7(8)})$ [32]. However, the benefits of using these algorithms becomes more apparent only once the sizes of the matrices reach the fifth order of magnitude, where a perfromance increase of roughly 32% may be achieved [32]. In this report we focus primarily on optimising by better utilizing available hardware rather than changing algorithms, as this yields considerably higher performance gains (up to 4000% in the final optimization stage).

7.1 Memory Layouts

The way a matrix is represented and stored in memory can significantly impact performance. Matrices are, conceptually, two-dimensional tables of numbers, however, computer memory is strictly one-dimensional. There are two main layouts in which a matrix can be represented in memory: **row major layout** (RM) and **column major layout** (CM). A graphical representation how the data is stored in memory with these layouts can be seen in Figure 7.1.

row major and column major layouts and their effect on matrix multiplication were explored by Johan Gronqvist and Anton Lokhmotov[12]. In this paper they referred to the layouts as **transposed** and **non-transposed**, and proceeded to redefine the cross product operation to compensate these transformations. We are, however, going to think of them in terms of row major and column major, as they have identical effects and open doors to explore various more complex layouts (Chapter 8).

For this work, we shall use the memory access notation presented in Johan Gronqvist's

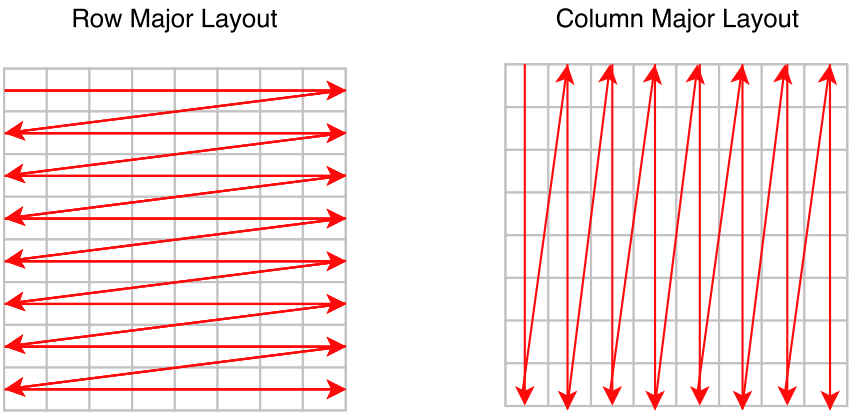

Figure 7.1: Row major and column major memory layouts

and Anton Lokhmotov's paper [12], with the slight alteration of replacing the comma with an up arrow indicating the direction of the accesses (more on this notation can be found in Appendix 12.8). In this notation, we may express the memory access pattern of our baseline matrix multiplication algorithm as seen in Equation 7.1. Equation 7.1 shows how the baseline matrix multiplication kernel accesses matrix B with a stride of size c.

$$\overset{b}{\underset{k=0}{\uparrow}} \Big(A[ib+k], B[kc+j] \Big)$$

Equation 7.1: Memory access pattern of a work item i, j in the baseline matrix multiplication kernel from Section 6.1, when multiplying matrices $A \times B$ of dimensions $a \times b$ and $b \times c$ respectively.

Accessing memory on the Mali GPU in strides is undesirable. We have established in Section 2.5.2 that Mali GPUs do not have hardware optimisation designated to coalesced memory accesses and all memory access optimisation is heavily cache-based. Strided memory accesses imply cache misses with *every* iteration of the loop. This means we may expect kernels which access memory sequentially, to perform better than strided kernels [1] and as such, that our current memory layout is sub-optimal.

In this section, we propose keeping the right-hand side matrix of the matrix multiplication kernel in Column Major form as a means of mitigating the rate of cache misses.

With this new proposed memory layout, every work item will be accessing data sequentially from both matrices while iterating through the `for` loop of the kernel. This way a cache miss will occur for both lines every 16 iterations, when the loop reaches the end of a cache line (one cache line fits 16 `float` values). This is significantly better than our baseline kernel, where cache misses occurred with *every* iteration.

This new proposed kernel has a memory access pattern as described in Equation 7.2

[1] strided kernels perform well on CUDA architecture, when concurrent work items of identical strides access coalesced memory simultaneously with every iteration of a loop

$$\overset{b}{\underset{k=0}{\uparrow}} \Big(A[ib+k], B[jb+k] \Big)$$

Equation 7.2: Memory access pattern of a work item i, j in the Row Major $\times$ Column Major (RM$\times$CM) matrix multiplication kernel when multiplying matrices $A \times B$ of dimensions $a \times b$ and $b \times c$ respectively.

(all access patterns are sequential). The kernel implementation can be found in Source 7.1.

```
1  __kernel void cross1( uint l, uint h,
2      global const HON_DATA_TYPE* a,
3      global const HON_DATA_TYPE* b,
4      global HON_DATA_TYPE* c)
5    {
6
7    size_t row = get_global_id(0);
8    size_t col = get_global_id(1);
9    HON_DATA_TYPE cumulative = 0;
10
11   for (int i = 0; i < l; i++) {
12     cumulative += a[row*l + i] * b[col*l + i];
13   }
14   c[col*h + row] = cumulative;
15 }
```

Source 7.1: Matrix multiplication kernel with optimised memory layouts. Parameter `l` stands for the length of the conjoining dimension of the matrices, while `h` stands for the height of the resulting matrix. Expects matrix a to be in row major layout and b to be in column major layout. Resulting matrix is saved in column major layout.

Dimensions	$RM \times RM$		$RM \times CM$	
	GFLOPS	$\pm$	GFLOPS	$\pm$
128	2.516	0.04684	2.406	0.00216
256	0.448	0.01002	0.599	0.03241
384	0.397	0.00215	0.480	0.00116
512	0.330	0.00070	0.409	0.00035
640	0.364	0.00052	0.449	0.00042
768	0.352	0.00042	0.447	0.00011
896	0.332	0.00058	0.443	0.00008
1024	0.304	0.00029	0.428	0.00012
1152	0.283	0.00023	0.392	0.00006

Table 7.1: Performance of matrix multiplication for square matrices with different memory layouts.

7.1.1 Evaluation

In order to evaluate the performance of the kernels in Source 6.1 and Source 7.1, we used them to multiply progressively larger square matrices, measuring the time and computing the amount of floating point operations performed per unit of time. The results can be seen in Table 7.1. Every multiplication was performed 10 times in order to account for deviations and to present an informative standard error.

We may make three observations from these values. Firstly, we see a large performance spike with matrices of size 128 by 128. This is because two single precision `float` matrices of these sizes exactly fit into the cache. As such, cache misses are minimal, since nothing needs to be evicted. This result shows us how sub-optimal this solution still is with regards to the cache.

We may further note that the Row Major - Column Major ($RM \times CM$) product outperformed the baseline by roughly 0.1 GFLOPS for every other matrix size, which we expected.

Lastly, it is clear from the negligible deviations in these results, that the kernels tend to have a stable performance for every test run. In further graphs and tables, error bars will be omitted unless they are significant, and you may safely assume that if error bars are not present in a table of graph, it is because they are negligible.

7.2 Workgroup Sizes

As mentioned in Section 7.1, we can limit cache misses to occur only once every 16 iterations of the matrix multiplication kernel loop. In the worst case, however, no two work items access the same cache lines at the same time, meaning when the cache misses *do* occur every 16 iterations, the *entire* cache may need to be evicted and replaced [2]. This would of course cause substantial stalling and unacceptable delays. In this section we address the issue of the worst case scenario, and ensure that the worst case scenario never occurs We do so using work group size specification.

In Section 6.1 we showed how when multiplying matrices $A \times B = C$ of sizes $a \times b$ and $b \times c$ respectively, the matrix multiplication kernel is mapped to the set of number pairs $\{0 \ldots (a-1)\} \times \{0 \ldots (c-1)\}$. Our baseline allowed OpenCL to decide which pairs get grouped together and execute on the same core at the same time, an approach which does not guarantee optimality.

For any work item with ids (i, j) executed on a core, the work item will access the i^{th} row in A and the j^{th} column in B. Trivially then, once work item (i, j) is executing on the core, work items $(x, j)\ \forall\ j$ and $(i, y)\ \forall\ y$ can also execute on the core, at a cost of only *one* additional cache line per work item at any given point.

[2] In the worst case scenario, the amount of cache space needed to hold all the pairs of cache lines traversed by individual kernels is exactly equal to the size of the cache.

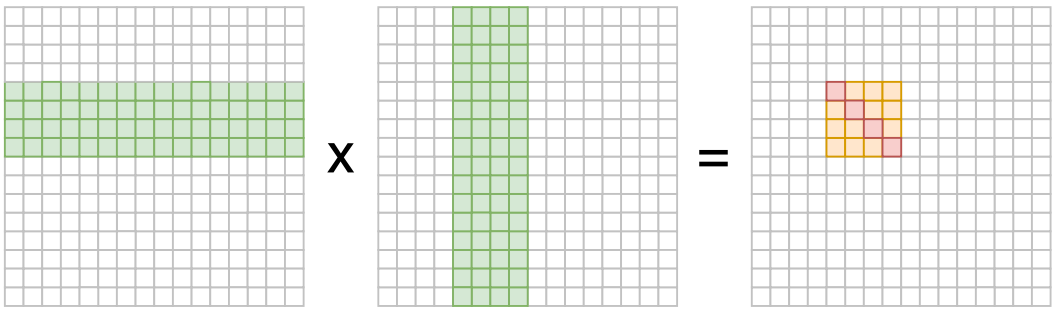

Figure 7.2: Memory access pattern of the matrix multiplication kernel when grouping into squares.

We may build upon this observation and group our work groups into squares. This way, every work group will compute a square region of the resulting matrix as seen in Figure 7.2. The work group computing the orange-red region of the resulting matrix will access the green rows and columns. The idea here is that the green rows and columns are needed to compute the red result elements alone and the orange elements may be computed concurrently with the red elements, reusing the same data, thus getting computed at a virtually zero additional memory access cost.

One may hence be tempted to make workgroups compute square regions in the resulting matrix, however, this approach would not be ideal. We know the Mali board has 4 compute units, meaning it can compute 4 workgroups at the same time. Therefore, we need the combination of all concurrent workgroups to collectively compute a region, the shape of which is as close to a square as possible [3].

In order achieve this coverage of concurrent workgroups, we must investigate the order in which individual workgroups are executed on the device.

7.2.1 Workgroup Execution order

For the purpose of simplicity, lets assume that we were to force OpenCL to have workgroups of size 1 on a device with a single core. Let λ_0 be the global id accessible via `get_grobal_id(0)` and λ_1 be the global id accessible via `get_grobal_id(1)`.

For the matrix product $A \times B = C$ where A and B have dimensions $a \times b$ and $b \times c$ respectively, we then know that $(\lambda_0, \lambda_1) \in \{0 \ldots (c-1)\} \times \{0 \ldots (a-1)\}$ (since λ_0 denotes the *row* and λ_1 denotes the *column*).

Related literature [12] suggests that kernels will be called in the order:

$$\overset{c-1}{\underset{\lambda_0=0}{\uparrow}} \left(\overset{a-1}{\underset{\lambda_1=0}{\uparrow}} \Big(kernel(\lambda_0, \lambda_1) \Big) \right)$$

[3] The cache is shared between the cores, as we established in Section 2.1, which makes this approach work.

This claim, however, never clearly refers to any source, so in this work we experimentally verified it.

To verify this, a testing kernel was set up. The kernel stored the current number of non-zero elements in the matrix + 1 into its corresponding element. Running this kernel on a 33-by-33 matrix, with workgroup size limited to 1, the kernel clearly demonstrated behaviour expected from such execution ordering. The resulting matrix is in Appendix 12.4 and it clearly shows how the workgroup execution flowed down the matrix in columns.

7.2.2 Choosing the Appropriate Group Size

With the knowledge from the previous subsection, we may deduce that the optimal dimensions of a matrix multiplication kernel workgroup will map to a rectangle of elements in the resulting matrix. The dimensions of this rectangle are $a \times 4a$ for some a, which is itself a multiple of 4 (due to the preferred workgroup size of the device). These restrictions combined with the maximum workgroup size being 256, effectively limit our workgroup size selection to two possible sizes: 4×16 and 8×32.

We have decided to set the workgroup sizes to 4×16, as this size outperformed the latter in all experiments, albeit by a tiny margin. This workgroup size is also one that reproduced the best results from the ARM paper [12] covering matrix multiplication on the Mali board.

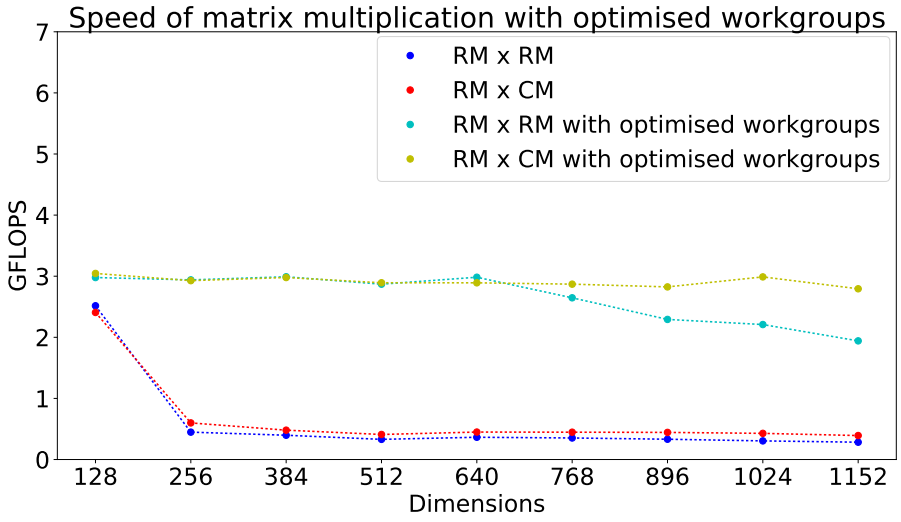

Figure 7.3: Performance of the matrix multiplication kernel when grouping into squares (Four workgroups of dimensions 4×16)

7.2.3 Evaluation

We have evaluated the performance of the matrix multiplication kernel with optimised workgroups by using it to compute the results our previous implementations were benchmarked against in Section 7.1. The results of these experiments can be seen in Figure 7.3, where they are compared to the baseline (blue) and kernel from Section 7.1 (red).

It is clear that the kernels run with optimised workgroup sizes significantly outperform kernels run with default OpenCL groupings. These results are consistent with all our assumptions and are in line with the results presented by ARM [12].

It is also worth mentioning that we have run the workgroup-optimised kernel for both RM $\times$ RM (cyan) *and* RM $\times$ CM (yellow), to demonstrate how Column Major Layout is indeed needed in order for the workgroup optimisation to take full effect for larger matrix sizes.

7.3 Vectorisation

So far we have been computing matrix multiplication by operating on individual values. The Mali GPU, however, is highly specialised for vector operations and can greatly accelerate computation when this architectural feature is exploited [14] [33][12].

The Mali GPU contains 128 bit multi-purpose registers, which can be used within kernels as the `float4` type.

Advantages of using vectors include being able to load multiple values from memory and perform operations on them with a single instruction (SIMD - Single Instruction Multiple Data), fully utilising the cores' hardware. OpenCL also includes several useful built-in functions for general vector operations, such as the `dot` function, which we make heavy use of in this section.

Disadvantages to using vectors are that the vectors may only be loaded from memory as sequential values (next to each other). This means kernels cannot simply be rewritten to use vectors, as there may be problems with memory layouts. This problem is averted for our case, as the RM $\times$ CM memory layout system introduced in Section 7.1 ensures consecutive memory accesses.

In the previous matrix multiplication kernels we computed the dot product of the corresponding row and column by iterating over all pairs, cumulatively adding intermediate results. Using vector types, we can iterate over quadruplets of these elements, computing the cumulative dot product using the built in `dot` function. A vectorised implementation of the RM $\times$ CM matrix multiplication kernel can be seen in Source 7.2.

It is worth mentioning that this kernel relies on the matrices' dimensions being multiples of 4. This means matrices need to be padded with zeros in order to achieve the required dimensions. We have faced a similar problem in Section 7.2, and how this is

dynamically achieved is further explained in Section 8.3 along with other assumptions made in later sections.

```
1  __kernel void crossv(uint l, uint h,
2      global const HON_DATA_TYPE_VECTOR *a,
3      global const HON_DATA_TYPE_VECTOR *b,
4      global HON_DATA_TYPE *c)
5    {
6    size_t row = get_global_id(0);
7    size_t col = get_global_id(1);
8
9    l >>= 2;
10   HON_DATA_TYPE cumulative = 0;
11   uint ai = row*l;
12   uint bi = col*l;
13
14   for (int i = 0; i < l; i++) {
15     cumulative += dot(a[ai], b[bi]);
16     ai++;
17     bi++;
18   }
19   c[col*h + row] = cumulative;
20 }
```

Source 7.2: Vectorised RM $\times$ CM matrix multiplication kernel. Parameter `l` stands for the length of the conjoining dimension of the matrices, while `h` stands for the height of the resulting matrix. Expects matrix a to be in row major layout and b to be in column major layout. Resulting matrix is saved in column major layout.

7.3.1 Evaluation

The vectorised implementation of matrix multiplication kernel has been compared to the other implementations by timing its performance on computing the same multiplication problems. The results of this experiment, compared to previous implementations, can be seen in Figure 7.4.

The graphed results in Figure 7.4 show how impactful vectorisation can be. The vectorised kernel outperforms the workgroup-optimised kernel from Section 7.2 by a factor of 2.8, and outperforms the baseline by a factor of 27.

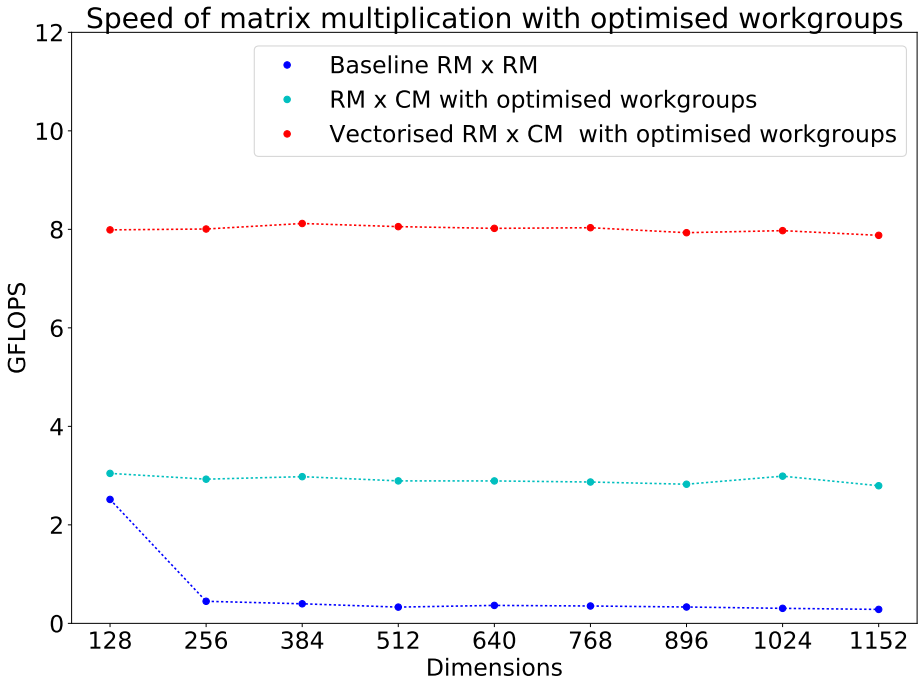

Figure 7.4: Performance of the vectorised matrix multiplication kernel, compared to the performance of previous implementations.

7.4 Blocking

The optimal Mali T-628 matrix multiplication kernel presented in ARM research [12] uses a technique called *blocking*. In this chapter we reproduce the results of this paper and aim at extending them by generalising the kernels to non-square matrices and by providing the optimal workgroup size (which the paper does not specify). In Chapter 8 we shall improve the performance of these kernels by making use of more complex memory layouts.

So far all the presented matrix multiplication kernels computed a *single* entry in the resulting matrix. Blocking is the idea of using vector types and SIMD operations to make a single kernel compute multiple results concurrently. Blocked kernels have one significant advantage compared to the vectorised one from Section 7.3: they do not require as many work items. Computing four results in one work item means that the amount of work items is reduced by a factor of four, and hence (for large matrices), the number of workgroups also drastically falls. Reducing the number of workgroups allows us to avoid the overhead associated with swapping workgroups in and out of the device's cores.

In this chapter we present a blocked variant of both RM $\times$ RM and RM $\times$ CM kernels, adapted from the ARM paper [12] to support non-square matrix multiplication.

7.4.1 Blocked RM × RM Matrix Multiplication Kernel

In this kernel variant, we compute 1×4 chunks of the resulting matrix per work item. With every iteration of the kernel, one `float4` is loaded from matrix A and 4 `float4`s are loaded from matrix B and their corresponding products are computed. The cumulative results are stored in a `float4`.

This kernel is best described with a diagram. Figure 7.5 shows the execution pattern of the blocked RM × RM kernel. Each iteration accesses a set of vectors in a sliding window, which moves along the row and columns of the matrices with a stride of 4 (the red squares signify the windows accessed in the second iteration of the kernel). For every iteration, the cumulative result is updated as seen at the bottom of the figure.

The full kernel can be found in Appendix 12.2.

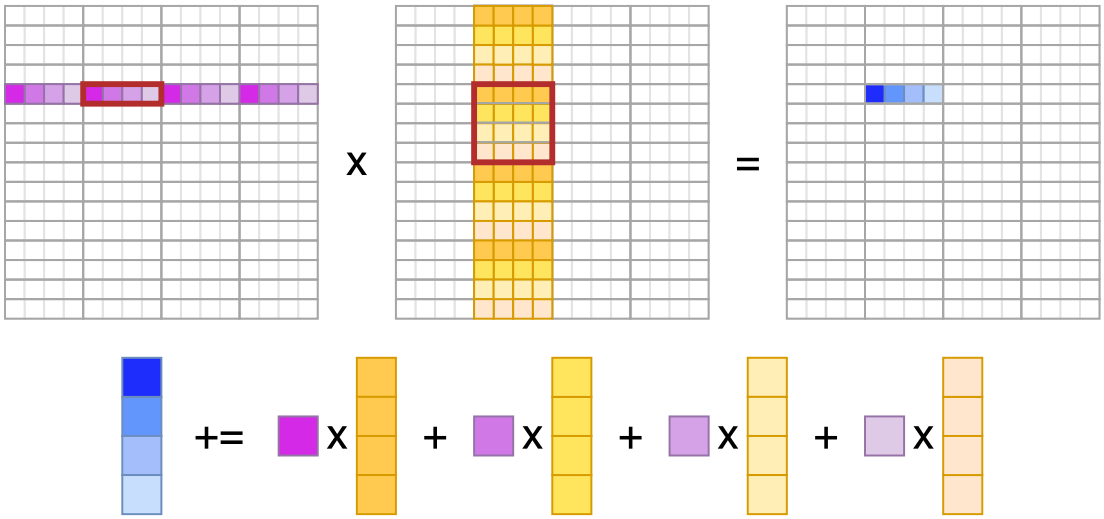

Figure 7.5: Execution diagram of the blocked RM × RM kernel.

7.4.2 Blocked RM × CM Matrix Multiplication Kernel

The blocked RM × CM matrix multiplication kernel works on a similar principle to the RM × RM variant, except it computes 2×2 chunks of the resulting matrix per work item. With every iteration of the kernel, two `float4` are loaded from matrix A and 2 `float4`s are loaded from matrix B, and their corresponding products are computed. The cumulative results are stored in a `float4`.

Figure 7.6 shows the execution pattern of the blocked RM × CM kernel. For every iteration, the cumulative result is updated as seen at the bottom of the figure. Note that in this instance, the 'x' operator denotes the OpenCL built in `dot` function, which computes the dot product, returning a scalar.

The full kernel can be found in Appendix 12.3.

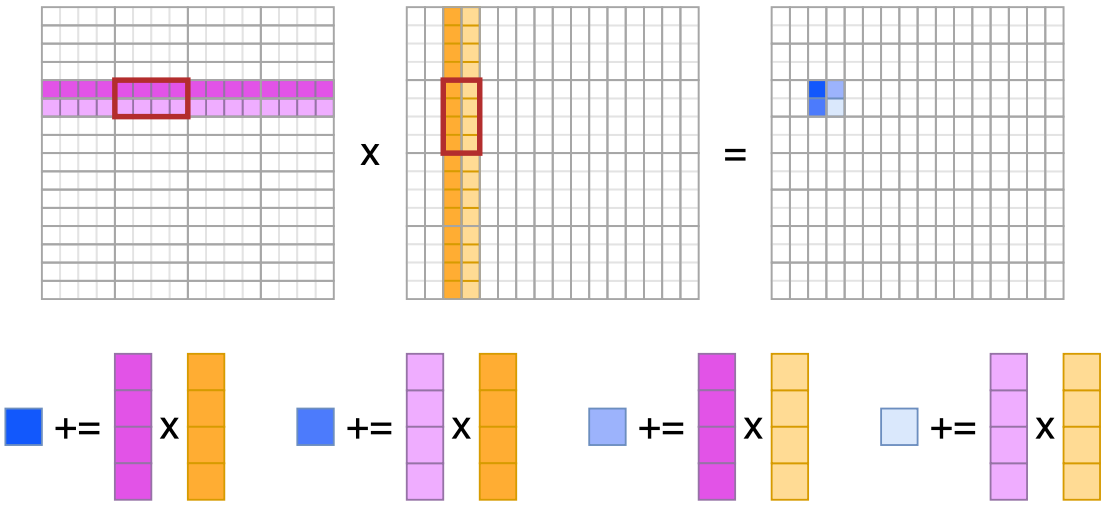

Figure 7.6: Test

7.4.3 Evaluation

We evaluate the blocked variants of RM × RM and RM × CM kernels on the same set of problems as the other kernels. The RM × RM is the first kernel, where a single work item does not cover a square area of the resulting matrix. In order to preserve the workgroup shapes investigated in Section 7.2, we must set the workgroup size of this kernel to 8×8. The workgroup sizes of the RM × CM kernel remain 4×16.

We can see in Figure 7.7, the RM × CM kernel presented in ARM research [12] outperforms all other implementations and is hence a good candidate for the neural network execution system. The RM × RM variant performed roughly as well as the vectorised kernel from Section 7.3.

The blocked RM × RM kernel is the best performing kernel which operates on matrices in row mayor layout only. As such, this would be the preferred kernel for applications where transforming matrices to other memory layouts is not possible.

7.5 Summary

In this chapter we applied various optimisation techniques to improve the performance of matrix multiplication. The best performing kernel made use of the blocking approach suggested by ARM for the Mali T-628 GPU. Once optimised, the kernel outperformed our baseline kernel by a factor of 33.

In the following chapter we optimise this kernel further using Morton Order memory layouts.

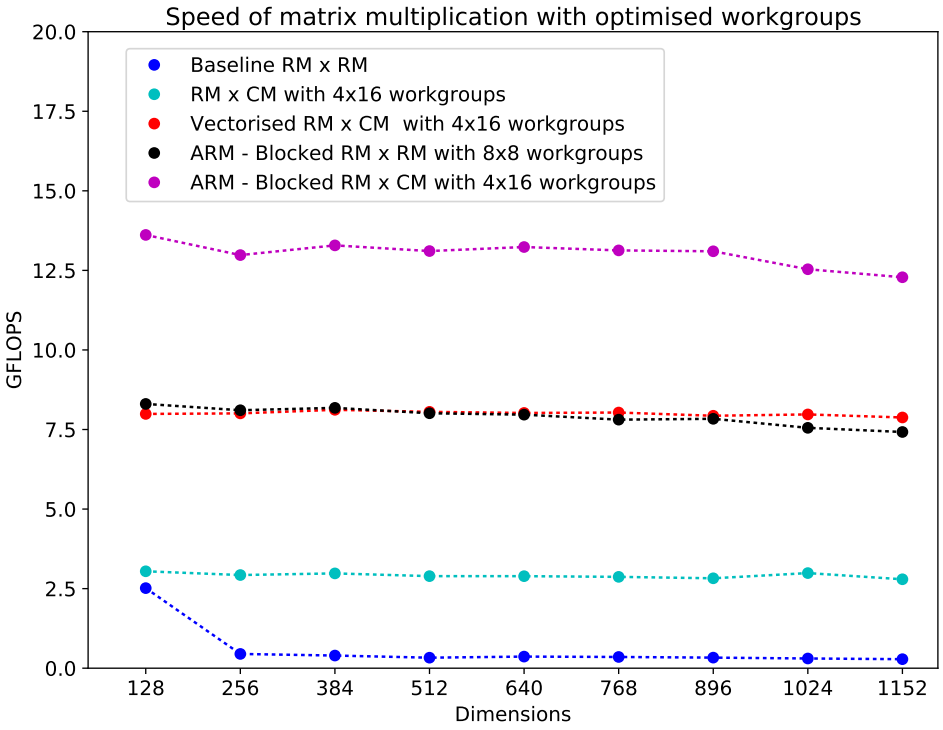

Figure 7.7: Execution diagram of the blocked RM $\times$ CM kernel.

Chapter 8

Morton Order Memory Layouts

Memory layouts have been touched upon in Chapter 7 and in this chapter we expand upon the initial ideas by introducing Morton-Order memory layouts.

Performance of matrix multiplication is highly dependent on whether data elements are accessed in the same order in which they are stored in memory. Research into memory access performance indicates that accessing elements in optimal order yields significant increases in performance and, contrary to that, accessing them with large strides has significant negative effects [34]. When we addressed memory access patterns in Sections 7.1 and 7.2 however, the performance differences were certainly not as significant as literature would have us expect them to be.

It is our assumption that the cause of this phenomenon is the concurrent access to individual sequential rows of data in the matrix. We have, in a sense, a combined access pattern, where data is accessed sequentially in strides. It is clearer what this means when expressed in a memory access formula. Suppose we are computing the matrix product $A \times B = C$, where matrices A and B have dimensions $a \times b$ and $b \times c$ respectively. Now suppose we compute this product using the blocked RM $\times$ CM kernel from Section 7.4, with workgroups of size $\lambda_0 \times \lambda_1$. The memory access pattern of this kernel will be as follows:

$$
\begin{aligned}
&\overset{c-1}{\underset{\lambda_0=0}{\uparrow}}\Bigg(\overset{a-1}{\underset{\lambda_1=0}{\uparrow}}\Bigg(\overset{\frac{b}{4}-1}{\underset{k=0}{\uparrow}}\Bigg(\overset{4}{\underset{core=1}{\uparrow}}\Bigg(\overset{\lambda_1}{\underset{j=0}{\uparrow}}\Bigg(\overset{\lambda_0}{\underset{i=0}{\uparrow}}\Big(A[2*i*\frac{b}{4}+k]\Big)\Bigg)\Bigg),\\
&\qquad\overset{4}{\underset{core=1}{\uparrow}}\Bigg(\overset{\lambda_1}{\underset{j=0}{\uparrow}}\Bigg(\overset{\lambda_0}{\underset{i=0}{\uparrow}}\Big(A[(2*i+1)*\frac{b}{4}+k]\Big)\Bigg)\Bigg),\\
&\qquad\overset{4}{\underset{core=1}{\uparrow}}\Bigg(\overset{\lambda_1}{\underset{j=0}{\uparrow}}\Bigg(\overset{\lambda_0}{\underset{i=0}{\uparrow}}\Big(B[2*j*\frac{b}{4}+k]\Big)\Bigg)\Bigg),\\
&\qquad\overset{4}{\underset{core=1}{\uparrow}}\Bigg(\overset{\lambda_1}{\underset{j=0}{\uparrow}}\Bigg(\overset{\lambda_0}{\underset{i=0}{\uparrow}}\Big(B[(2*j+1)*\frac{b}{4}+k]\Big)\Bigg)\Bigg)\Bigg)\Bigg)\Bigg)\Bigg)
\end{aligned}
$$

This is rather complicated to analyse, so we shall make a couple of simplifying assumptions in order to properly analyse the pattern:

- We shall only analyse the pattern on a single workgroup on a single core.
- We shall assume that concurrent accesses to the same memory location made by multiple work items have the same cache miss cost as a single cache miss cost
- Precedence of variables looping over the workgroup size is commutative, as the work items execute concurrently.

Applying these assumptions to the memory access pattern, we arrive at the following simplified formula:

$$\begin{aligned}\overset{\frac{b}{4}-1}{\underset{k=0}{\uparrow}}\Bigg[&\overset{\lambda_0}{\underset{i=0}{\uparrow}}\left(A[2*i*\frac{b}{4}+k]\right),\overset{\lambda_0}{\underset{i=0}{\uparrow}}\left(A[(2*i+1)*\frac{b}{4}+k]\right),\\&\overset{\lambda_1}{\underset{j=0}{\uparrow}}\left(B[2*j*\frac{b}{4}+k]\right),\overset{\lambda_1}{\underset{j=0}{\uparrow}}\left(B[(2*j+1)*\frac{b}{4}+k]\right)\Bigg)\Bigg]\end{aligned}$$

The outer loop of this equation signifies the sequential access pattern in memory. The inner loops, however, signify the exact opposite - strided access pattern caused by the concurrency of threads.

It is clear from this formula that as the outer loop progresses, we need to store at least $2\lambda_0$ cache lines of matrix A and $2\lambda_1$ cache lines of matrix B in the cache if we want to avoid a cache miss on every iteration.

We know from the specifications in Table 2.1 that the size of a cache line is 64 bytes, meaning it can store 16 floating point numbers or four `float4` vectors. This, along with the memory access pattern, implies that every 4 iterations of the kernel, $2(\lambda_0+\lambda_1)$ cache misses will occur, as the sliding windows from Figure 7.6 reach the end of a cache line. We shall use the terms **cache miss period** and **cache miss bulk** to refer to the amount of loops between cache miss occurrences and amount of simultaneous cache misses that occur respectively.

Prefetching is a common technique used to mitigate cache miss latencies by preemptively fetching data likely to be requested in the near future into the cache. Prefetching algorithms usually exploit spacial and temporal locality and it is reasonable to assume that similar algorithms are at work in the Mali GPU cache. What is not clear is how quickly the prefetcher can respond to accesses reaching the end of a cache line and whether the memory bandwidth allows the cache to prefetch the full cache miss bulk in time.

This is where Morton Order layouts come in. Morton Order layouts are a middleground approach to access patterns and they minimise the cost of using a combination of row-wise and column-wise access patterns. They do this by reducing the cache miss

bulk at the cost of increasing the cache miss period. So for instance, in our kernel from Section 7.4, we used workgroups of size 4x16. By the logic of the memory access pattern, this would imply our program needs to handle $2 \times (4 + 16) = 40$ cache conflicts every 4 iterations of the kernel loop. We can use Morton Order layouts to change this to a more distributed miss rate of 10 cache misses every 1 cycle, or 20 cache misses every 2 cycles.

8.1 Hybrid Morton Order Layouts

The idea behind Morton Order layouts is to cache lines cover a region of the matrix, instead of having cache lines linearly stretch out over either the rows or columns of a matrix. One of the most popular variants of the Morton order layouts is the **Z-Morton** layout, which can be seen in Figure 8.1. The spots in this Figure represent the elements in the matrix, while the lines connecting them represent the layout of the elements in memory.

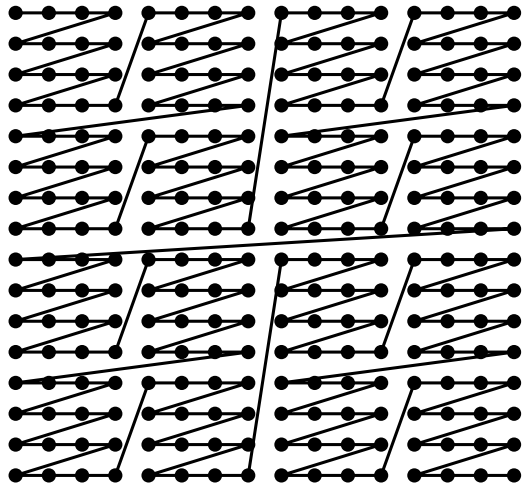

Figure 8.1: Z-Morton memory layout.

In this work we do not use the Z-Morton layout itself, we make use of its **Hybrid** variants. Examples of these can be seen in Appendix 12.5. The reason for using hybrid variants as opposed to the Z-Morton layout is that, as mentioned previously, Z-Morton Layout is useful for situations in which it is unclear what order memory accesses will happen in. In our case, however, we have a certain amount of control over this using workgroup size specification and as such, we do not need the recursive layout, only a tiled one.

8.2 Computing Hybrid Morton Order Indices

In order to work with Hybrid Morton Order layouts we first need a means of converting between element coordinates in row-column format to memory index. This is relatively straight-forward for Row Major and Column Major layouts, however, it becomes significantly more complex for Hybrid Morton Order layouts. We start by designing a

Context Free Grammar for generating labels, which fully convey the layout features of different Hybrid Morton Order Layouts as follows:

$$
\begin{aligned}
\mathit{Layout} &\rightarrow Y_0\ X_0\ \mathit{Base}_0\ \mathit{Recursion}_1 \\
\mathit{Recursion}_k &\rightarrow 1\ 1\ A \mid \mathit{Conversion}\ \mathit{Recursion}_{k+1} \\
\mathit{Conversion}_k &\rightarrow Y_k\ X_k\ \mathit{Base}_k \\
\mathit{Base} &\rightarrow R \mid C \\
Y &\rightarrow \mathit{Any\ natural\ number} \\
X &\rightarrow \mathit{Any\ natural\ number}
\end{aligned}
$$

While this grammar may seem complicated, it simply conveys the basic layout structure. Every Hybrid Morton Order layout is composed of *chunks* (or *tiles*) of elements in either Row Major or Column Major layout. These chunks are subsequently arranged in either Row Major or Column Major layout to form a larger chunk. In our grammar, the individual layers of chunk layouts are represented in *X Y Base* triplets. *X* and *Y* denote the size of the current chunk, while *Base* denotes the layout of smaller chunks within this chunk.

In this notation, the Z-Morton layout from Figure 8.1 would be denoted as `16 16 R 4 4 R 1 1 A`. The leading `16 16` are the full dimensions of the matrix and are hence redundant in the naming. The trailing `1 1 A` denotes the final 1x1 elements in *Arbitrary* layout, which is also redundant for our notation and will be omitted. The shortened notation for the Z-Morton layout is hence `R 4 4 R`.

For a term of this grammar to describe a proper layout, the following condition must be met:

$$
\forall\, k > 0 \left[\ X_k \mid X_{k-1}\ \mathit{and}\ Y_k \mid Y_{k-1}\ \right]
$$

We may now begin constructing the formula. Since the grammar is based on Row Major and Column Major layouts of chunks, we need the formula for converting from row-column coordinates to the index within these layouts. We may construct it as follows:

$$
\mathit{Index}(\mathit{layout}, \mathit{rows}, \mathit{cols}, \mathit{row}, \mathit{col}) = \begin{cases} \mathit{row} \times \mathit{cols}\ +\ \mathit{col},\ \mathit{if\ layout} = R \\ \mathit{col} \times \mathit{rows}\ +\ \mathit{row},\ \mathit{otherwise} \end{cases}
$$

With this formula we may define a recursive relationship between the triplets of $Num_x\ Num_y\ Base$ and the Hybrid Morton Order Layout index in the corresponding layout as follows:

$$MortonIndex(row, col) = MIndex(X_0,\ Y_0,\ Base_0,\ row,\ col)$$

$$\begin{aligned}
&MIndex(1,\ 1,\ row,\ col)\ =\ 1\\
&MIndex(X_k, Y_k, Base_k, row, col)\\
&\qquad = Index\big(Base_k,\ \frac{X_k}{X_{k+1}},\ \frac{Y_k}{Y_{k+1}},\ \frac{row}{Y_{k+1}},\ \frac{col}{X_{k+1}}\big)\ \times X_{k+1}\ \times Y_{k+1}\\
&\qquad + MIndex\big(X_{k+1},\ Y_{k+1},\ Base_{k+1},\ row\ \%\ Y_{k+1},\ col\ \%\ X_{k+1}\big)
\end{aligned}$$

Using these definitions, we may implement a reshaping system, which transforms matrices from one layout to the other. This is explained in the following section.

8.3 Implementation

There have been multiple instances throughout Chapter 7, where certain assumptions were made about the alignment of matrix dimensions and altered memory layouts, yet their implementation was never addressed until now.

In this section we address the infrastructural integration of Morton Order layouts into our neural network evaluation framework, as well as mechanisms, which enforce all assumptions made in previous Sections.

When we defined the Kernel Library JSON file in Section 4.4.1, we included an empty *restrictions* key in the kernel definition. This key contains all necessary information for the Performers (Section 5.2) to correctly instantiate the kernel on the GPU, enforcing all the assumptions we made in previous Sections.

This section lists the series of keys added to the *restrictions* JSON, what they mean, and which assumptions/requirements they are used to ensure are met.

8.3.1 *Arglist* key

Throughout Chapter 7, we presented matrix multiplication kernels which expected different arguments (Source 7.1 and Soucre 6.1 being examples). Since these kernels are launched by the same Performer, the Performer needs to know which arguments the kernel expects. This is encoded in the string value under the key *arglist*, and may be interpreted by the respective Performer by convention.

In our case, for matrix multiplication kernels, this key contains the string ”lwh”, where l, w and h stand for the common dimension, width of the resulting matrix and height of the resulting matrix respectively. The Performer will only pass those arguments to the kernel, which are present under this key.

8.3.2 *Scale* key

Until blocking was introduced in Section 7.4, matrix multiplication kernels were mapped to *all* elements of the resulting matrix. Since blocked kernels compute multiple results within a single work item, fewer kernels need to be launched. The *scale* key contains the downscaling factors for the kernel id ranges. So, for example, the 1x4 blocked kernel from Section 7.4.1 would have the *scale* key set to the json {`"rows":1, "cols":4`}, which ensures only one kernel is launched for every 1x4 chunk of the resulting matrix.

8.3.3 Matrix Restrictions key

Throughout Sections 7.1, 7.2 and 7.3 we wrote kernels which relied on matrices following certain memory layouts and having dimensions aligned to a factor of some constant (usually 4). Our framework ensures these requirements are met before initiating a kernel by reading the *matrix restrictions* for each matrix passed as an argument to this kernel, and, if necessary, reshaping this matrix to the correct form automatically. As such, matrix multiplication kernel entries in the Kernel Library JSON File will have three matrix restriction entry keys in the restriction JSON, one for each matrix (two input matrices and the output matrix).

The matrix *restriction* entry contains a JSON. This JSON specifies three requirements for the matrix: row alignment, column alignment and memory layout. Being a String, the memory layout key may contain the representation of the required memory layout in the Context Free Grammar syntax we defined in Section 8.2.

8.3.4 *Flop* key

The *flop* key defines the number of floating point operations associated with computing a certain part of the result. For matrix multiplication, this is the price of computing the product of two elements and their subsequent summation (2 floating point operations). This key is used by the Performer to compute the GFLOPS metric for every kernel execution.

An example *restriction* entry in a matrix multiplication kernel definition in the Kernel Library JSON file can be seen in Figure 8.2.

```
"restrictions": {
  "flop": 2,
  "a": {
    "representation": "R",
    "colAlign": 4,
    "rowAlign": 4
  },
  "b": {
    "representation": "C_4_4_C",
    "colAlign": 4,
    "rowAlign": 4
  },
  "r": {
    "representation": "C_4_4_C",
    "colAlign": 4,
    "rowAlign": 4
  },
  "arglist": "lh",
  "scale": {"rows": 1, "cols": 4}
}
```

Figure 8.2: Sample matrix multiplication kernel restrictions entry of the Kernel Library JSON file.

8.4 Hybrid Morton Order Matrix Multiplication

Kernel	Matrix Layouts			Statistics	
	Left	Right	Result	Miss bulk size [1]	Miss period [2]
Baseline	R	C	C	40	4
Morton 4-2	R 2 4 R	C 4 2 C	R 4 2 C	20	2
Morton 4-4	R 4 4 R	C 4 4 C	C 4 4 C	10	1

Table 8.1: Breakdown of Morton Order matrix multiplication kernels proposed to outperform the optimal ARM implementation.

In this section we use Hybrid Morton Order layouts to reduce memory latency for the fastest Mali T-628 matrix multiplication kernel proposed by ARM [12], the general form of which we implemented in Section 7.4.2. This kernel will be our baseline and we propose two additional kernels (Morton 2-4 and Morton 4-4) with varying cache miss bulk sizes and cache miss periods. The breakdown of these kernels can be seen in Table 8.1. The visual representation of memory layouts used by all three implementations can be found in Appendix 12.5.

All kernels assume we are transforming the input matrix I as $W \times I = R$. Since deep

neural networks contain multiple chained layers, we need to ensure that the resulting matrix R has the same memory layout as the matrix I, so it can be fed into another transformation using the same kernel.

The Morton 4-2 and Morton 4-4 kernels can be found in Appendices 12.7 and 12.6 respectively.

8.5 Evaluation

As our baseline for this chapter was heavily inspired by the blocked matrix multiplication kernel published by ARM [12] (referred to in their paper as *sgemmNT*), we shall evaluate our baseline and the two proposed kernels against the results ARM published for their kernel. We multiply square matrices of dimensions 96, 192, 384, 768, 1440 and 2880, using workgroup sizes of 4×16, which we established as optimal in Section 7.2. The results of these experiments may be seen in Table 8.2.

Size	ARM paper	Baseline		Morton 4-2		Morton 4-4	
	GFLOPS	GFLOPS	±	GFLOPS	±	GFLOPS	±
96	11.2	11.65	0.05	**13.04**	0.03	12.05	0.03
192	13.0	13.65	0.05	**14.86**	0.01	13.93	0.01
384	13.2	13.08	0.00	**14.19**	0.00	12.94	0.00
768	13.1	12.90	0.00	**13.67**	0.00	12.45	0.00
1440	13.1	12.98	0.00	**13.61**	0.00	12.02	0.00
2880	10.8	2.14	0.23	**13.65**	0.00	11.98	0.00

Table 8.2: Performance of the ARM-proposed Mali T-628 optimised matrix multiplication kernel compared to the performances of kernels using Morton Order memory layouts.

It is clear from these results that our proposed Morton 4-2 kernel outperformed the optimal kernel presented by ARM by **1.44 GFLOPS** on average, which is a **12%** performance increase.

Considering the dimensions of the sliding window, computing array indices in the Morton 4-4 kernel is extremely complicated and error-prone for development. Kernel indices for the Morton 4-2 kernel are much more intuitive, making it a viable candidate for further research.

A comparison of these kernels' performance against kernels from the previous chapter can be seen in Figure 8.3.

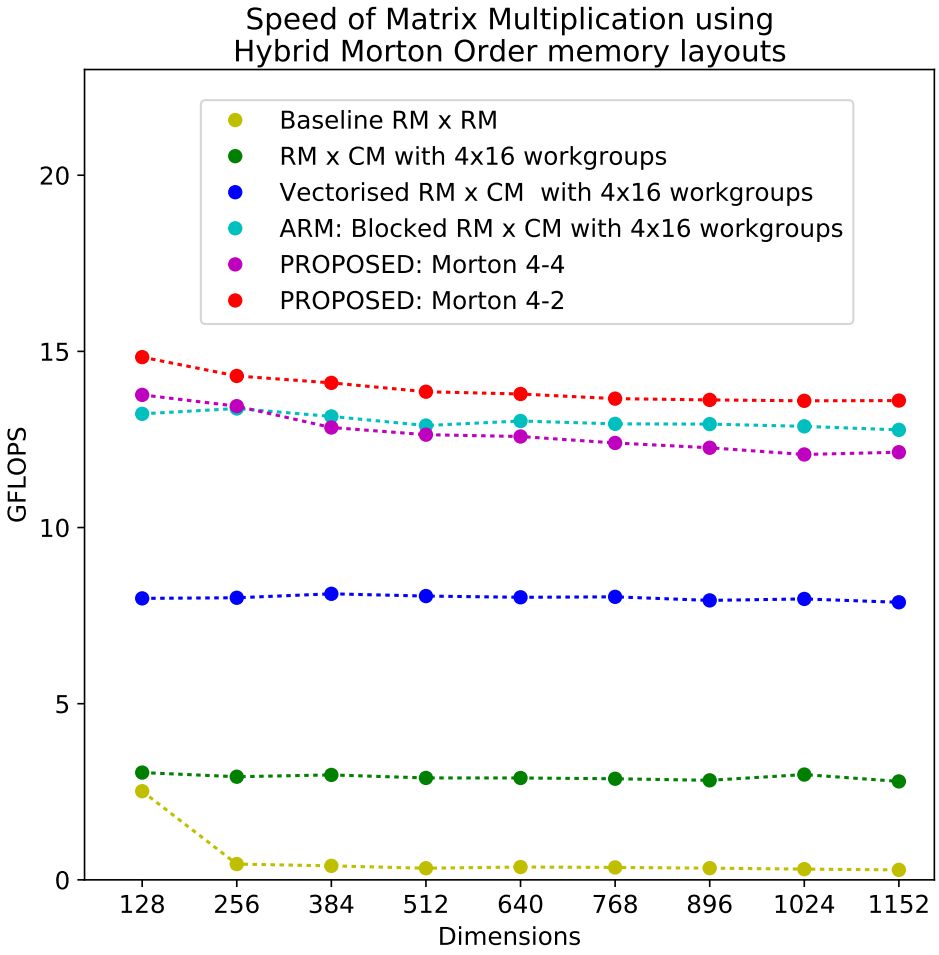

Figure 8.3: Performance of matrix multiplication kernels using Hybrid Morton Order memory layouts compared to kernels from previous chapters.

8.6 Summary

In this chapter we improved the best performing ARM matrix multiplication kernel for the Mali T-628 GPU using Hybrid Morton Order memory layouts. We defined a Context Free Grammar for defining these complex layouts and defined the recursive function used to compute the memory index from row-column coordinates. Performance of two Hybrid Morton Order layout kernels was benchmarked against the results from Johan Gronqvist and Anton Lokhmotov's paper [12], demonstrating a 12% performance gain from using the appropriate layouts.

This concludes optimisation of matrix multiplication and we shall now focus on optimising the remaining kernels required for forward propagation.

Chapter 9

Optimisation of Bias Addition and Activation Functions

Over the course of optimising matrix multiplication in Chapters 7 and 8, we made changes in the memory layouts of the matrices storing intermediate results propagated through the neural network. In doing so, it became necessary to appropriately alter the bias addition kernel as well as the kernels of activation functions. Furthermore, it is likely that the techniques used in previous chapters to optimise matrix multiplication may also be relevant to increasing the performance of the other kernels.

In this chapter we assess the effects of optimisation techniques presented in Chapter 7 and 8 with regards to the bias addition kernel and the activation function kernels.

The respective functionalities of these two kernels fundamentally differ from that of the matrix multiplication algorithm, especially in terms of memory access patterns, which we have shown to be the deciding factor in kernel execution speed. Bias addition accesses two matrices but, unlike the matrix multiplication algorithm, one of these matrices is always one-dimensional (the bias vector). The activation functions on the other hand, accesses only a single matrix. Both kernels store results in the original matrix, unlike the matrix multiplication kernel, which stores results in a third matrix.

In light of these differences, we decided to investigate the effects of previously mentioned optimisation techniques on these kernels. This chapter presents the findings without going into as much depth as previous chapters have, instead, for context, we refer to the respective section where each optimisation technique was first applied to matrix multiplication.

The goal of this section is to choose appropriate variants of the bias addition kernel and activation function kernels to maximise overall classification performance.

9.1 Bias Addition

In this section we present 7 variants of the bias addition kernel and benchmark them against each other. The variants chosen for benchmarking are:

Row Major (RM) – This is the baseline kernel from Section 6.3. Adds a bias vector to a matrix in Row Major layout.

Column Major (CM) – This variant adds a bias vector to a matrix in Column Major layout, similar to the RM variant in all other aspects.

Grouped CM – Identical to the CM variant, however, workgroup sizes are set to 4x16 for spacial locality (see Section 7.2)

Vectorised CM – Identical to the CM variant, yet computes the addition using SIMD vector instructions (see Section 7.3)

Grouped and Vectorised CM – Utilises both workgroup size optimisation and SIMD vector instructions at the same time.

Morton 4-2 – Vectorised bias addition kernel for matrices in `C 4 2 C` layout. Workgroup allocation is automatic.

Grouped Morton 4-2 – Vectorised bias addition kernel for matrices in `C 4 2 C` layout. Workgroup sizes set to 1x16 (in order to process the matrix in square chunks).

9.1.1 Results and Discussion

Figure 9.1 shows the performance of each bias addition variant. Experiments were run on input matrices of sizes 1024 x 1024, 256 x 4096 and 4096 x 256, as these are three matrices of vastly differing shapes, yet identical number of elements, meaning the same amount of floating point operations were required to add the bias to each of these matrices.

It is clear from the graph in Figure 9.1 that grouping work items negatively impacted the performance of both the CM variant and the Vectorised CM variant. Variants CM, Grouped and Vectorised CM, and Grouped Morton 2-4 performed roughly equally.

The best performing kernel is the simple Vectorised CM variant.

Given the performance of the Vectorised CM variant it would be highly recommended to use this variant with Column Major layouts. It is also worth noting that using the Vectorised CM variant is not possible with the best performing matrix multiplication algorithm from Chapter 8, as the memory layouts do not match.

Given the distribution of time spent running each kernel in our baseline breakdown (Figure 6.2), it is clear that matrix multiplication is the primary bottleneck of the execution. As such, it would be unwise to choose the Vectorised CM variant over the

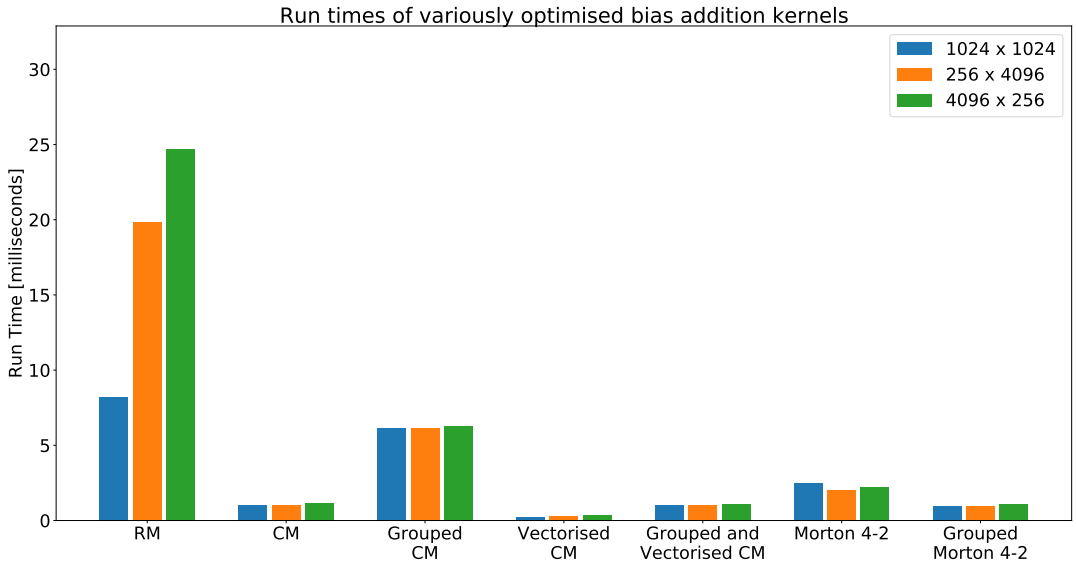

Figure 9.1: Run times of variously optimised bias addition kernels on three different matrices.

Grouped Morton 4-2 variant at the cost of not being able to use the optimal matrix multiplication algorithm.

Given these facts and the results in Figure 9.1, it has been decided that the Grouped Morton 4-2 variant will be used in the final evaluation.

9.2 Activation Functions

In this section we present 5 variants of the sigmoid activation function kernel and benchmark them against each other (later applying the best performing optimisations to ReLU as well). Activation functions do not require the knowledge of row-column coordinates. Specifying a memory layout is hence not necessary, since we may simply map the function to a one-dimensional NDRange covering the entirety of the matrix. Variants designed for 1-dimensional mapping will be marked as ”Unbound”.

It is worth noting that mapping the kernel onto a 2-dimensional set of ids is necessary for implementing memory access pattern optimisation using specified workgroup sizes. This is why some variants do use 2-dimensional id mappings.

The sigmoid function kernel variants we investigated are:

Unbound – Sigmoid kernel simply mapped to the entire matrix (baseline kernel from Section 6.2).

Row Major (RM) – Performing Sigmoid function on a Row Major matrix in columns (strided accesses). Expected to perform the worst.

Column Major (CM) – Performing Sigmoid function on a Column Major matrix in columns (sequential accesses). This enforces ordering very similar to the Unbound variant and is expected to have similar performance.

Grouped CM – Identical to the CM variant, however, workgroup sizes are set to 4x16 for spacial locality (see Section 7.2).

Grouped Morton 4-2 – Sigmoid kernel for matrices in `C 4 2 C` layout. Workgroup sizes set to 4x16 (in order to process the matrix in square chunks).

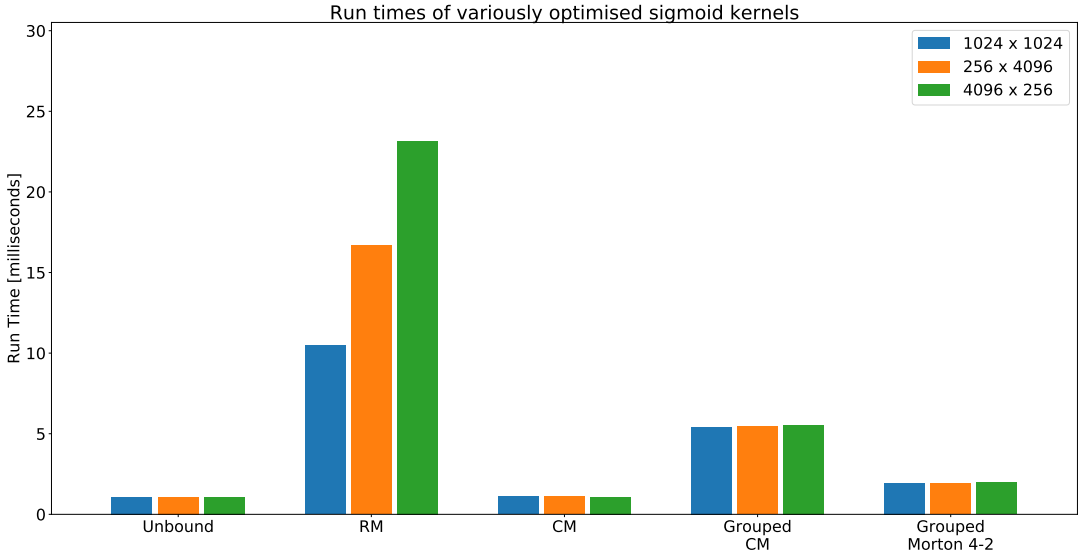

Figure 9.2: Run times of variously optimised sigmoid kernels on three different matrices.

9.2.1 Results and Discussion

Similarly to bias addition, the sigmoid kernels do not benefit from grouping, as seen in Figure 9.2. In the case of sigmoid kernels, however, the reason behind this is more intuitive than for bias addition. Default OpenCL grouping of work items increments the least significant kenrel id, which, in the case of a single-dimensional mapping, is just the single id. Thus for the sigmoid function which only requires one matrix, the automatic OpenCL grouping already exploits spacial locality as best as can be done, since it sequentially traverses the space of ids which map directly onto the indices of the matrix. This makes the Unbounded variant optimal.

Conveniently, as the Unbounded variant does not depend on specific memory layouts, we may use it together with the Morton 4-2 layout required by the optimal matrix multiplication algorithm from Chapter 8.

9.3 Summary

In this section we presented multiple variants of bias addition and activation function kernels. Investigation into their performance allowed us to choose appropriate variants for use in our neural network execution system such as to maximise performance of these kernels as well as to preserve the optimal performance of matrix multiplication from previous sections.

Our chosen variants were the Grouped Morton 4-2 bias addition kernel (see Section 9.1) and the Unbound approach to activation function kernels (see Section 9.2). The respective kernels for Morton 4-2 bias addition, sigmoid function and ReLU function can be found in Appendix 12.9.

Chapter 10

Evaluation

In this section we evaluate our best performing system against other publicly available neural network frameworks. We do this by selecting a series of neural networks from literature, recreating the network structures and using the systems to forward propagate values over it.

For this task we chose the series of network models by Ikuro Sato et.al. [35] for classifying the CIFAR [36] picture set and the MNIST [37] database of handwritten digits. For both datasets they propose two models: a deep convolutional one and a shallow fully connected one. We shall compare performance on all four models.

It is important to point out that our system does not support convolution at this moment, meaning we cannot execute the convolutional models in their entirety. The two convolutional models proposed by Ikuro Sato et.al. [35] do however, contain a series of fully connected layers following all convolution, meaning we are able to benchmark our system against other systems for this subsection of the convolutional models. Seeing that convolution is going to be added to the system in the second part of the project, we have decided to include these partial comparisons into our work to give us a rough idea of how well the current system performs against the problems it will ultimately be used for when it is completed.

The fully-connected models on the other hand, have been evaluated in their entirety.

The paper used these models to forward propagate feature vectors in batches of 100 and our evaluation will adhere to this as well. For convenience, we shall refer to the fully-connected MNIST model, fully-connected CIFAR model, convolutional MNIST model and convolutional CIFAR model as **Full MNIST**, **Full CIFAR**, **Partial MNIST** and **Partial CIFAR** respectively. The following tables contain a breakdown of each model's layer structure, keeping only the final fully-connected stages of the two deep convolutional models.

Layer	Input Dimensionality	Output Dimensionality	Activation Function
1	784	2500	ReLU
2	2500	2000	ReLU
3	2000	10	

Table 10.1: Layer breakdown of the fully connected MNIST model (Full MNIST)

Layer	Input Dimensionality	Output Dimensionality	Activation Function
1	3072	4096	ReLU
2	4096	3072	ReLU
3	3072	10	

Table 10.2: Layer breakdown of the fully connected CIFAR model (Full CIFAR)

Layer	Input Dimensionality	Output Dimensionality	Activation Function
1	640	150	ReLU
2	150	10	

Table 10.3: Layer breakdown of the final stages of the convolutional MNIST model (Partial MNIST)

Layer	Input Dimensionality	Output Dimensionality	Activation Function
1	256	128	ReLU
2	128	10	

Table 10.4: Layer breakdown of the final stages of the convolutional CIFAR model (Partial CIFAR)

10.1 TensorFlow

In order to gain insight into how well our system performs compared to other GPU systems, we chose to compare its performance on the four models to that of the popular neural network framework, TensorFlow [19]. Considering TensorFlow is written for the CUDA architecture and as such we needed to execute the models on a different GPU, we cannot objectively compare the execution times of these experiments to our system. Instead we shall compare the systems in terms of **GPU Utilisation**.

To do this, we compute the total amount of floating point operations required to forward propagate the inputs over the neural network, and then compute the GPU utilisation in terms of the time taken to propagate and the maximum attainable performance of the GPU in GFLOPS as:

$$GPU\ Utilisation = \frac{\frac{floating\ point\ operations\ computed}{time\ taken\ to\ propagate}}{Maximum\ floating\ point\ operations\ per\ second}$$

This formula yields a performance value in the interval [0, 1], which we may use to compare the performance.

Timing individual kernels in our system was done using the built-in OpenCL profiler tools. Timing TensorFlow kernels was slightly more complicated and required the use of TensorFlow's profiler when running our model in the TensorFlow session, using the `options=tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)` argument. Google Chrome provides a sensible visualisation tool for the profiling results and we used this tool to extract the run times of the appropriate kernels from the profiler trace file.

We know the maximum performance of the Mali T-628 GPU is 17 GFLOPS per core from Section 2.1. We perform our experiments on the NVIDIA GTX960M, which has the maximum performance of approximately 1400 GFLOPS [1].

10.1.1 Results and Discussion

The results in Figure 10.1 clearly show that our system's level of GPU utilisation remained constant throughout all experiments. For the convolutional models our system outperforms TensorFlow by a small margin. For the fully connected models, however, TensorFlow's utilisation of the GPU went up, while our system's remained constant.

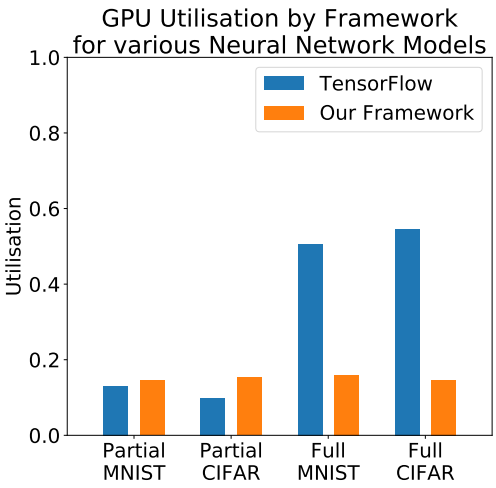

Figure 10.1: GPU utilisation of our system compared to that of TensorFlow.

Given our project aims to focus on deep neural networks in the second part, and our system's GPU utilisation for these models being comparable to TensorFlow's, we may consider our system adequate for this task for these models.

Future research may look into improving matrix multiplication for large matrices. We have seen a decrease in performance with increasing matrix sizes in every experiment and the matrices from both fully-connected models are larger than any we have experimented on so far.

10.2 Numpy

The goal of our project is to optimise neural network execution on the Odroid XU3 board, which contains both a CPU and a GPU. In this section we compare our GPU neural network execution performance against the performance of a CPU optimised

[1] Finding this performance metric was a difficult task, as NVIDIA does not explicitly provide it. It was computed as 2 (single precission) times the number of parallel threads times the clock rate. The value was cross-checked with multiple sources to give us a reliable result [38] [39] [40] [41]

system in an attempt to establish a clear performance gain in using our GPU framework over a CPU one.

As our CPU implementation we selected Numpy [42], which is a Python package for scientific computing. It provides a powerful N-dimensional array object with CPU optimisations for mathematical operations including matrix multiplication, bias addition and both activation functions we implemented.

We shall run the same experiments using Numpy on the Odroid-XU3 board as we ran for TensorFlow, this time comparing the raw execution time to that of our OpenCL implementation.

10.2.1 Results and Discussion

Model	CPU time	GPU time	Speedup
Partial MNIST	38.14	1.99	19.20
Partial CIFAR	11.20	0.66	16.99
Full MNIST	3705.50	129.33	28.65
Full CIFAR	19673.92	506.15	38.87

Table 10.5: Run times of each model in milliseconds for both the Numpy CPU implementation and our Mali T-628 GPU implementation. The speedup factor shows the attainable speedup by using the GPU implementation over the CPU one.

Table 10.5 shows how our OpenCL implementation outperforms the Numpy CPU implementation by an order of magnitude, with the largest speedup by a factor of 38 observed for the fully-connected CIFAR model. The observed performance gains on smaller fully-connected layers from the ends of both convolutional models were slightly smaller, yet still substantial.

10.3 Summary

In this chapter we compared the performance of our Mali T-628 optimised neural network execution system against other systems. Our system successfully demonstrated a level of GPU Utilisation comparable to that of TensorFlow for convolutional models, yet TensorFlow greatly outperformed our system for fully-connected models. With regards to the CPU on the Ordoid XU3 board, our system's execution time was substantially smaller than that of the CPU optimised Numpy package for all models.

Chapter 11

Conclusion

The goal of this work was to design a neural network execution system optimised for the Odroid-XU3 board. Building on top of the Khronos group C++ OpenCL bindings, we wrote a high-level OpenCL framework using which we constructed a neural network execution system. Our final system is capable of executing fully-connected neural network models on the Odroid board using the available Mali T-628 GPU.

11.1 Critical Analysis

The main problem undertaken in this research was optimising matrix multiplication on the GPU, as this is the most computationally expensive operation out of those required for our system. We built upon research published by ARM concerning optimisation of matrix multiplication kernels on the Mali T-628 and we present a method of improving the best performing ARM kernel using Hybrid Morton Order memory layouts. Our best performing matrix multiplication kernel outperforms the best kernel from ARM by 12%.

The final product demonstrated similar performance and GPU utilisation to the TensorFlow framework for fully-connected layers in convolutional models, yet TensorFlow performed better for larger fully-connected models. Within the scope of the Odroid-XU3 board, our system significantly outperforms Numpy (a CPU alternative), making it the system of choice on the device.

11.2 Future Work

The current system is capable of executing fully-connected neural networks with ReLU and Sigmoid activation functions. There is a number of additional operations frequently used in deep neural networks such as convolution, batch normalisation and pooling. Future work will focus on implementing and optimising these operations in order to make the system fully capable of executing a majority of deep neural networks.

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[2] A. Graves, A. rahman Mohamed, and G. E. Hinton, "Speech recognition with deep recurrent neural networks," *CoRR*, vol. abs/1303.5778, 2013. [Online]. Available: http://arxiv.org/abs/1303.5778

[3] Y. Wu, MSc, M. L. G. PhD, K. D. PhD, C. J. V. M. PhD, R. A. S. MD, and C. E. M. PhD, "Artificial neural networks in mammography: Application to decision making in the diagnosis of breast cancer." [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.122.5805&rep=rep1&type=pdf

[4] K. Hinum, "Smartphone and tablet graphics cards - benchmark list and comparison." [Online]. Available: http://www.notebookcheck.net/Smartphone-Graphics-Cards-Benchmark-List.149363.0.html

[5] "Opencl support," GitHub. [Online]. Available: https://github.com/tensorflow/tensorflow/issues/22

[6] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2010.69

[7] "javacl," Google. [Online]. Available: https://code.google.com/archive/p/javacl/

[8] A. Klockner, "Pyopencl." [Online]. Available: https://mathema.tician.de/software/pyopencl/

[9] J. G. M. Benedict R. Gaster, "Embedding opencl in ghc haskell." [Online]. Available: http://benedictgaster.org/wp-content/uploads/2013/01/embedding.pdf

[10] B. R. Gaster, "The opencl c++ wrapper api," KHRONOS GROUP. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.1.pdf

[11] H. F. Karthik Hariharakrishnan, Anthony Barbier, "Opencl on mali faqs," 2013. [Online]. Available: http://malideveloper.arm.com/downloads/OpenCL_FAQ.pdf

[12] A. L. Johan Gronqvist, "Optimising opencl kernels for the arm mali-t600 gpus," ARM. [Online]. Available: http://malideveloper.arm.com/downloads/GPU_Pro_5/GronqvistLokhmotov_white_paper.pdf

[13] P. Bialas and A. Strzelecki, "Benchmarking the cost of thread divergence in cuda," NVIDIA, 2015. [Online]. Available: https://arxiv.org/pdf/1504.01650.pdf

[14] C. Adeniyi-Jones, "Optimal compute on arm mali gpus," ARM. [Online]. Available: http://www.cs.bris.ac.uk/home/simonm/montblanc/OpenCL_on_Mali.pdf

[15] F. Viegas, G. Andrade, J. Almeida, R. Ferreira, M. Gonçalves, G. Ramos, and L. Rocha, "Gpu-nb: A fast cuda-based implementation of naive bayes," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2013 25th International Symposium on*. IEEE, 2013, pp. 168–175.

[16] R. Farivar, D. Rebolledo, E. Chan, and R. H. Campbell, "A parallel implementation of k-means clustering on gpus." in *Pdpta*, vol. 13, no. 2, 2008, pp. 212–312.

[17] T. Sharp, *Implementing Decision Trees and Forests on a GPU*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 595–608. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88693-8_44

[18] ——, "Implementing decision trees and forests on a gpu," in *ECCV (4)*, vol. 5305. Springer, January 2008, pp. 595–608. [Online]. Available: https://www.microsoft.com/en-us/research/publication/implementing-decision-trees-and-forests-on-a-gpu/

[19] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[20] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[21] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[22] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[23] "Opencl-caffe." [Online]. Available: https://github.com/amd/OpenCL-caffe

[24] P. J. R. Anthony E. Nocentino, "Optimizing memory access on gpus using morton order indexing." [Online]. Available: https://john.cs.olemiss.edu/~rhodes/papers/Nocentino10.pdf

[25] J. Siegel, J. Ributzka, and X. Li, "Cuda memory optimizations for large data-structures in the gravit simulator," *Journal of Algorithms & Computational Technology*, vol. 5, no. 2, pp. 341–362, 2011.

[26] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing memory access patterns for heterogeneous systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 13:1–13:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063401

[27] I. Grasso, P. Radojkovi, N. Rajovi, I. Gelado, and A. Ramirez, "Energy efficient hpc on embedded socs: Optimization techniques for mali gpu," Barcelona Supercomputing Center, 2009. [Online]. Available: http://www.petarradojkovic.com/publications/IPDPS-2014_Grasso.pdf

[28] S. S. Latifi Oskouei, H. Golestani, M. Hashemi, and S. Ghiasi, "Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android," in *Proceedings of the 2016 ACM on Multimedia Conference*, ser. MM '16, 2016, pp. 1201–1205.

[29] "Renderscript." [Online]. Available: https://developer.android.com/guide/topics/renderscript/compute.html

[30] N. Lohmann, "Json for modern c++." [Online]. Available: https://github.com/nlohmann/json

[31] Y. Lecun and C. Cortes, "The MNIST database of handwritten digits." [Online]. Available: http://yann.lecun.com/exdb/mnist/

[32] S. S. Junjie Li, Sanjay Ranka, "Strassens matrix multiplication on gpus," University of Florida. [Online]. Available: https://www.cise.ufl.edu/~sahni/papers/strassen.pdf

[33] *Mali-T600 Series GPU OpenCL Version 1.1.0 Developer Guide*, ARM, 2012. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dui0538e/DUI0538E_mali_t600_opencl_dg.pdf

[34] P. H. J. K. Jeyarajan Thiyagalingam, Olav Beckmann, "Is morton layout competitive for large two-dimensional arrays, yet?" Imperial College. [Online]. Available: http://www.doc.ic.ac.uk/~ob3/Publications/CandCPEMorton2004.pdf

[35] I. Sato, H. Nishimura, and K. Yokoi, "APAC: augmented pattern classification with neural networks," *CoRR*, vol. abs/1505.03229, 2015. [Online]. Available: http://arxiv.org/abs/1505.03229

[36] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[37] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[38] "Nvidia geforce gtx 960 (oem) specifications page." [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-960-oem/specifications

[39] "Nvidia geforce gtx 960 specifications page." [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-960/specifications

[40] "Geforce gtx 960m." [Online]. Available: http://gpuboss.com/graphics-card/GeForce-GTX-960M

[41] J. Walton, "Nvidia launches gtx 960m/950m and geforce 940m/930m/920m." [Online]. Available: http://www.anandtech.com/show/9077/nvidia-launches-gtx-960m950m-and-geforce-940m930m920m

[42] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *CoRR*, vol. abs/1102.1523, 2011. [Online]. Available: http://arxiv.org/abs/1102.1523

Chapter 12

Appendix

12.1 CLManager Interface

```
1  // Read a library of OpenCL code
2  void readLibrary(const std::string& path);
3
4  // Compile the current library
5  int compileLibrary();
6
7  // Allocate memory on the device
8  cl::Buffer * createBuffer(size_t size);
9
10 // Deallocate memory on the device
11 void deleteBuffer(cl::Buffer * buffer);
12
13 // Verify that the memory is still valid
14 bool verify(cl::Buffer * buffer);
15
16 // Choose a different device
17 void setDevice(int computeUnits);
18
19 // Get an instance of DynamicCL::Kernel for the
20 // given codename
21 inline Kernel getKernel(const std::string& codename);
```

Source 12.1: Primary methods in the `CLManager` class

12.2 Blocked RM $\times$ RM Matrix Multiplication Kernel

```
1  kernel void
2  blockedRMRM(uint l, uint w,
3    global float4 * const A,
4    global float4 * const B,
5    global float4 *C)
6  {
7    uint i = get_global_id(0);
8    uint j = get_global_id(1);
9    uint nv4 = l >> 2;
10   float4 accum = (float4) 0.0;
11   for (uint k = 0; k < nv4; ++k)
12   {
13     float4 a = A[i * nv4 + k];
14     float4 b0 = B[((k << 2) + 0)* (w >> 2) + j];
15     float4 b1 = B[((k << 2) + 1)* (w >> 2) + j];
16     float4 b2 = B[((k << 2) + 2)* (w >> 2) + j];
17     float4 b3 = B[((k << 2) + 3)* (w >> 2) + j];
18     accum += a.s0 * b0 + a.s1 * b1 + a.s2 * b2 + a.s3 * b3;
19   }
20   C[i * (w>>2) + j] = accum;
21 }
```

Source 12.2: Blocked matrix multiplication kernel for RM $\times$ RM matrices.

12.3 Blocked RM × CM Matrix Multiplication Kernel

```
1  kernel void
2  blockedRMCM(uint l, uint h,
3    global float4 * const A,
4    global float4 * const B,
5    global float2 *C)
6  {
7    uint i = get_global_id(0);
8    uint j = get_global_id(1);
9    uint nv4 = l >> 2;
10   float4 ab = (float4) 0.0;
11   for (uint k = 0; k < nv4; ++k)
12   {
13     float4 a0 = A[2 * i * nv4 + k];
14     float4 a1 = A[(2 * i + 1)* nv4 + k];
15     float4 b0 = B[2 * j * nv4 + k];
16     float4 b1 = B[(2 * j + 1)* nv4 + k];
17     ab += (float4)(dot(a0, b0), dot(a1, b0),
18               dot(a0, b1), dot(a1, b1));
19   }
20   uint ix = 2 * j * (h >> 1) + i;
21   C[ix] = ab.s01;
22   C[ix + (h >> 1)] = ab.s23;
23 }
```

Source 12.3: Blocked matrix multiplication kernel for RM × CM matrices.

12.4 Order of workgroup computation

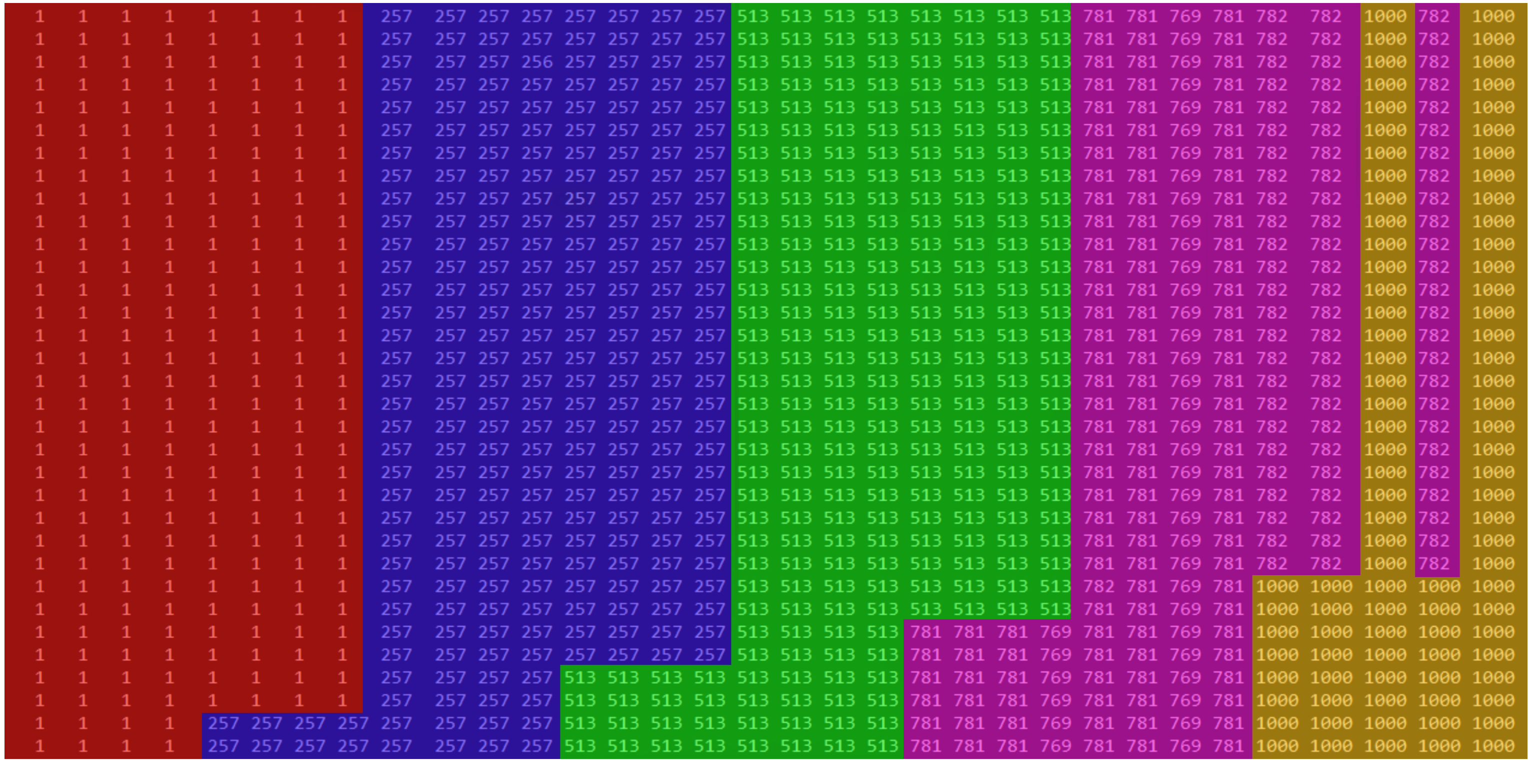

Figure 12.1: Pattern in which workgroups are executed. Global ids 0 and 1 were the row and column indices respectively. Kernel counted the amount of non-zero entries in the matrix and saved that number into its respective entry of the matrix (causing subsequent entries to see more non-zero entries). Figure shows the resulting matrix.

12.5 Complex Hybrid Morton Order Memory Layouts

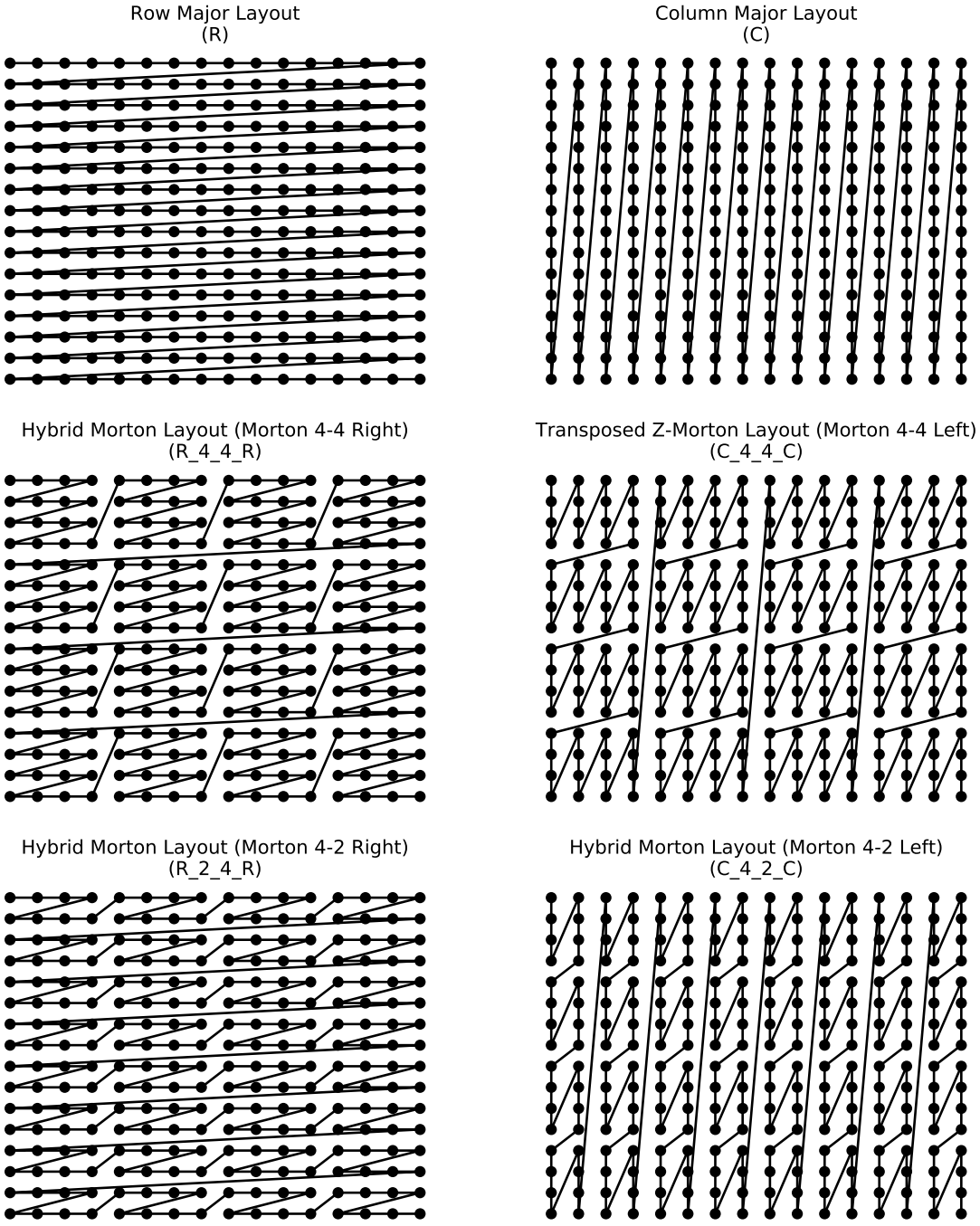

Figure 12.2: Variety of memory layouts for comparison and visualisation purposes. All are supported by the system.

12.6 Morton 4-4 Kernel

```
1  __kernel void
2  morton_4_4(uint l, uint h, uint w,
3      global float4 * const A,
4      global float4 * const B,
5      global float2 *C)
6  {
7    uint row = get_global_id(0);
8    uint col = get_global_id(1);
9
10   uint rowChunk = row / 2;
11   uint colChunk = col / 2;
12
13   float4 ab = (float4) 0.0;
14   for (uint k = 0; k < l; k += 4)
15   {
16     float4 a0 = A[rowChunk * l + k + ((row*2) & 0b11)];
17     float4 a1 = A[rowChunk * l + k + ((row*2) & 0b11) + 1];
18     float4 b0 = B[colChunk * l + k + ((col*2) & 0b11)];
19     float4 b1 = B[colChunk * l + k + ((col*2) & 0b11) + 1];
20     ab += (float4)(dot(a0, b0), dot(a1, b0),
21               dot(a0, b1), dot(a1, b1));
22   }
23   uint ix = (h * 4 / 2) * colChunk + rowChunk * 8
24           + ((col * 2) & 0b11) * 2 + (row & 0b1);
25   C[ix] = ab.s01;
26   C[ix + 2] = ab.s23;
27 }
```

Source 12.4: Blocked matrix multiplication kernel for `R 4 4 R` × `C 4 4 C` matrices.

12.7 Morton 4-2 Kernel

```
1  __kernel void
2  morton_4_2(uint l, uint h,
3      global float4 * const A,
4      global float4 * const B,
5      global float2 *C)
6  {
7    uint row = get_global_id(0);
8    uint col = get_global_id(1);
9
10   uint rowChunk = row / 2;
11   uint colChunk = col / 2;
12
13   float4 ab = (float4) 0.0;
14   for (uint k = 0; k < l/4; k++)
15   {
16     float4 a0 = A[row*l/2 + 2*k];
17     float4 a1 = A[row*l/2 + 2*k + 1];
18     float4 b0 = B[col*l/2 + 2*k];
19     float4 b1 = B[col*l/2 + 2*k + 1];
20
21     ab += (float4)(dot(a0, b0), dot(a1, b0),
22               dot(a0, b1), dot(a1, b1));
23
24   }
25   uint ix = (h/4)*col*4 + (row/2)*4 + (row & 0b1);
26
27   C[ix] = ab.s01;
28   C[ix + 2] = ab.s23;
29 }
```

Source 12.5: Blocked matrix multiplication kernel for `R 2 4 R` × `C 4 2 C` matrices.

12.8 Memory Access Pattern Notation

In this work we use the following access pattern notation:

$$\overset{1}{\underset{i=0}{\uparrow}} \left(\overset{1}{\underset{j=0}{\uparrow}} \left(operation(i,j) \right) \right)$$

This notation conveys the relative order in which the inner operation is called. This operation may be an instruction, memory access, a function call or anything else. In this particular example, the following calls to the `operation` function are going to be made, in this exact order:

1. operation(0,0)
2. operation(0,1)
3. operation(1,0)
4. operation(1,1)

12.9 Other Kernels

12.9.1 Morton 4-2 Bias Addition Kernel

```
__kernel void colAddMorton44(uint rows, uint cols,
    global HON_DATA_TYPE * mat,
    global HON_DATA_TYPE * column
    ) {
  size_t row = get_global_id(0);
  size_t col = get_global_id(1);

  uint col4 = col >> 2;
  uint row4 = row >> 2;
  uint row1 = row & 0b11;
  uint col1 = col & 0b11;

  uint index = col4 * 4 * rows + row4 * 16 + col1 * 4 + row1;
  mat[index] += column[row];
}
```

Source 12.6: Final bias addition kernel for `C 4 2 C` matrices.

12.9.2 Sigmoid Kernel

```
__kernel void sigmoid(global HON_DATA_TYPE * mat) {
  size_t index = get_global_id(0);
  mat[index] = 1.0 / (1 + exp(-mat[index]));
}
```

Source 12.7: Final ReLU kernel.

12.9.3 ReLU Kernel

```
__kernel void sigmoid(global HON_DATA_TYPE * mat) {
  size_t index = get_global_id(0);
  mat[index] = 1.0 / (1 + exp(-mat[index]));
}
```

Source 12.8: Final sigmoid kernel.