Optimising Neural Network Inference on Low-Powered GPUs

Simon Rovder

Master of Informatics

School of Informatics
University of Edinburgh

2018

Abstract

This work presents effective optimisation techniques for accelerating neural network inference on low-powered heterogeneous devices with OpenCL. Using LeNet and VGG-16 as test networks, we implement a custom neural network system in OpenCL and optimise it to minimise their inference times.

We outline two methods for fast convolution: an iterative vectorised approach and a Morton GEMM based approach. The two approaches demonstrate VGG-16 inference speeds up to three times faster than current state-of-the-art systems and outperform other custom neural network systems by speedup factors of up to 1.42.

Acknowledgements

I would like to thank my supervisor Michael O'Boyle for his guidance, advice and support throughout this project.

Special thanks also goes to Jose Cano for providing valuable insight into related works.

Finally, I would like to thank Constance Crowe, Paul Sinclair, James Renwick and William Mathewson for proofreading my work and providing helpful feedback.

Table of Contents

Chapter 1

Introduction

Neural networks are currently at the forefront of the field of machine learning. They have shown incredible versatility across a large spectrum of problems ranging from image and sound recognition all the way to natural language processing [1, 2, 3]. This makes them a desirable tool for all platforms and devices.

The primary driving force behind recent neural network research is the progress in development of Graphics Processing Units (GPUs). Until recently, neural networks were too computationally complex to be trained or executed in reasonable amounts of time. GPUs have enabled fast training on affordable hardware and thus brought neural networks into the research spotlight.

The rapidly growing popularity of smartphones has attracted GPUs to the world of battery powered devices, opening the doors to utilising neural networks on these devices as well.

The main issue faced when attempting to execute neural networks on these low-powered GPUs is a noticeable lack of software support. Considerable amounts of research and development has been dedicated to training complex neural network models on desktop GPUs and CPUs, resulting in numerous neural network frameworks like Tensorflow [4], Caffe [5], Torch [6] or Theano [7]. These frameworks, however, are either not supported on the low-powered devices or are not optimised for the architectural differences between desktop GPUs and low-powered GPUs.

In this work, we address these issues by investigating recent research into neural network execution on low-powered GPUs, and by benchmarking state-of-the art solutions on the Mali T-628 GPU. We take our research a step further by implementing custom OpenCL kernels for neural network execution to show how exploiting detailed knowledge of the device architecture can be used to outperform state-of-the-art automatically tuned systems.

The overarching goal of this work is to execute neural networks effectively on the Mali T-628 GPU, though the specific goals are further refined in Section 1.2.

1.1 Motivation

There are considerable benefits to correctly utilising the GPUs on mobile devices. First of all, it is worth noting that neural networks are already used on mobile devices, albeit not always in an efficient way. Some implementations will execute small models directly on a smartphone CPU while other implementations make use of deep learning APIs to offload the computation to a cloud solution (a notable example of this being the Google Cloud Vision API [8]). The former approach unnecessarily wastes CPU time especially if an unused GPU is present on the device, while the latter consumes bandwidth, since it requires sending data to a remote server for processing.

Having a neural network execution system which is well optimised for low-powered GPUs would enable alternative solutions. **In an ideal case, one could pre-train a model using powerful hardware and then deploy this model to low-powered devices**, which could execute it on their own embedded GPUs. This would eliminate the need for CPU execution, freeing the CPU for other processes to use. It would also allow more complex models to run on the device, meaning cloud execution would only be needed for extremely complex models.

Such an approach would open doors for any embedded system to utilise neural models. It would also enable using machine learning to customise the behavior of a device on the fly, which is already a highly desirable feature in smartphone user experience optimisation, and one which we can expect to become more relevant with the growth of the Internet of Things.

1.2 Goal Outline

The primary goal of this work is to evaluate the viability of CLBlast (an automatically-tuned deep-learning-enabled library further discussed in Section 3.2) for executing deep neural networks on the Mali T-628 GPU, investigate the shortcomings and limitations of this library, and attempt to improve upon its performance by hand-crafting kernels for the task.

Our hypothesis is that the CLBlast automatic tuning process does not find optimal configurations for the device because it cannot exploit detailed knowledge about the architecture. Hence it follows that tailoring kernels specifically to the Mali GPU will outperform CLBlast on all benchmarks.

To compare CLBlast with our custom-made system, we will benchmark the two systems on executing LeNet [9] (one of the very first convolutional models ever published) and VGG-16 [10] (one of the state-of-the-art models for image classification).

These two models were chosen to allow us to benchmark the systems against two scenarios: propagating a batch of 100 inputs over a small network (LeNet), and propagating a single input over a large network (VGG-16). The two scenarios are similarly challenging from a computational perspective, yet architecturally differ enough to potentially require different optimisations, which is why we chose to cover both.

Layer	LeNet	VGG-16	CLBlast Support
Fully-connected	✓	✓	✓
ReLU		✓	
Sigmoid	✓		
Convolutional	✓	✓	✓
Max-pooling		✓	✓
Subsampling	✓		✓

Table 1.1: Layer requirements of LeNet and VGG-16 compared to the capabilities of CLBlast

Table 1.1 outlines the functional requirements of LeNet and VGG-16, as well as the functionality supported by CLBlast.

The refined goals of this work are the following:

1. Use CLBlast subroutines to implement as many layers specified in Table 1.1 as possible given the library's range of support.
2. Use the CLBlast implementation to forward propagate 100 inputs over LeNet and a single input over VGG-16, investigating performance and behavior.
3. Implement all functionality required for layers specified in Table 1.1 in custom OpenCL kernels.
4. Use the custom implementation to forward propagate 100 inputs over LeNet and a single input over VGG-16, investigating performance and behavior.
5. Optimise the custom implementation to outperform CLBlast on all benchmarks.

1.3 Contributions

A high level overview of our contributions to neural network execution on the Mali T-628 GPU includes:

- Presenting a critical review of related and competing technologies
- Designing a custom system for executing neural networks
- Optimising the custom system to outperform other state-of-the-art systems

Our contributions in more detail are:

- We present a critical review of CLBlast and research done by Loukadakis et al. into neural network kernels optimised for low powered GPUs
- We compile, autotune and benchmark CLBlast subroutines relevant to neural network execution and use the subroutines to construct neural network layers
- We benchmark the CLBlast layer implementation by forward propagating 100 inputs over LeNet and 1 input over VGG-16 on the Mali T-628

- We implement a custom neural network system by hand crafting OpenCL kernels and optimise them for the GPU
- We demonstrate a 17x speed increase over LeNet with CLBlast, a 3x speed increase over VGG-16 with CLBlast, and a 1.42x speed increase over research done by Loukadakis et al.

1.4 Outline

Chapter 1: Introduction provides a brief introduction to the topic at hand, motivation for the work, and content of the report. The goal outline and contributions are also presented in this chapter.

Chapter 2: Background and Terminology provides a very condensed overview of relevant technologies and neural networks. We describe the OpenCL paradigm and provide some basic terminology for both OpenCL and neural networks with the goal of disambiguating and clarifying content in subsequent chapters.

Chapter 3: Related Work presents the most recent research into the topic at hand. We enumerate other state-of-the-art systems and results from competing research, providing a critical evaluation of the implementations. We also give a brief summary of our previous work of which this is a continuation.

Chapter 4: CLBlast on the ODROID Board delves into the CLBlast library. We compile the library, use the build-time autotuning procedure to tune it to the Mali GPU on the ODROID XU3 board, and benchmark the library subroutines relevant to neural networks. This chapter also outlines the library's key fundamental limitations, knowledge of which will guide our implementation in the following chapters.

Chapter 5: Executing LeNet and VGG-16 with CLBlast describes the process of implementing neural network layers using CLBlast subroutines benchmarked in the previous chapter. We benchmark the implemented neural network evaluation system on LeNet and VGG-16, investigating any other limitations of the library.

Chapter 6: Executing LeNet with Custom System outlines our process of designing a custom OpenCL baseline neural network system and compares its performance to CLBlast using LeNet. CLBlast is outperformed by our baseline by a speedup factor of 17x.

Chapter 7: Executing VGG-16 with Custom System contains the most important results gathered in this research. We adapt the baseline to VGG-16 and optimise it using two different approaches. We benchmark the two optimised systems against CLBlast and research by Loukadakis et al. [11], **outperforming them with speedups of 3x and 1.42x respectively**.

Chapter 8: Conclusion reiterates our goals, referencing parts of the work where they were accomplished, reevaluates our contributions, and outlines potential further improvements.

Chapter 2

Background and Terminology

This chapter aims to introduce the fundamental concepts and technology used throughout this work. We focus here on OpenCL and neural networks, briefly discussing what they are and how they work, providing definitions for the basic terminology. If familiar with OpenCL and neural networks, this section can be skipped without missing any integral part of our research.

2.1 OpenCL

OpenCL [12] is a framework for writing and executing code in parallel across one or more heterogeneous platforms, and in this work we use it to offload the computationally expensive neural network operations onto the Mali GPU. The OpenCL paradigm is designed around splitting a program into *host-code* and *device-code*.

Host-code is the part of the program intended to run on the CPU. This part can be written in a complex high-level object oriented language and it runs under the same conditions as any other program we run in every-day computer use. It can be written in any language for which OpenCL bindings are available. [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. Device-code is usually very simple and is designed to perform one particular task very quickly. The OpenCL code can be stored in strings or in separate files, from where it will be retrieved and passed to OpenCL by host-code.

The OpenCL paradigm is based on taking loop-based algorithms and parallelising their execution across a subset of the loops. In the case of nested loops, one may safely parallelise any subset of the loops as long as no dependency criteria are violated, which could lead to memory race conditions. In some cases, even violating the

[1]Bindings have been written for many programming languages, the most popular of which include Java [13], C++ [14], Python [15] and Haskell [16]. In our work we use Python and C++ bindings.

dependency criteria can be safely done if memory race conditions are handled using memory fences, though we do not use fencing in this work [2].

When discussing parallelisation of loops we will often refer to what is known as an *iteration vector*. An iteration vector defines the particular combination of values in all the loop variables at the innermost loop. In other words, a set of all possible iteration vectors of an algorithm defines all the combinations of loop variables for which the innermost loop will be executed. Dependencies between loops can then be detected and expressed in terms of *direction vectors*, which restrict the parallelisation flexibility of an algorithm. These concepts are often used in the field of compilers to optimise loops, yet they come in handy when analysing GPU parallelisation as well.

We provide a diagram of the OpenCL paradigm in Figure 2.1 and define the basic terminology:

OpenCL Device – An OpenCL compliant device capable of executing OpenCL code. The Mali T-628 GPU is one such device. OpenCL devices are not restricted to GPUs and also include integrated graphics devices, co-processors, even CPUs themselves. An OpenCL device will contain one or more *Compute Units*.

Compute Unit (or Core) – A hardware component within an OpenCL device capable of executing a *kernel* on a *workgroup*. An OpenCL device will usually contain one or more compute units.

Kernel – A single function within the device-code. Their function signature contains the `__kernel` keyword, which indicates to OpenCL that the function is a potential point of entry for execution on the GPU. Host-code will enqueue the kernel for parallel execution on the OpenCL device.

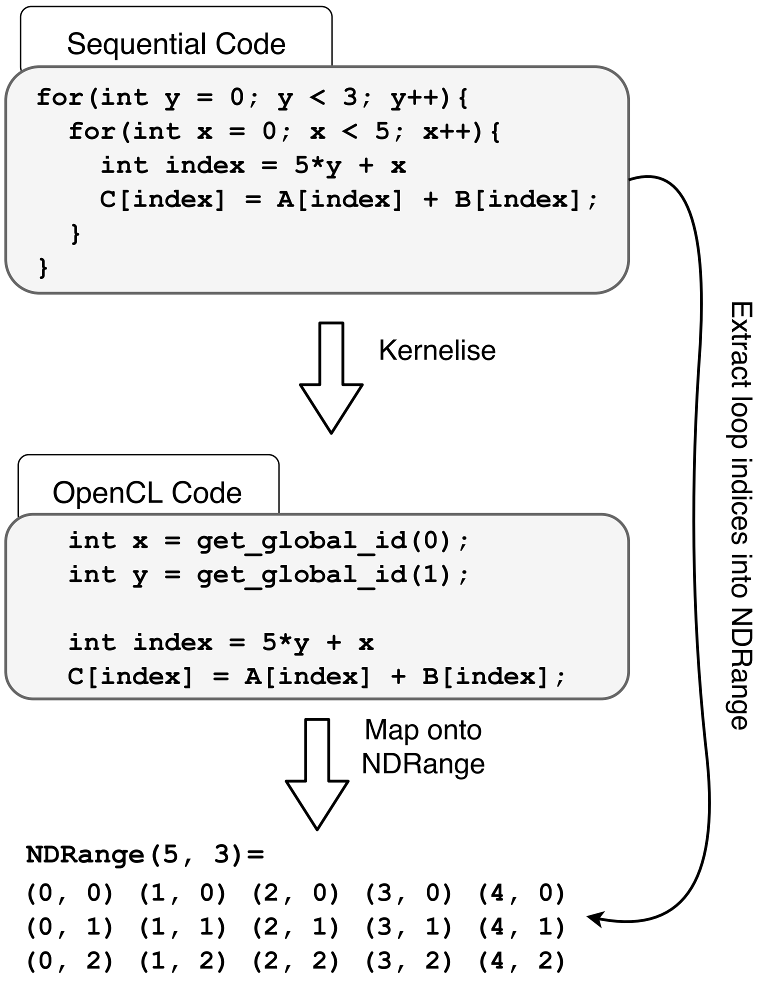
NDRange (N-Dimensional Range) – The term for the range of ids a kernel is mapped onto.

Workgroup – A subset of the *NDRange* small enough to execute its corresponding *work item*s on a single compute unit.

Work Item – A particular instantiation of a kernel. When executing, every kernel is mapped onto one set of ids in the NDRange and the kernel+ids combination is referred to as a work item (it fully describes what work needs to be done).

Buffer – A host-code object that references device memory. To perform an operation on data using an OpenCL device, the data must be moved from host memory (usually RAM) to the device's own memory.

[2]Fencing is also used to ensure synchronisation of kernels on devices with a single program counter and instruction register for several parallel tasks, which is not the case with the Mali T-628 GPU [17].



```
for(int y = 0; y < 3; y++){
  for(int x = 0; x < 5; x++){
    int index = 5*y + x
    C[index] = A[index] + B[index];
  }
}
```

```
  int x = get_global_id(0);
  int y = get_global_id(1);

  int index = 5*y + x
  C[index] = A[index] + B[index];
```

```
NDRange(5, 3)=
(0, 0) (1, 0) (2, 0) (3, 0) (4, 0)
(0, 1) (1, 1) (2, 1) (3, 1) (4, 1)
(0, 2) (1, 2) (2, 2) (3, 2) (4, 2)
```

Figure 2.1: Visual representation of the OpenCL paradigm: a piece of sequential code is kernelised and mapped onto an NDRange

2.2 Neural Networks

Neural networks are a machine learning model inspired by the operation of biological neural networks. The model comprises of mathematical operations chained together to transform input vectors into output vectors. Most common operations include affine transformations, pooling operations, convolution, normalisation and nonlinearities, the internal parameters of which are learned via gradient descent.

In this work we focus on effectively executing already trained models, which significantly trims down the terminology used. Some key definitions are as follows:

Forward Propagation / Inference – applying the entire set of transformations in a neural network to some input.

Input / Feature Vector – a vector that can be propagated through a layer or network.

Feature / Dimension – a single element of a feature vector.

Channel / Feature Map – a subset of dimensions that conveys information about a single feature's value, conditional on some positional information. (Typical examples can be the red, green and blue channels of an image).

Weights – refers to the internal parameters learned by transformation layers.

Nonlinearities – nonlinear transformation applied to propagated values element-wise. Common transformations include the sigmoid function and ReLU (rectified linear unit).

Batch – a set of one or more inputs that are propagated over a layer or network together. Inputs are usually batched to improve computational performance and the batching has no effect on the computed outputs.

Filter – the weights used by convolutional layers to evaluate a certain slice across the channels of propagated inputs. Is also referred to as a kernel, though we are not going to use this term due to its ambiguity with an OpenCL kernel. Whenever the word kernel is used in this work, it refers to the OpenCL kernel, not the convolutional filter.

Stride – the offsets between individual slices to which filters are applied during convolution.

Chapter 3

Related Work

3.1 TensorFlow and Other Frameworks

TensorFlow [4] is one of the most popular neural network and machine learning frameworks used today. TensorFlow provides an interface for expressing machine learning algorithms in terms of Tensor operations and maps these operations onto optimised GPU kernels.

TensorFlow often makes use of existing highly optimised kernels which are parts of other libraries, including cuBLAS [18], cuda-convnet [19] and cuDNN [20]. These libraries are optimised for the NVIDIA CUDA architecture and optimisations that improve performance on NVIDIA GPUs often have a negative impact on the Mali architecture. The Mali developer guide published by ARM explicitly recommends removing some of the optimisations tailored to other architectures [21].

3.2 CLBlast

The most recent general-purpose machine-learning-enabled library optimised for the Mali T-628 GPU was released in May 2017 under the name CLBlast [22]. Inspired by the clBLAS library [23], CLBlast is an automatically tuned BLAS (Basic Linear Algebra Subroutines) library, which implements all BLAS operations as well as batched versions of some of them.

Apart from BLAS operations, CLBlast also provides us with an im2col subroutine, which can be used to perform convolution. With this functionality, CLBlast is a suitable candidate for implementing a deep neural network execution system. The library was even presented as a “Deep learning enabled BLAS Library” by its creator Cedric Nutegren [24].

There are several reasons why CLBlast is very relevant to our research. Firstly, it tunes itself to the device it runs on and the Mali T-628 is one of the 40 devices used to test the library during its creation [22]. Secondly, it is capable of running matrix multiplication on the Mali T-628 GPU at 8 GFLOPS [22], which is extremely fast compared to other research [25, 17] if it is accomplished without rearranging matrices in memory. These facts, in combination with having im2col support and batched versions of GEMM make it as close to state-of-the-art as we can get with our device.

A disadvantage of using CLBlast is its lack of pooling layer support and activation function support, which are important and popular neural network features. We touch on this more in Section 4.3 and attempt to work around other limitations in Section 6.

3.3 Previous Work - Morton GEMM

This project is the continuation of our previous work published by Rovder in April 2017 under the name "Evaluating Neural Networks on Low-Powered GPUs" [17].

The main focus of the previous work was to investigate the Mali T-628 architecture and how it can be correctly utilised to optimise basic neural network operations. We implemented fully-connected layers, bias addition layers and two activation layers.

The primary contribution of our previous work was the Hybrid Morton Order based general matrix multiplication kernel (*Morton GEMM* kernel for short). This kernel was inspired by the memory blocking GEMM kernel presented in ARM reference literature specifically for the Mali T-628 GPU [25], which made use of the OpenCL `dot` function and computed a 2-by-2 patch of the resulting matrix in a single work item (visualised in Figure 3.2).

Rovder improved the performance of the ARM kernel by 12% using optimal memory layouts, which enabled better prefetching on the device.

The standard Morton Order layout (also known as Z-Order layout) is optimised for single threaded access along both rows and columns of a matrix [1]. Rovder exploited control over OpenCL workgroup sizes and the order in which workgroups are executed to predict exactly which rows and columns of which matrix will be traversed at what time. The Hybrid Morton Order layout was then designed to optimise prefetching, ultimately creating the Morton GEMM kernel. The layouts are visualised in Figure 3.1.

The optimal workgroup size for the Morton GEMM kernel is 4 rows by 16 columns of the NDRange. As such, a single workgroup computes an 8 row by 32 column patch of the resulting matrix (because each work item computes a 2-by-2 patch). This means the output matrix has to be padded such that its dimensions are multiples of these constants, leading to some memory redundancy. This redundancy is, however, relatively small and is a price worth paying for the substantial speed increase gained from using the Morton GEMM kernel.

The Morton GEMM kernel performs matrix multiplication at 13.5 GFLOPS at the *least* and exhibits no performance diminishing as matrix sizes increase.

[1] Using the regular Morton Order layout, row-wise and column-wise matrix accesses should perform roughly equally well.

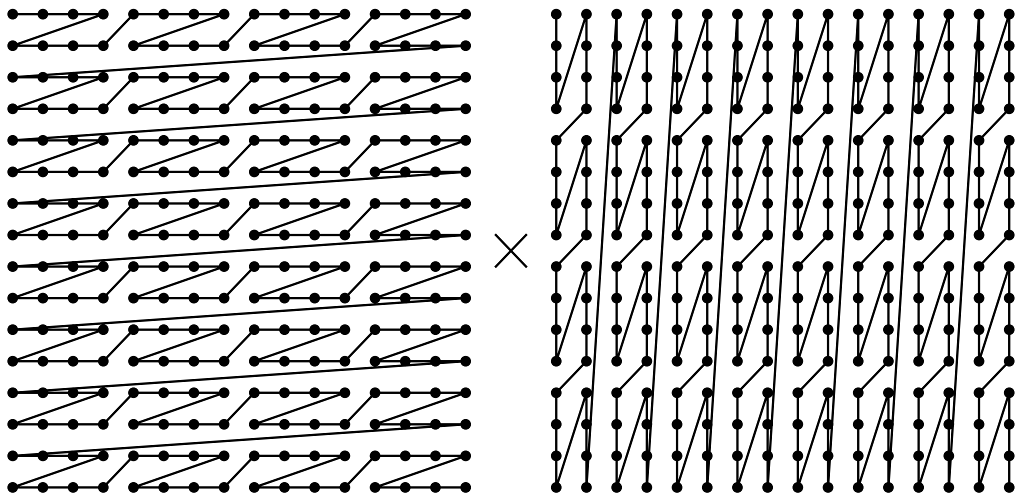


Figure 3.1: Memory layouts used by the Morton GEMM kernel for the left and right matrices of the operation respectively. The resulting matrix may be any layout, depending on the preference of any subsequent operation. Rovder referred to these layouts as R_2_4_R and C_4_2_C respectively [17].

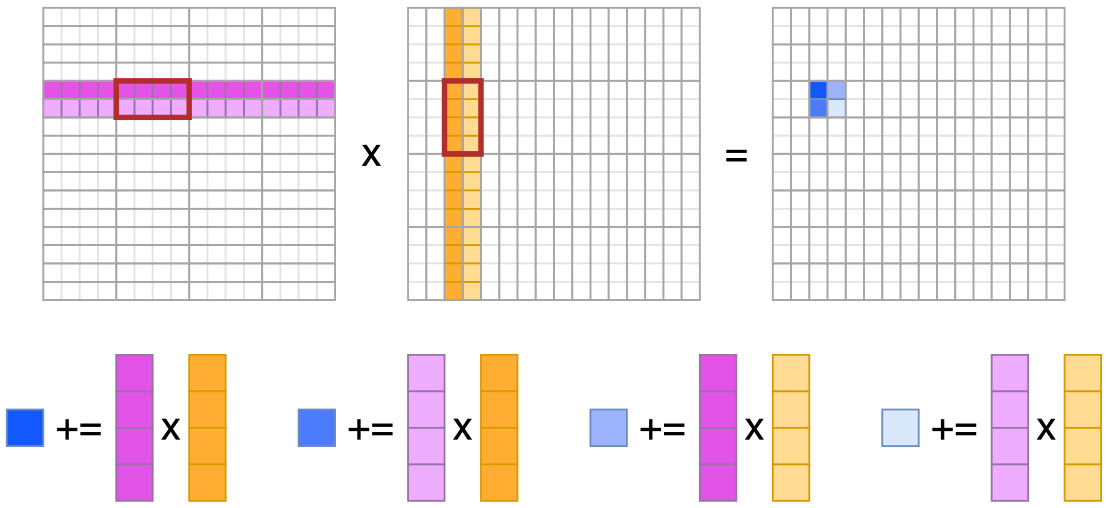


Figure 3.2: Visualisation of the work done within a single work item of the Morton GEMM kernel. Values are loaded from matrices in quadruplets into the `float4` OpenCL type in a sliding window fashion. Pairwise `float4` dot products are computed using the `dot` function and added to the cumulative result.

3.4 Neural Networks with CLBlast

Similar research to ours has recently been done by Loukadakis et al. in “Accelerating Deep Neural Networks on Low Power Heterogeneous Architectures” [11].

Loukadakis et al. compared VGG-16 inference time of several systems against one another on the Mali T-628 GPU. The most relevant of these systems were a baseline C version, a CLBlast accelerated version, and a hand-optimised OpenCL version.

Their research demonstrated how optimising workgroup sizes and using vector data types in convolution kernels can be used to outperform CLBlast convolution when forward propagating a single input image over the VGG-16 network.

The reported results indicate that CLBlast can achieve a six-fold speedup over a baseline serial C version (taking inference time down to 28 seconds from the 79 second baseline). The report also shows how a custom OpenCL implementation utilising SIMD instructions (**S**ingle **I**nstruction **M**ultiple **D**ata) and workgroup size optimisation reduced inference time to 8.5 seconds. These findings are in line what we observed in our previous work.

The primary issue with this research is its focus on convolutional layers only. Max-pooling, ReLU and dense layers were executed using custom written unoptimised OpenCL kernels, and im2col was performed on the CPU. This means that the only acceleration implemented using CLBlast was the general matrix multiplication used to compute im2col convolution. Furthermore, the paper’s reference to inference time applies only to convolutional layers, meaning it does not provide any metric for how long the full VGG-16 inference time was. These facts were found by consulting with paper co-author Jose Cano and investigating the code used to gather the results published in the paper.

Chapter 4

CLBlast on the ORDOID Board

Ideally, we would be using a neural network library as a baseline for this project. However, since there is no neural network library tuned specifically to the Mali T-628 GPU, we resort to implementing a simple neural network evaluation system using the best available neural-network-enabled BLAS library optimised for the device.

The CLBlast library [22] was inspired by AMD's clBLAS [23]. It provides us with the same matrix and vector operations as clBLAS, along with an implementation of the im2col operation (not a BLAS subroutine). The presence of im2col enables the library to execute convolutional layers and subsampling layers on the GPU, making it suitable for deep learning [1].

Furthermore, CLBlast makes use of a third party library to tune itself to a particular device. The Mali T-628 GPU was one of the 40 target devices used to test the library during its writing. Given the library's tuning to the target device of our research and its support for deep learning, it is currently the closest research to ours.

This section outlines our process of compiling, tuning and benchmarking the CLBlast library using the Mali T-628 GPU on the ODROID UX3 board.

4.1 Compilation and Tuning

CLBlast comes pre-tuned to the Mali T-628 GPU out-of-the-box, making the compilation extremely straight-forward, using the unix **make** tool. The issue with this approach is that we cannot be certain what device the GPU was on during the tuning process. Our Mali GPU is on the ODROID board, which may have different hardware from the device used during pre-tuning.

To ensure a completely fair benchmark of the library, we decided to run the tuning process on our device before any benchmarking took place.

[1]CLBlast has been advertised as "A Tuned BLAS Library for Faster Deep Learning" by its author Cedric Nutegren in May 2017 [24]

4.1.1 Tuning CLBlast

The tuning process is done between two runs of compilation.

First the library is compiled with the `-DTUNERS=ON` compilation flag, which includes the tuning logic into the build.

Next, the tuning process is launched. The tuning process is defined in the library CMakeLists, meaning it could be launched simply using the **make alltuners** command. This process took at least 8 hours, potentially more; the precise time is not known since it was left running over night.

The tuning process stores the measured results into JSON files and generates a *database.py* file. This *database.py* file is run after the tuning process completes and it evaluates the results, storing optimal settings into `C++` header files to be used in the second compilation.

After the *database.py* script stores optimal settings into the header files, the library is compiled a second time, yielding a fully optimised binary.

Tuning Issues

There were a number of issues that arose during the tuning process. This section outlines what they were and how they were bypassed.

Firstly, some configurations of the routines did not yield any results on our system. The reason for this is unknown and the *database.py* script did not handle this (crashed). The offending routines were the three `xgemv_fast` kernels with 16-bit precision. Since these routines were not used in our research, the problem was bypassed by replicating the results of the 32-bit precision routine equivalents and copying them into the 16-bit precision JSON files.

Secondly, the *database.py* script is meant to find one unique optimal setting for each routine, however, it failed to guarantee the uniqueness on our system. We managed to trace this problem down to the script's failure to disambiguate between clock frequencies of different devices. The problem was bypassed by hard-coding the clock frequency of our device into the innermost loop of the *database.py* script to ensure the correct setting was found.

The reason for two clock frequencies appearing in the results is not known for certain. Investigation of the script indicates that the system attempts to default to settings that came with the library in cases where the tuning results were inconclusive. If the pre-tuning process was done on a Mali T-628 GPU running at 600Mhz while our device is running on 533Mhz, it could explain the discrepancy between clock speeds. This is all, however, speculative.

4.2 Benchmarking CLBlast Subroutines

Despite being advertised as a deep-learning-enabled BLAS library, CLBlast does not actually provide any neural network layer subroutines out-of-the-box. Instead, CLBlast provides a set of lower-level subroutines that can be chained together to emulate the behavior of individual neural network layers.

In this section, we select appropriate lower-level subroutines from CLBlast for implementing the layers required to fulfill the goals outlined in Section 1.2. We do not construct the layers themselves yet (this is done later in Section 5.1), we first benchmark the subroutines individually, looking for behavioral anomalies and bottlenecks.

Table 4.1 provides a summary of which CLBlast subroutines are used by which neural network layer. The five used subroutines are `xCOPY`, `xGEMM`, `xGEMV` [2], `xXGEMMBATCHED` and `xIM2COL`.

Layers	CLBlast subroutines
Fully-connected layers	`xCOPY` & `xGEMM`
ReLU Layers	Unsupported
Sigmoid Layer	Unsupported
Convolutional layers	`xCOPY` & `xGEMMBATCHED` & `xIM2COL`
Pooling layers	Unuspported
Subsampling layers	`xCOPY` & `xGEMMBATCHED` & `xIM2COL`

Table 4.1: CLBlast subroutine usage in neural network layers

4.2.1 xGEMM

General matrix multiplication is arguably the most important BLAS operation needed to implement neural networks. In our benchmarks, we will use the `xGEMM` subroutine to execute fully-connected layers.

To investigate the performance of CLBlast's `xGEMM` subroutine, we ran 16 experiments with progressivelly larger square matrices. The performances can be seen in Table 4.2 and Figure 4.1.

The first observation we can make is that none of our experiments reached the performance of 8 GFLOPs presented in the CLBlast paper on the Mali T-628 GPU [22]. The paper does not specify which setting this performance was observed at, rather obstructing any attempts to reproduce the results. Our results indicate that the overall performance in GFLOPs increases with the matrix sizes, so this 8 GFLOPs performance may simply only apply to extremely large matrices.

[2] We can use `xGEMV` (matrix-vector multiplication) if we are forward propagating a single input vector over a fully-connected layer at a time.

Size	Run Time [ms]	GFLOPS	Size	Run Time [ms]	GFLOPS
96	327.94	0.01	1024	654.98	3.28
128	328.14	0.01	1152	789.73	3.87
256	371.12	0.09	1280	966.69	4.34
384	486.82	0.23	1408	1173.08	4.76
512	711.93	0.38	1536	1434.94	5.05
640	1084.99	0.48	1664	1730.32	5.33
768	1639.94	0.55	1792	2096.49	5.49
896	548.39	2.62	1920	2499.03	5.66

Table 4.2: Performance of CLBlast `xGEMM` subroutine on various-sized square matrices

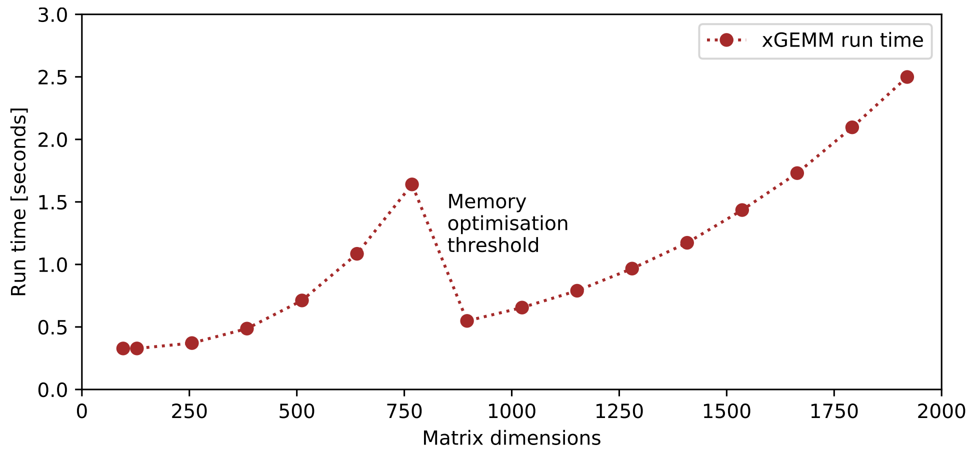


Figure 4.1: Performance of CLBlast `xGEMM` subroutine on various-sized square matrices

The second observation we can make is that there is a drastic increase in performance after the matrices exceed the size of 895-by-895 [3]. We investigated the CLBlast source code to find the optimisation responsible for this speedup and discovered that the `xGEMM` subroutine decides whether to perform "direct" or "indirect" multiplication based on the size of the matrices. When multiplying an $m \times k$ matrix with a $k \times m$ matrix, if the product mnk exceeds a certain threshold, the "indirect" approach is selected. This threshold is a value called `XGEMM_MIN_INDIRECT_SIZE` and it is automatically found by CLTune during the tuning stage.

When taking the ”indirect” approach, CLTune will re-shape the matrix in memory, changing layouts to perform multiplication faster. This is a crucial observation, since in our previous work we demonstrated how general matrix multiplication can be performed at up to 14 GFLOPs on the Mali T-628 if the matrices are appropriately arranged in memory. As such, our previous work already outperforms CLBlast in this aspect.

[3]This value is not explicitly seen in the graphs, it was found by repeating experiments, using binary search to find the switching point.

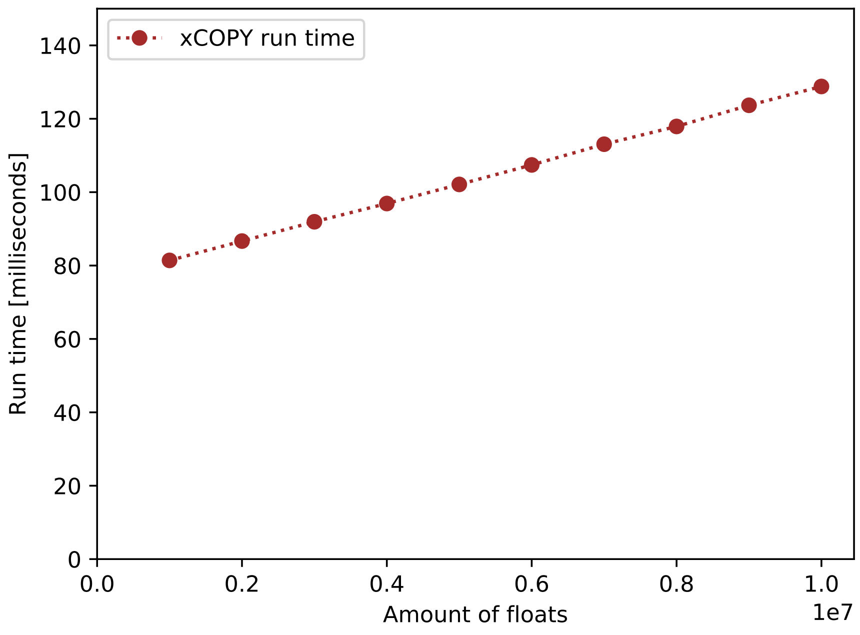


Figure 4.2: CLBlast `xCOPY` performance for various-sized buffers

4.2.2 xCOPY

CLBlast does not support resetting the content of a matrix. That is, there is no subroutine that sets all the values in a specific buffer to zeros (or any other specified value).

Most CLBlast subroutines add the result of whatever they compute to the values in the target buffer as opposed to overwriting them. The general behavior is as follows:

$$Y \leftarrow \alpha\, operation(inputdata) + \beta\, Y$$

This behavior is a particular issue in neural networks, since forward propagation without zeroing out would simply accumulate intermediate values at every step, compromising all results other than the ones computed by the very first propagation.

We solve this issue by re-initialising the matrix buffers not to zeros, but to the bias values using `xCOPY`. Thus setting β to 1 makes the subroutines add results to the bias-initialised buffer.

Performance of `xCOPY` only depends on the size of the copied buffer. Figure 4.2 shows a clear linear relation between the run time of `xCOPY` and the size of the buffer. In general, the run time of `xCOPY` can be approximated by the following equation:

$$T_{xCOPY}(x) = 0.005\, x + 76403$$

This relation shows that the run time of `xCOPY` is dominated by the kernel initialisation time (the constant term) and the gradient at which the time grows is very small.

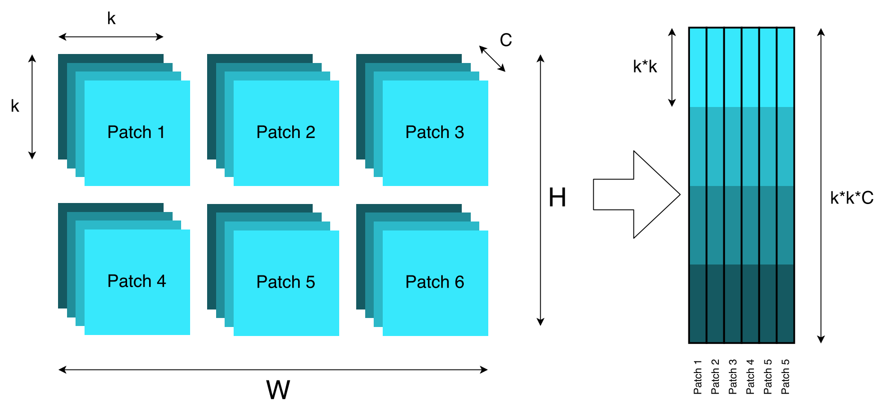


Figure 4.3: Visualisation of the im2col operation

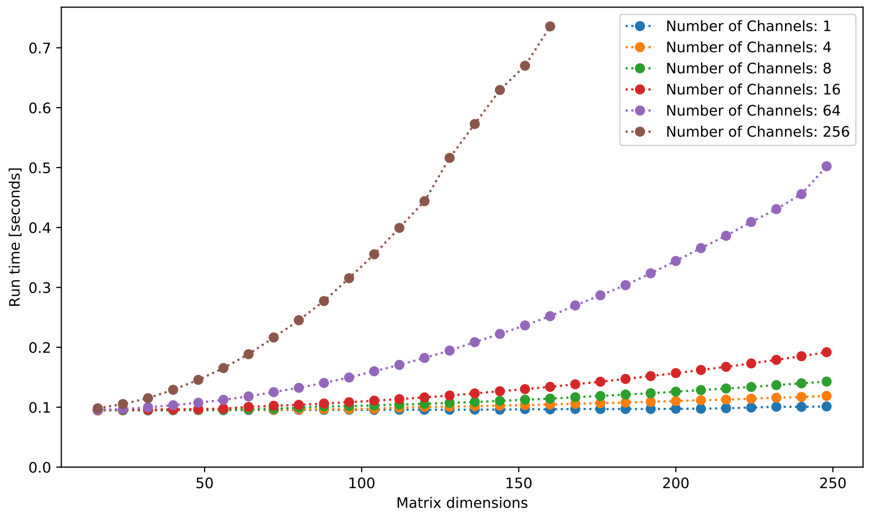


Figure 4.4: Performance of the im2col subroutine on varying size square matrices and numbers of channels

4.2.3 xIM2COL

The im2col operation is often used in conjunction with GEMM to emulate convolution [26]. The approach is so popular it is even used in some machine learning frameworks like Caffee [27] and it is the main way in which CLBlast supports deep learning.

CLBlast provides us with an `xIM2COL` subroutine, which performs a 3-dimensional im2col operation, meaning it assumes the input matrix represents a single 2-dimensional image with C channels. Figure 4.3 shows a visualisation of the im2col operation with a stride of k. This stride can, however, be smaller than k, in which case there will be an overlap in the patches.

We shall benchmark the performance of this subroutine using square patches of size $k = 5$ on square inputs of increasing widths and heights. We will also test it out against inputs with various amounts of channels to check whether it scales well.

Figure 4.4 shows how when the number of channels is small, the `xIM2COL` subroutine performs and scales very well. For high number of channels like 256, however, the time increase is quite drastic with increasing input sizes. Most convolutional networks keep the number of parameters constant across the layers, increasing the number of channels and reducing the dimensions of the inputs. This would mean that by the time an input reaches 256 channels, it will be quite small in its width and height and likely small enough to perform fast.

An important observation here is that for inputs of 256 channels, larger than 160-by-160, the subroutine is killed and does not compute the result. This is expected, since the resulting matrix of the im2col operation on such an input would exceed the device's MAX_MEM_ALLOC_SIZE restriction.

4.2.4 xGEMV

In the particular case of forward propagating a single input vector over a fully-connected layer at a time, the operation can be reduced to matrix-vector multiplication. This section investigates the performance of `xGEMV` to determine whether there is any advantage of using `xGEMV` over `xGEMM` in such cases.

To do this we will use both `xGEMV` and `xGEMM` to perform the same linear transformation of a single vector using a square transformation matrix. We will gradually increase the dimensions of the transformation matrix and observe the differences in performance.

We predict that `xGEMV` will perform better than `xGEMM`, since matrix-vector multiplication should require an NDRange of only a single dimension to perform the operation. This should result in a less complex kernel and hence faster performance.

The results of our experiment in Figure 4.5 and Table 4.3 clearly indicate that this is in fact the case. The performance of `xGEMV` is greatly superior to that of `xGEMM`, especially for large matrix sizes. This fits the needs of VGG-16 perfectly and will be made use of in Section 5.3

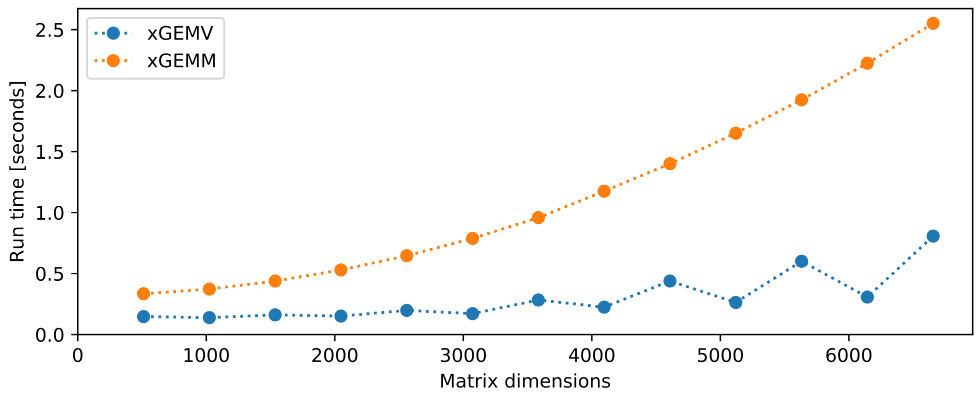


Figure 4.5: Comparison between `xGEMM` and `xGEMV` performance on various-sized square matrices with different batch sizes

Dimensions	xGEMV time [s]	xGEMM time [s]
512	0.15	0.33
1024	0.14	0.37
1536	0.16	0.44
2048	0.15	0.53
2560	0.20	0.65
3072	0.17	0.79
3584	0.28	0.96
4096	0.22	1.18
4608	0.44	1.40
5120	0.26	1.65
5632	0.60	1.92
6144	0.31	2.22
6656	0.81	2.55

Table 4.3: Comparison between `xGEMM` and `xGEMV` performance on various-sized square matrices with different batch sizes

4.2.5 xGEMMBATCHED

The `xGEMMBATCHED` subroutine allows us to multiply several pairs of matrices in a single operation. This means we can emulate the effect of running `xGEMM` multiple times without the cumulative overhead of actually instantiating the individual `xGEMM` operations.

There are some restrictions to what this subroutine can do. First of all, the multiplied matrices need to be placed in the same buffer (though their start and end points may overlap). Second of all, the matrices must all be the same dimensions.

In order to benchmark the performance of this subroutine, we ran several experiments, multiplying progressively larger square matrices in batches of one, two, three and four. The resulting average times of three runs can be seen in Table 4.4 and Figure 4.6.

There are two key observations we can make here. Firstly, the run time of the subroutine appears to scale linearly with the size of the batch. This is a very promising result, since running convolution will require many batches (more on this in Section 5.1.2).

The second observation is less positive: the memory optimisation we saw with `xGEMM` in Section 4.2.1 was not applied to `xGEMMBATCHED`. This means that the performance of the `xGEMMBATCHED` subroutine hovers below 0.5 GFLOPS in all the run benchmarks, which is much slower than even numpy (a Python library) can multiply the matrices on the CPU of the ODROID board.

Take for instance the 1152-by-1152 benchmark with 4 batches. The `xGEMMBATCHED` subroutine took 32.48 seconds to compute the products, implying a speed of roughly 0.09 GFLOPS. Meanwhile, numpy can compute the same results in 3.44 seconds at 0.88 GFLOPS, which is almost 10 times faster than the CLBlast subroutine.

Further exploration of `xGEMMBATCHED`'s viability for convolution is in Section 5.1.2.

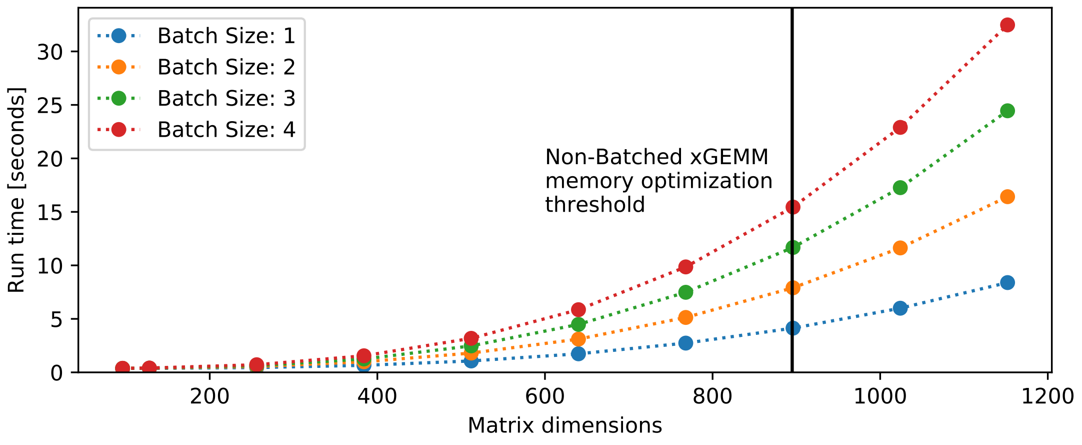


Figure 4.6: Performance of CLBlast `xGEMMBATCHED` subroutine on various-sized square matrices with different batch sizes

Size	Batches: 1	Batches: 2	Batches: 3	Batches: 4
96	0.37s	0.38s	0.37s	0.38s
128	0.37s	0.39s	0.39s	0.40s
256	0.45s	0.55s	0.62s	0.71s
384	0.66s	0.99s	1.25s	1.55s
512	1.07s	1.80s	2.47s	3.18s
640	1.73s	3.11s	4.48s	5.86s
768	2.73s	5.13s	7.48s	9.86s
896	4.13s	7.91s	11.68s	15.45s
1024	6.00s	11.63s	17.26s	22.90s
1152	8.39s	16.42s	24.45s	32.48s

Table 4.4: Performance of CLBlast `xGEMMBATCHED` subroutine on various-sized square matrices with different batch sizes

4.3 Limitations of CLBlast

As a whole, CLBlast is a well performing BLAS library. Even on the Mali T-628 GPU, the subroutines perform reasonably well and are on-par with the basic preliminary kernel benchmarks we ran in our previous work. The library is, nonetheless, far from optimal and has numerous flaws.

First of all, the general matrix multiplication subroutine does not perform as well as the baseline GEMM kernels published by ARM for this particular GPU [25] and is in fact 30% slower than it's ARM counterpart. The ARM kernels require reshaping matrices in memory and aligning them to multiples of 4, which could have explained the performance slack of CLBlast had CLBlast not also reshaped matrices in memory. But it does, meaning there is no reason the CLBlast subroutines couldn't perform as well as the kernels published by ARM. Considering that the entire deep learning aspect of CLBlast hinges on the performance of general matrix multiplication, this is not ideal.

Second of all, the optimisations that CLBlast does put in place do not extend to the batched version of `xGEMM`. This is a major flaw because `xGEMBATCHED` is required for forward propagating a batch of inputs over a convolutional layer, and, without these optimisations, convolution utilises less than 1% of the GPU computational power.

Furthermore, one major functional limitation of the library is that it does not support setting the values in a buffer to another value (without performing any operation on them) and only allows adding to values that are already in the buffers. Resetting buffer values is, arguably, not a BLAS operation, however, CLBlast already goes beyond BLAS operations by implementing im2col. CLBlast is advertised as a "BLAS Library for Faster Deep Learning" [24] and not being able to recycle buffers without explicitly copying zeros to them is an unacceptable shortcoming.

The library also lacks any support for activation nonlinearities and pooling operations.

In conclusion, while CLBlast has selective support for some deep neural network operations, it does not support deep learning very well.

4.4 Summary

In this chapter we took the latest research into accelerating mathematical operations on the Mali T-628 GPU, hand-picked the subroutines relevant to our goals and tested them out on our device. We benchmarked the subroutines, identifying strengths, shortcomings and limitations, and we now have enough information about the system's capabilities to construct a neural network execution system with it.

Chapter 5

Executing LeNet and VGG-16 with CLBlast

In this chapter, we use CLBlast to execute LeNet and VGG-16 on the Mali T-628 GPU. Our goal is to observe the execution time and look for any shortcomings or limitations that may help us identify ways to outperform it.

5.1 Layer Implementation

While CLBlast does support deep learning, it does so by providing low-level subroutines which we can use as building blocks for the layers. It does not provide implementations for the layers themselves, so this section is dedicated to implementing those layers we need to execute LeNet and VGG-16.

The combinations of subroutines used to emulate each individual layer are given in Table 4.1 of the previous chapter.

5.1.1 Fully-Connected Layers

Fully-connected layers are essentially an affine transformation of the input values. An input matrix X with N rows (one for each input vector) and D_x columns (one for each dimension of the input vectors) is multiplied by a weight matrix W and has a bias B added to it to compute the resulting matrix Y:

$$Y = X \times W + B$$

The bias B is a matrix of identical rows, since there is one bias for each column of the output matrix. Libraries like numpy allow addition of a single-row matrix to a multi-row matrix, simply adding the same row to all rows of the multi-row matrix. CLBlast supports no such operation, nor does it support resetting matrices, meaning we need to deal with biases in a separate step. Thus the fully-connected layer is a two-step operation in CLBlast.

First, we reset the values currently in the buffer of the output matrix Y using `xCOPY`. We will reset them to the bias values:

$$Y \leftarrow B$$

Then, we may use `xGEMM` to compute the product of X and W, and add the result to the values currently in Y:

$$Y \leftarrow X \times W + Y$$

Since Y was initialised to the bias values, this sequence of steps achieves an affine transformation.

In the particular case of VGG-16, we will only be forward propagating a single input vector at a time. This means we can use the `xGEMV` subroutine instead of `xGEMM` and make use of the performance gain as discussed in Section 4.2.4.

5.1.2 Convolutional Layers

Convolutional layers are by far the trickiest neural network layer to implement using CLBlast subroutines. While the combination of `xIM2COL` and `xGEMM` outlined by Chellapilla et al. [26] is sufficient for forward propagating a single input over VGG-16, it is not sufficient for forward propagating a batch of 100 inputs over LeNet.

The techniques outlined by Chellapilla et al. [26] work for batched inputs by simply repeating the im2col operation multiple times. Repeating operations on a GPU comes with kernel initialisation overheads and is hence undesirable.

Instead, in this chapter we present a method that makes use of a single run of `xIM2COL` and a single run of `xGEMMBATCHED` to extend the single-input technique from Chellapilla et al. to cover batched inputs.

Figure 5.1 shows a visualisation of the basic Chellapilla et al. convolution technique. Each of the input images (green, blue and purple) is transformed into its column-based counterpart using im2col (one by one) and then multiplied with the kernel matrix.

Under ideal circumstances we would either have a batched version of the `xIM2COL` subroutine, or the `xIM2COL` subroutine would support strides across another dimension. Neither of these are the case, so we emulate them by pretending all the images are a single high-dimensional image.

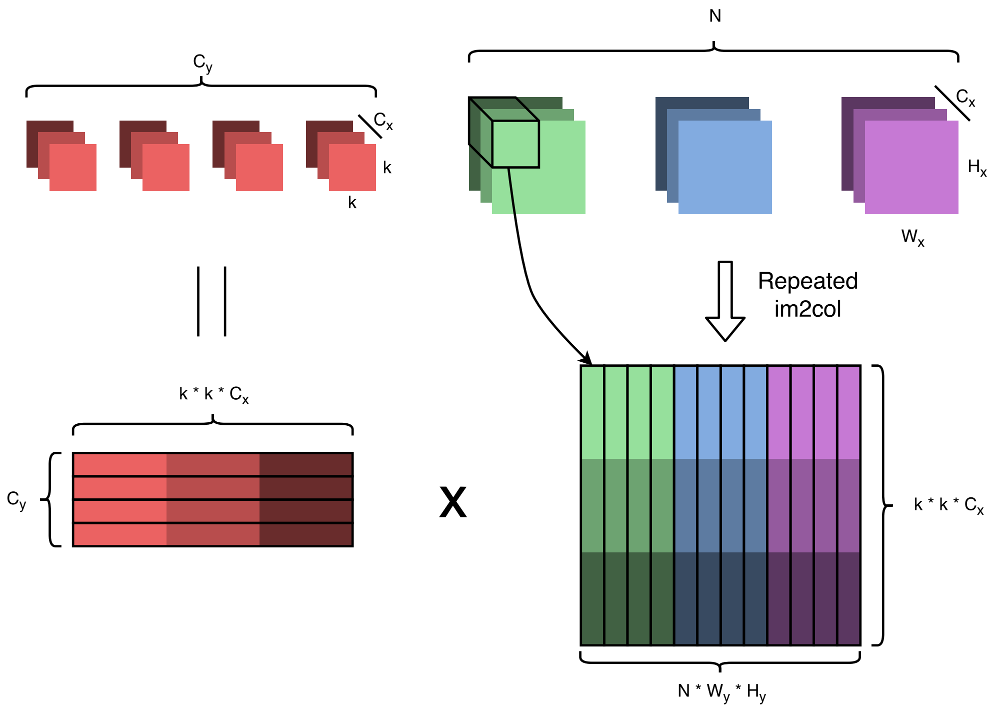


Figure 5.1: Chellapilla et al. unrolled convolution visualised: kernels (red) remain untransformed, input images (green, blue, purple) are transformed into columns

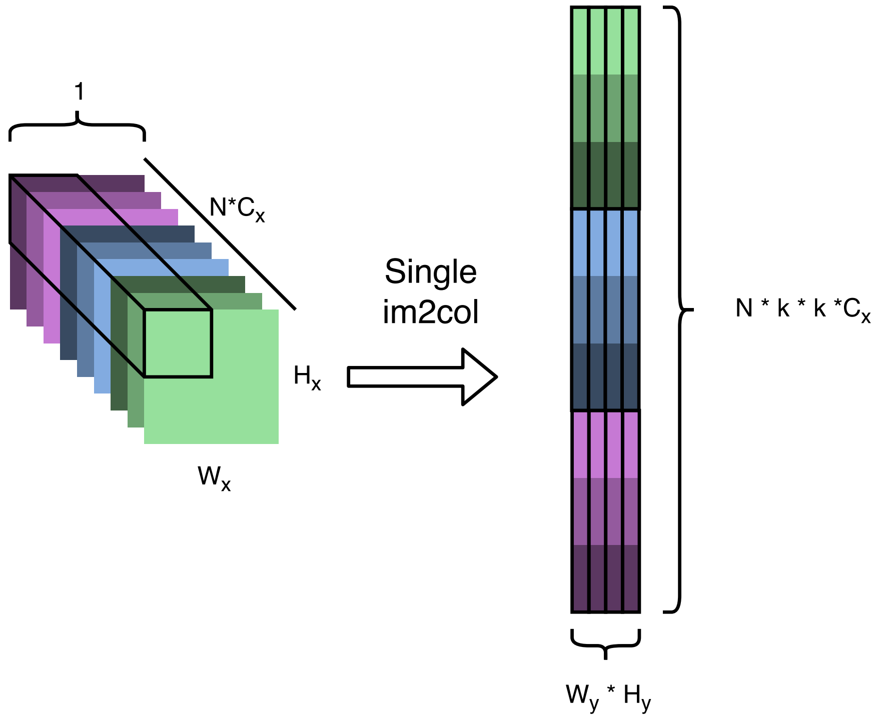


Figure 5.2: Alternative approach to using im2col: all input images are transformed in a single pass

Supposing we are trying to forward-propagate N input images of C_X channels, we could pretend that we are forward-propagating a single input image with NC_X channels. This would achieve a result as visualised in Figure 5.2, which is different from that of Chellapilla et al. in Figure 5.1.

While we have avoided the issue of having to run `xIM2COL` multiple times, we now have the issue of having to run `xGEMM` multiple times to compute convolution (once for each chunk of rows in the matrix). This is, however, something CLBlast supports via the `xGEMMBATCHED` subroutine.

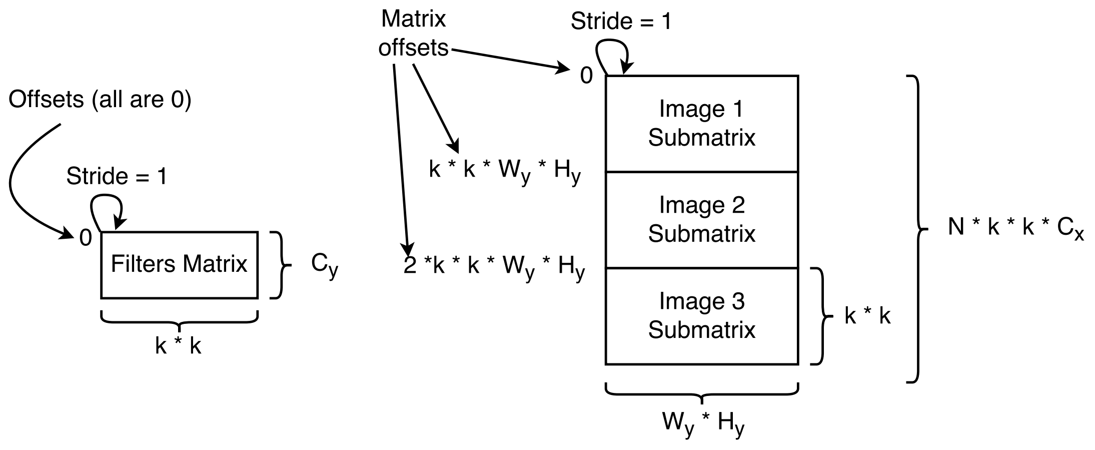


Figure 5.3: Visualisation of `xGEMMBATCHED` used to perform convolution on im2col output

Using `xGEMMBATCHED` with offsets and strides as shown in Figure 5.3, we can compute the same matrix product as is shown in Figure 5.1, individually for each image. The resulting matrix is arranged differently, but this is arguably better than the result of Chellapilla et al., since our batched approach does not require additional rearrangement after the operation completes. The Chellapilla et al. approach groups the image content by placing all the channels in separate rows, effectively splitting the image up in memory. Our approach does not do this and groups the images by image, meaning we can forward propagate it to the next layer without further rearrangement.

One major issue with the im2col approach is the massive memory redundancy it introduces. Performing convolution with small strides (which is very common) leads to duplicating values in the column matrix. With filters of size 5-by-5 (as will be used by LeNet) this can introduce a twenty-five-fold memory blowup, making this approach very expensive, especially when running on a low-powered GPU with smaller memory, like the Mali T-628.

5.1.3 Subsampling Layers

It is important to stress the difference between subsampling layers and pooling layers. When we talk about subsampling layers in this work, we mean subsampling layers as described by LeCun et al. in “Gradient-Based Learning Applied to Document Recognition” [9], where LeNet and convolution was first published as a concept.

By this definition, subsampling layers were almost the same as convolutional layers but operated on individual channels of each image as opposed to slices across all channels of an image. A subsampling layer is connected to a patch of k-by-k neighborhood in the corresponding feature map. The k^2 values are averaged together, multiplied by a scaling factor and have a bias added to them[1]. As such, each subsampling layer has a number of internal parameters equal to twice the number of incoming channels.

We can implement these networks by abstracting the problem to convolution. We create a single filter for every channel of the inputs. All these filters are initialised to zeros. We then set the values in each filter that correspond to the desired input channel to the scaling factor, which is scaled down by a factor of k^2 to account for the averaging operation of the subsampling layer. Once this is done, we can simply use convolution to execute the subsampling layer.

This approach is slightly redundant in computation, as we are convolving instead of executing what is very close to a pooling operation. The reason is that CLBlast does not support subsampling (or pooling for that matter) and we are forced to improvise in order to avoid the massive kernel instantiation overhead that would arise from performing subsampling using alternative subroutines in a loop.

5.1.4 Unsupported Layers

Three of the seven layers described in Section 1.2 cannot be implemented using CLBlast, namely ReLU layers, sigmoid layers and pooling layers. Activation layers are easily computable and usually cost negligible time, as demonstrated by our previous research [17], so their absence is not too consequential. The absence of pooling layers is, however, quite detrimental to CLBlast. Pooling layers are extremely common in deep learning and it is unclear how the library can be advertised as viable for deep learning when it lacks pooling functionality.

5.2 Executing LeNet with CLBlast

Now that we have a working CLBlast implementation of the required layers, we can chain them to execute LeNet and VGG-16 of the Mali T-628 GPU. We start by executing LeNet.

LeNet is composed of two convolutional layers (C1 and C3) interleaved with subsampling layers (S2 and S4) much like modern networks have max-pooling. These layers are then followed by two fully-connected layers (F5 and F6). A diagram of the architecture can be seen in Figure 5.4 and a more granular architecture overview can be found in Tables 5.1 and 5.2.

[1]Paraphrasing Gradient-Based Learning Applied to Document Recognition [9]

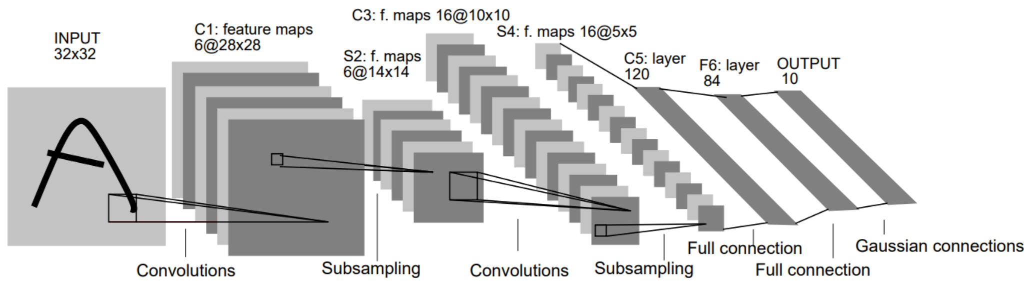


Figure 5.4: LeNet architecture from *Gradient-based learning applied to document recognition* by LeCun et al. [9]

Layer	Inputs			Filter Info				Outputs		
	W	H	C	W	H	Sh	Sw	W	H	C
C1	32	32	1	5	5	1	1	28	28	6
S2	28	28	6	2	2	2	2	14	14	6
C3	14	14	6	5	5	1	1	10	10	16
S4	10	10	16	2	2	2	2	5	5	16

Table 5.1: LeNet convolutional architecture breakdown. W=Width, H=Height, C=Channels, S=Stride

Layer	Input Dimensionality	Output Dimensionality
F5	400	120
F6	120	84

Table 5.2: LeNet fully-connected architecture breakdown

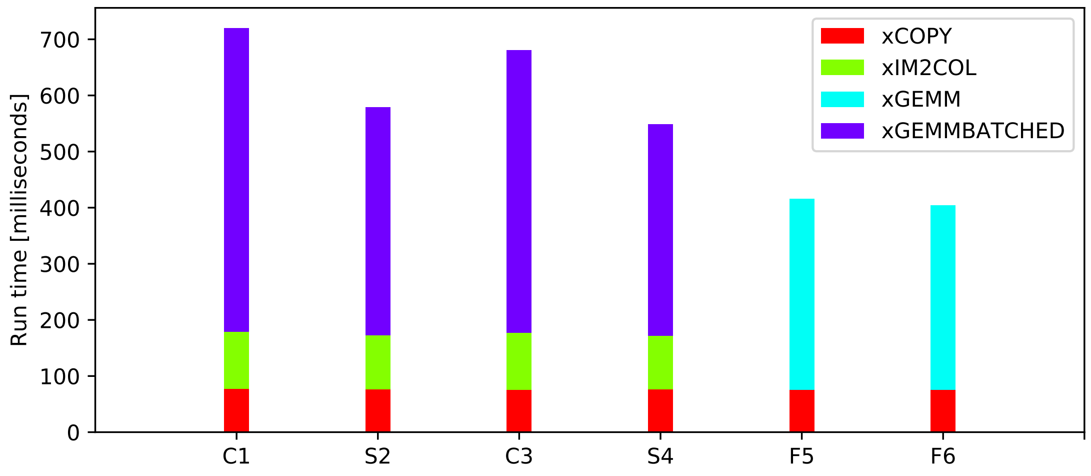


Figure 5.5: Run times of LeNet layers executed with CLBlast. Time visually split between individual CLBlast subroutines

Layer	Subroutine	Run Time	Total Run Time
C1	xCOPY	77.07	721.14
	xIM2COL	102.78	
	xGEMMBATCHED	541.29	
S2	xCOPY	76.48	579.78
	xIM2COL	97.20	
	xGEMMBATCHED	406.10	
C3	xCOPY	76.52	681.30
	xIM2COL	101.46	
	xGEMMBATCHED	503.32	
S4	xCOPY	76.32	550.48
	xIM2COL	96.68	
	xGEMMBATCHED	377.48	
F5	xCOPY	76.04	416.53
	xGEMM	340.49	
F6	xCOPY	75.96	405.32
	xGEMM	329.36	
Total			3354.55

Table 5.3: Breakdown of forward propagation time of 100 inputs through LeNet using CLBlast

We are not including the final gaussian layer seen in Figure 5.4 in our work, since this is outside the scope of the network itself [2].

Using the implementation of the layers as specified in earlier sections, we forward propagated a batch of 100 input images over LeNet on the GPU. The resulting run times of the layers can be seen in Figure 5.5, where the individual layer execution times are broken down by subroutine. The actual numeric values can be seen in Table 5.3.

It is clear from these results that the overheads associated with having layers composed of individual subroutines is quite high. This is most visible on the `xCOPY` and `xIM2COL` subroutines, the run times of which hover around 80 milliseconds, which is almost purely kernel initialisation time as per the formula in Section 4.2.2.

Given these observations, the highest priority improvement to CLBlast would be to support higher-level neural network operations in single subroutines to avoid this cumulative initialisation cost. This will also be one of our primary design goals for our custom system going forward in this work.

[2]The gaussian connections at the end served to probabilistically narrow down the outputs to the required classes. This is as intricacy of the particular model rather than a general neural network feature.

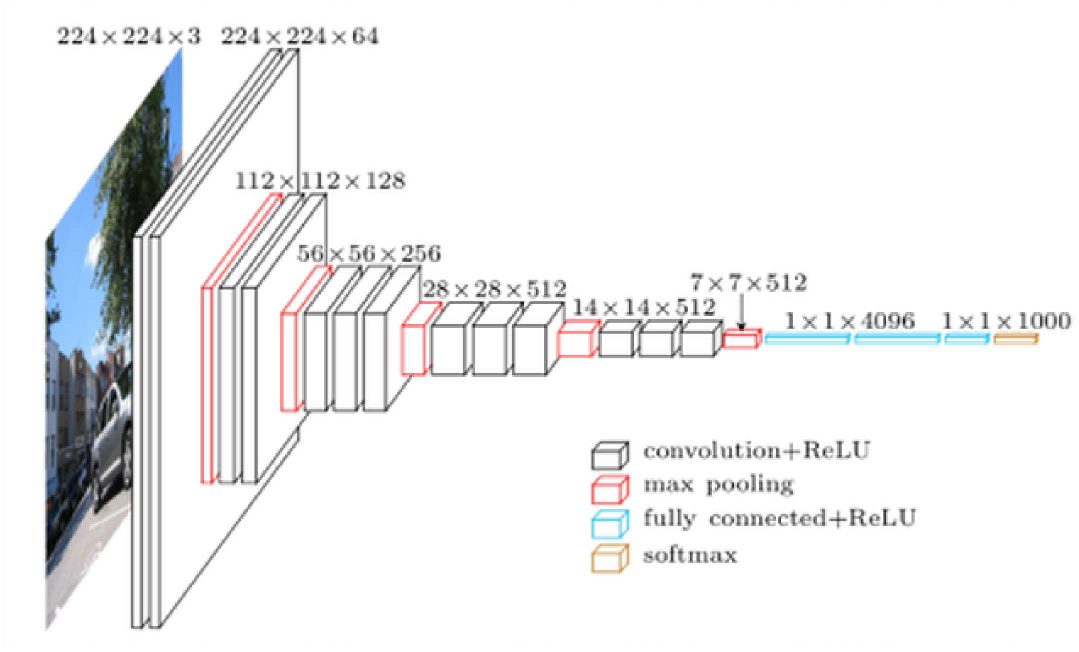


Figure 5.6: VGG-16 architecture from "Very Deep Convolutional Networks For Large-Scale Image Recognition" by Simonyan and Zisserman [10]

5.3 Executing VGG-16 with CLBlast

The second neural network we investigate in this work is VGG-16. We forward propagate a single input over this network at a time.

Propagating inputs one at a time means we can simplify our CLBlast implementation of convolutional layers by using the `xGEMM` subroutine instead of the `xGEMMBATCHED` subroutine, which, as we have seen in Sections 4.2.1 and 4.2.5, benefits from significant speed optimisations compared to its batched counterpart. We can also replace the affine layers' `xGEMM` subroutine with the `xGEMV` subroutine, which we have shown to be more effective than `xGEMM` at transforming a single input vector in Section 4.2.4.

These steps explicitly lock our implementation into only being able to propagate a single input at a time. This is not an issue, since it is one of our goals to test out CLBlast on forward propagating a single input over a large network.

There are several different network layouts published in "Very Deep Convolutional Networks For Large-Scale Image Recognition" [10], the most commonly used one being Model C (Table 1 of their paper). This is the model we focus on. Its architecture is visualised in Figure 5.6 and broken down in Tables 5.4 and 5.5.

Due to CLBlast limitations mentioned in Table 4.1, there are a number of operations required by VGG-16 for which there are no suitable subroutines available, namely max-pooling and ReLU. Loukadakis et al. [11] sidestepped this problem by implementing their own max-pooling and ReLU kernels, focusing mainly on accelerating convolution (for which they had CLBlast benchmarks). We are going to take a similar approach and time only those layers for which there is a CLBlast implementation.

Layer	Inputs			Filter Info				Outputs		
	W	H	C	W	H	Sh	Sw	W	H	C
conv1-1	224	224	3	3	3	1	1	224	224	64
conv1-2	224	224	64	3	3	1	1	224	224	64
mp1	224	224	64	2	2	2	2	112	112	64
conv2-1	112	112	64	3	3	1	1	112	112	128
conv2-2	112	112	128	3	3	1	1	112	112	128
mp2	112	112	128	2	2	2	2	56	56	128
conv3-1	56	56	128	3	3	1	1	56	56	256
conv3-2	56	56	256	3	3	1	1	56	56	256
conv3-3	56	56	256	3	3	1	1	56	56	256
mp3	56	56	256	2	2	2	2	28	28	256
conv4-1	28	28	256	3	3	1	1	28	28	512
conv4-2	28	28	512	3	3	1	1	28	28	512
conv4-3	28	28	512	3	3	1	1	28	28	512
mp4	28	28	512	2	2	2	2	14	14	512
conv5-1	14	14	512	3	3	1	1	14	14	512
conv5-2	14	14	512	3	3	1	1	14	14	512
conv5-3	14	14	512	3	3	1	1	14	14	512
mp5	14	14	512	2	2	2	2	7	7	512

Table 5.4: VGG-16 convolutional architecture breakdown. W=Width, H=Height, C=Channels, S=Stride

Layer	Input Dimensionality	Output Dimensionality
fc1	25088	4096
fc2	4096	4096
fc3	4096	1000

Table 5.5: VGG-16 fully-connected architecture breakdown

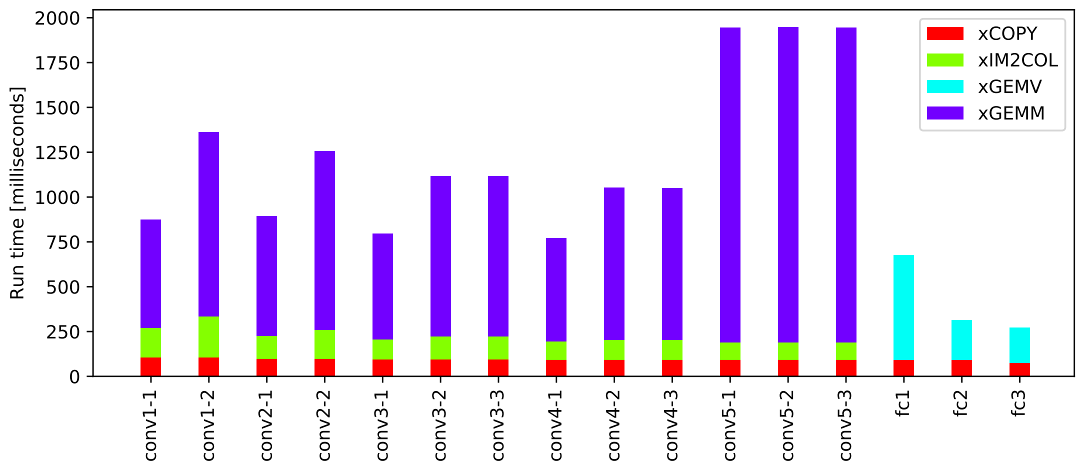


Figure 5.7: Run times of VGG-16 layers executed with CLBlast, time visually split between individual CLBlast subroutines

Figure 5.7 shows the execution time of each convolutional and fully-connected layer in VGG-16 using CLBlast, broken down by subroutine. The results are strikingly similar to those observed for LeNet in Figure 5.5. There are two reasons that conspire to cause this similarity: firstly, since we propagated 100 images over LeNet at a time and only propagated a single image over VGG-16 at a time, the amount of data forward propagated over the networks is the same order of magnitude for both VGG-16 and LeNet. This causes the similar run times of `xCOPY` and `xIM2COL`. Secondly, even though it is much more expensive to compute each of the propagated values (larger filters and feature counts for VGG-16), since we now use `xGEMM` instead of `xGEMMBATCHED` the memory optimisations seen in Section 4.2.1 are taking effect, bringing the convolutional layer times close to those seen in LeNet.

A noteworthy observation is how conv5 layers performed significantly slower than conv4 layers despite being exactly 4 times less computationally expensive. Upon investigation of this result we found that the matrix dimensions multiplied by `xGEMM` in conv5 layers fall just a bit short of the memory optimisation boundary mentioned in Section 4.2.1 and hence will perform very poorly.

While CLBlast does support convolution, it does not support zero padding of the input feature maps. This is a problem for VGG-16, since VGG-16 convolutional layers preserve feature map dimensions (for which we need zero padding functionality). In order to gain reasonable speed estimates for convolution, we assume that this functionality exists and we forward propagate larger feature maps over the network to simulate the effects. In other words, we measure times by performing convolution on feature maps that are as large as they would have been had CLBlast been able to pad them with zeros.

The full breakdown of run times of each subroutine are in Appendix A. The total inference time of all convolutional layers combined was measured at 16.14 seconds, which is 3 seconds more than measured by Loukadakis et al. [11]. This discrepancy can be attributed to the fact that Loukadakis et al. did not use the CLBlast `xIM2COL` subroutine and instead performed im2col using their own custom implementation directly on the CPU. This may indicate that CLBlast's GPU im2col implementation might be less efficient than a CPU implementation.

5.4 Summary

In this chapter we used CLBlast to execute LeNet and VGG-16 on the Mali T-628 GPU. We investigated the shortcomings of CLBlast as well as its performance bottlenecks to identify potentially viable methods for outperforming it in subsequent chapters.

The major issues we found with CLBlast are the massive memory blowup introduced by using the im2col approach for convolution, the kernel initialisation time overhead introduced by having layers composed of multiple subroutines, and the computational redundancy introduced by implementing subsampling layers as convolution with sparse filters.

Furthermore, CLBlast does not support any method for padding feature maps with zeros to perform dimensionality-preserving convolution. This is not a problem for LeNet but is a major issue for VGG-16, since without this functionality CLBLast is not able to perform convolution properly.

All these issues are going to be addressed in the subsequent chapters.

Chapter 6

Executing LeNet with Custom System

This chapter outlines the process of creating custom neural network operation kernels for the Mali T-628 GPU. In order to test our hypothesis from Section 1.2 we shall use the CLBlast LeNet benchmarks from Section 5.2 Figure 5.5 as our target metric and attempt to outperform it.

6.1 Baseline Implementation

The first step will be to create a very simple baseline system that can execute LeNet, which we can then build on top of. In the previous chapter we identified several issues with the CLBlast approach, so in the initial iteration of our system we are going to address them as follows:

Memory Blowup introduced by im2col: while im2col is a simple and effective approach to performing convolution, it is not practical on a small embedded GPU such as the Mali T-628. We will implement our baseline convolutional layer using an iterative approach.

Instantiation overhead introduced by chaining kernels: instead of chaining together several simpler kernels to compose our layers, we shall implement each layer as a single kernel.

Computational redundancy in subsampling layers: we will implement a separate subsampling layer kernel to avoid having to simulate subsampling using convolution.

Another key observation is that once a kernel is compiled for a particular layer [1] the kernel can be reused for every forward propagation. This means we only need to compile all required kernels once, build the model out of them, and simply forward propagate input batches without ever having to undergo the compilation time overhead again.

[1]Due to compiler flags, the kernels may be different for different layers even if the layers are the same type (eg. convolutional layers operating on differing amounts of features)

6.1.1 Fully-Connected Layers

Fully-connected layers in LeNet are an affine transformation followed by a sigmoid nonlinearity.

Sequential implementations would compute a fully-connected layer using a triple loop method of matrix multiplication. A GPU implementation would remove some of the loops and map the remaining logic to a space of indices. For our baseline fully-connected layer implementation we will use the baseline matrix multiplication kernel used by Rovder [17] in his work and amend it with bias addition and sigmoid activation, in line with our goal of a single-kernel implementation.

The resulting OpenCL implementation can be seen in Source 6.1. It assumes that matrices are in row-major layout and is mapped onto an index range covering the rows and columns of the resulting matrix. Each work item computes a single element in the resulting matrix.

The `K` and `COLS` placeholders are filled in using compile-time definitions as the conjoining dimension and the output matrix column count respectively. This way we can re-use the same kernel repeatedly for each batch of inputs, in line with our goal of reusable kernels.

```
__kernel void mat_cross(
  __global const float* a, __global const float* b,
  __global float* c,
  __global const float* biases
){
    size_t row = get_global_id(0);
    size_t col = get_global_id(1);
    float cumulative = 0.0;
    for(int i = 0; i < K; i++){
        cumulative += a[row*K + i] * b[col + i*COLS];
    }
    c[row*COLS + col] =
      1.0 / (1 + exp(-cumulative - biases[col]));
}
```

Source 6.1: OpenCL implementation of a fully-connected layer. (Assumes matrices in row-major layout). K = conjoining dimension size, COLS = output matrix columns.

6.1.2 Convolutional Layers

Convolutional layers apply a filter onto a field of view across all the channels of each input. In our baseline implementation we are simply going to implement a loop-based variant of convolution, avoiding any memory blowup that is introduced by alternative methods such as im2col.

Placeholder	Value
N	Amount of inputs
XC	Amount of channels in each input value
XH	Height of input values
XW	Width of input values
YC	Amount of channels in each output value
YH	Height of output values
YW	Width of output values
FH	Filter height
FH	Filter height
SW	Convolution stride across the width (X dimension).
SH	Convolution stride across the height (Y dimension).

Table 6.1: Naive convolution implementation placeholders and their meaning

The loop-based implementation consists of seven nested loops: we must compute every column of every row of every feature map of every output (4 loops, which we will call the *outer loops*), and to compute each of these values we must iterate over every column of every row of every feature map in the corresponding filter (3 loops, which we will call the *inner loops*).

So far we have always mapped a kernel onto a one-dimensional or two-dimensional NDRange. OpenCL, however, allows us to map a kernel onto three-dimensional NDRanges as well, meaning we can kernelise up to three of the loops.

When mapping loops of a computation onto a space of indices it is important to group loops that operate on the same data together, to avoid race conditions. In this particular example it is quite obvious that the three inner loops must be executed within the same work item, since they cumulatively compute a single value.

The loops chosen to be mapped onto the index space were the loops over features, rows and columns of the outputs. Thus, the kernel will contain the three inner loops as well as the outer loop over the individual inputs themselves. The reason we chose this approach stems from the general convolutional model construction method, which aims to keep the amount of propagating values constant: convolutional layers increase the number of values by increasing the amount of features, while pooling/subsampling layers reduce the number of values by reducing the width and height of the features. As such, the number of output values in every input stays approximately constant and, consequently, the number of GPU work items will also remain approximately constant with every convolutional layer. This should ensure consistent performance across the network and is a good design choice for our baseline implementation.

The kernel implementation of this layer can be seen in Appendix B. Once again, we have integrated bias addition and the sigmoid activation into the layer directly, in line with our single-kernel implementation goal.

There are also numerous compile-time placeholders in the code, which are explained in Table 6.1.

6.1.3 Subsampling Layers

Subsampling layers are functionally very similar to convolutional layers. We exploited this similarity in Section 5.1.3 when we implemented subsampling layers as convolution with sparse filters.

For our baseline custom implementation we are, however, not going to re-use the convolutional kernel from the previous subsection. Instead we are going to implement a custom kernel for subsampling. This addresses the third and final goal of our initial implementation by removing the redundant computation from subsampling altogether.

The primary difference between convolution and subsampling from a GPU perspective is that subsampling only has 2 inner loops while convolution has 3. The loop across input features required by convolution (seen in Source B.1 line 14) is not needed for subsampling, since every item in the resulting feature map only depends on a patch of inputs from one single input feature map. Otherwise, the index space onto which a subsampling kernel is mapped is exactly the same as that of the convolutional layer: it is mapped over indices corresponding to the output width, height and channel count. Internally, each kernel computes its corresponding output value across all output images. This is done using a loop, much like was done in the convolutional layer.

6.2 Results and Discussion

Combining the three custom layer implementations from the previous section, we can fully execute LeNet on the Mali T-628 GPU. To provide a fair benchmark against the performance of CLBlast, we will use the same setup as we used in Section 5.2. We will forward propagate a batch of 100 input images through the custom LeNet implementation and measure its run time.

We timed both kernel time and total wall time[2]. Timing both is important because CLBlast does not give us access to the underlying OpenCL `Event` objects of the executed kernels, meaning the measurements seen in Section 5.2 are wall time. Comparing CLBlast's wall time to our kernel time would not be a representative comparison, so we shall compare wall times. The reason for timing kernel time as well is to give us insight into the kernel initialisation overhead.

The resulting times can be seen in Table 6.2 as well as Figure 6.1. There are a number of observations we may take from these results.

First of all, we now have concrete measurements of kernel initialisation times. We can find this value by subtracting the kernel time from the wall time, which gives us the infrastructural overhead of executing the GPU kernel in the first place. This overhead varies with the kernel itself yet remains between 16 and 31 milliseconds for all of them, which is substantially less than the 76 millisecond overhead CLBlast demonstrated in Section 4.2.2. This result is expected, as our implementation compiles each kernel

[2]Wall time being the actual amount of time the operation appeared to take

Layer	CLBlast	Our custom system			Wall Speedup
		Wall time	Kernel time	Overhead	
C1	721.14	34.81	14.49	20.32	**20.72**
S2	579.78	17.30	1.02	16.28	**33.52**
C3	681.30	60.04	40.27	19.77	**11.35**
S4	550.48	17.35	0.63	16.72	**31.72**
F5	416.53	33.90	4.20	29.70	**12.29**
F6	405.32	30.92	0.77	30.15	**13.11**
Total	3354.55	194.31	61.37	132.94	**17.26**

Table 6.2: Comparison of CLBlast runtime to our initial baseline implementation runtime at forward propagating 100 inputs through LeNet (times in milliseconds)

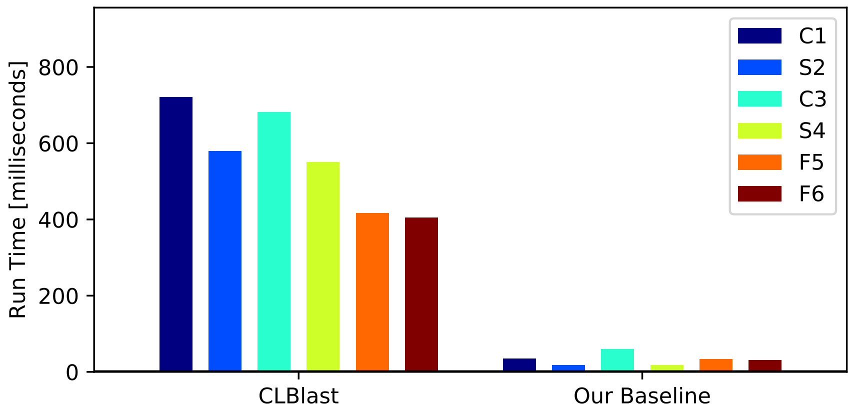


Figure 6.1: Comparison of CLBlast runtime to our baseline custom implementation runtime at forward propagating 100 inputs through LeNet

only once and recycles the compiled kernel for every forward propagated batch on inputs. CLBlast re-compiles the kernels with every call to the subroutines and does not support recycling in any way. The results in Table 6.2 clearly show how impactful this disadvantage is. Reusing kernels cuts down the initialisation time of our kernels by half compared to CLBlast.

Second of all, these results strongly indicate that even an unoptimised basic OpenCL implementation can outperform CLBlast by a significant margin. Claiming that any naive baseline implementation can outperform any competing state of the art system is a bold statement and we did not make it without justification. Upon measuring these results, out initial intuition was that these results are incorrect, since the performance difference was so drastic. We double-checked these results using alternative timing methods, verifying outputs (and intermediate computation results) against a third system [3] and measuring the full wall time of our entire benchmarking system (including all possible overheads) using the Unix **time** utility and the results undistupably checked out.

[3]We used a numpy implementation of all layers to verify every intermediate result of the propagation

6.3 Summary

In this Chapter we wrote custom baseline implementations of neural network layers in OpenCL and demonstrated how they outperform CLBlast at executing LeNet on the Mali T-628 GPU. We did this to test the hypothesis that OpenCL kernels manually optimised for the Mali T-628 GPU can outperform their automatically tuned CLBlast counterparts on all benchmarks. In a way, we proved this hypothesis to be true since we did outperform CLBlast. On the other hand, the hypothesis explicitly mentioned optimising the kernels, which we have not done since it was our baseline that already outperforms CLBlast.

We expected some improvement over CLBlast, since we did addressed the three main shortcomings of CLBlast identified at the beginning of this chapter. We did not expect the improvement to be this significant. It would seem that the shortcomings we identified were the primary bottleneck of CLBlast's performance at LeNet inference.

The main issue we face now is that the results from Table 6.2 indicate that kernel initialisation is the primary component of our baseline's inference time, meaning we cannot meaningfully improve upon these results. Optimising the layer kernels would result in a reduction of kernel time, which would have insignificant impact on the overall runtime of many of the layers. Take F6 as an example: the overall run time of F6 is 30.15 milliseconds, 0.77 of which is kernel time.

We did not expect our baseline to be this performant at LeNet inference.

As such, we shall conclude that our baseline outperforms CLBlast's inference speed by a factor of 17, which is a sufficient improvement over state-of-the-art for small models like LeNet, and we shall now move on to a larger model: VGG-16.

Chapter 7

Executing VGG-16 with Custom System

This chapter outlines the process of adapting our baseline to VGG-16 and optimising it. In order to test our hypothesis from Section 1.2 we shall use the CLBlast VGG-16 benchmarks from Section 5.3 Figure 5.7 as our target metric and attempt to outperform it.

Loukadakis et al. succeeded in a similar task in “Accelerating Deep Neural Networks on Low Power Heterogeneous Architectures” so we shall use their results as a reference as well and attempt to accelerate VGG-16 beyond them.

7.1 Adapting the Baseline to VGG-16

Before we execute VGG-16 with our custom system we must adapt it to support the additional requirements of this more complex network.

Before we add functionality, however, we are also able to remove some: since we are only forward propagating one input image at a time, all code relevant to iteration over individual inputs is no longer relevant. This reduces the complexity of all our baseline convolution kernels by removing one of the `for` loops from the kernel.

The first major requirement of VGG-16 is feature map size preserving convolution (using zero padding) which was discussed in Section 5.3. As far as a convolutional layer is concerned there is no difference between a zero-padded input and a non-padded input. The only information a convolutional layer requires to be able to operate on padded data is whether the output feature maps are padded or not, so as to be able to place results into correct locations in memory.

We can easily adapt the kernel from Appendix B to include padding by adding the relevant offsets to lines 33 to 36 using compile time definitions much like we did with all other constant terms from Table 6.1. The new placeholders are defined in Table 7.1. Using these we can ensure that every convolutional layer pre-pads the data for the

Placeholder	Value
YPH	Total vertical output matrix padding
YPHT	Total padding on top of output matrix feature maps
YPW	Total horizontal output matrix padding
YPWT	Total padding on the left of output matrix feature maps

Table 7.1: Extension to Table 6.1: output matrix padding placeholders

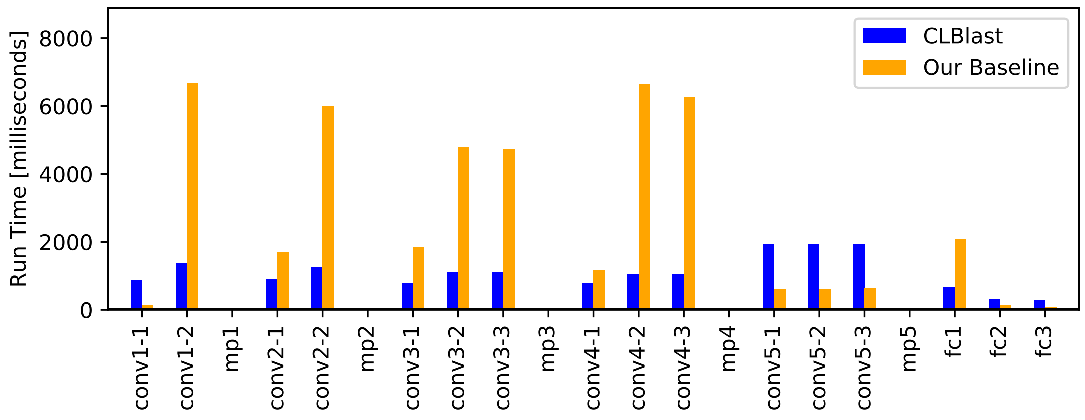


Figure 7.1: Comparison of CLBlast runtime to our baseline layer implementation run time of forward propagating a single input through VGG-16

subsequent layer, meaning as long as the initial inputs into the network are padded, the padding is preserved across all layers.

The final kernel also includes ReLU activation instead of the sigmoid used with LeNet and can be seen in Appendix C.

The second requirement for VGG-16 is a max-pooling layer implementation. While max-pooling layers are very straightforward to implement, we must ensure that padding produced by the preceding convolutional layer or required by the subsequent convolutional layer is properly handled. This is done using the same compile time placeholder technique we use for convolution. We also allow our max-pooling layers to optionally strip down the padding in order to bridge the connection between the last max-pooling layer and the first fully-connected layer (fully-connected layers do not expect zero padding).

Using this adapted baseline to forward propagate one image over VGG-16 yielded the results shown in Figure 7.1. As expected, our baseline does not perform very well at executing large convolutional layers, taking a total of 44.24 seconds to forward propagate the image. As opposed to the LeNet results in Table 6.2, however, the VGG-16 run times are mostly kernel time, so we may now proceed with optimising these kernels to drive the total inference time down by meaningful amounts.

7.2 Optimising Convolution

Since convolution is the primary performance bottleneck of our network, we optimise this kernel first.

From our previous work as well as research from ARM [25], we know that optimising memory layouts and workgroup sizes for better memory access patterns, and utilising vector SIMD instructions are the two most effective ways to optimise Mali kernels. These are the optimisations we will employ. Loukadakis et al. [11] attempted to use these optimisations to accelerate VGG-16 as well, though their work primarily focused on the im2col and GEMM approach while we focus on the iterative approach, in line with our goal to avoid the im2col memory blowup. Comparing the results should yield insight into which approach is more effective on the Mali GPU.

7.2.1 Memory Layouts and Workgroup Optimisation

Every work item that runs on the GPU will operate on data in memory. Fetching the data from memory into the cache is expensive, so our goal is to ensure maximum overlap across the data used by all work items across all work groups executing together at any given point in time. This can be done by optimising memory layouts and specifying workgroup sizes.

So far we have allowed OpenCL to execute our kernels across the NDRange in any order it chooses under the hood. By specifying workgroup sizes, OpenCL can be forced to execute chunks of the NDRange in a particular order of our choice. The memory layouts of data can then be modified to achieve the desired data overlap under the forced chunk ordering, which improves kernel execution time.

Intuitively, we know that computing a tightly packed cube of values in the convolution output will yield maximum data overlap across the work items required to compute that cube of output values [1]. However, the solution to this problem is not as simple as setting the workgroup dimensions to small cubes, since there are several other factors that need to be considered.

The first factor to consider is that there are four workgroups executing in parallel on the GPU. These four workgroups must compute the tightly packed cube of outputs together, meaning the shape of a workgroup should resemble a thin rectangular prism with a large square face along which the four workgroups line up to form the cube, rather than making the workgroups cubes themselves.

The second factor to consider is the order in which the workgroups iterate over the NDRange. Rovder [17] has demonstrated that workgroups iterate over global id 0 first, though along four stripes of global id 1s at the same time, followed by iterating over global id 1 and finally global id 2 (see Figure 2.1 for global id definitions). We can use this knowledge to narrow down the space of possible optimal workgroup sizes.

[1]This is similar to understanding how computing a tightly packed square of the output matrix of a matrix multiplication operation achieves the most data overlap in the original two matrices.

Since global id 0 is iterated over first, the workgroups should only cover one global id 0 each (creating the thin prism). We know that global id 0 is iterated over in stripes of four global id 1s, which restricts the dimensions of our desired cube shape to 4 ids along each coordinate. As such, the ideal workgroup size should be (1, 4, 4). Small variations on these dimensions can be used to increase or decrease the memory requirements depending on the amount of feature maps in the layer. Other reasonable workgroup sizes may include: (2, 2, 4), (1, 2, 4), (1, 1, 4).

There are several different memory layouts commonly used for convolution. The layout we used until now lists feature maps in memory one by one, each in row major layout. This is known as the NCHW layout (**N**umber of inputs, **C**hannels, **H**eight, **W**idth). We know from Rovder [17] that the size of a Mali cache line is exactly 16 floats, which, in combination with our workgroup sizes, does not yield optimal memory usage. Since we compute a 4x4x4 cube of the output at a time, we will need Cx6x6 floats from the input matrix[2]. If the input matrix is in NCHW layout, this will result in only 36% data utilisation, since cache lines will span 16 columns of the feature map while we only need 6.

A much more sensible layout is the NHWC layout, which allows us to fetch a chunk of Cx6x6 floats from the input matrix into the cache with no redundancy. To preserve locality, the filters were also be transformed into this layout.

We executed VGG-16 with these modifications, yielding the results shown in Figure 7.2 and Table 7.2. Exactly as expected, the optimisations improved the inference time by a considerable margin, though not enough to outperform CLBlast. The decline in performance for conv5 layers compared to our baseline was not something we expected, though, upon investigation, we discovered the cause: at 14x14 in size, the feature maps are small enough to completely fit into the cache and are also small enough to be computed by a single baseline workgroup each. As such, while running our baseline kernel, the GPU could slowly stream through four filters, consistently keeping the current feature map in the cache for all work items to use. This is a coincidence that arises from the feature map being so close to 16x16 in size, which is the exact size of a workgroup, and it cannot be relied on across layers of any other size.

7.2.2 Vectorisation

The Mali T-628 GPU's primary strength lies in its SIMD instructions, specifically with the OpenCL `dot` function. On the Mali T-628, the `dot` function executes on dedicated hardware, meaning we can perform the dot products concurrently with any other regular floating point multiplication. This essentially means that the computational power of the GPU *cannot* be fully utilised without making use of the `dot` function.

Due to the NHWC memory layout we introduced in the previous section, adding vectorisation to our code is trivial: we reduce the **XC** compile time definition by a factor of 4, change the kernel function signature to include inputs and filters as `float4` types, and we replace the innermost loop multiplication with a call to `dot`.

[2]Not Cx4x4, because filters are 3x3.

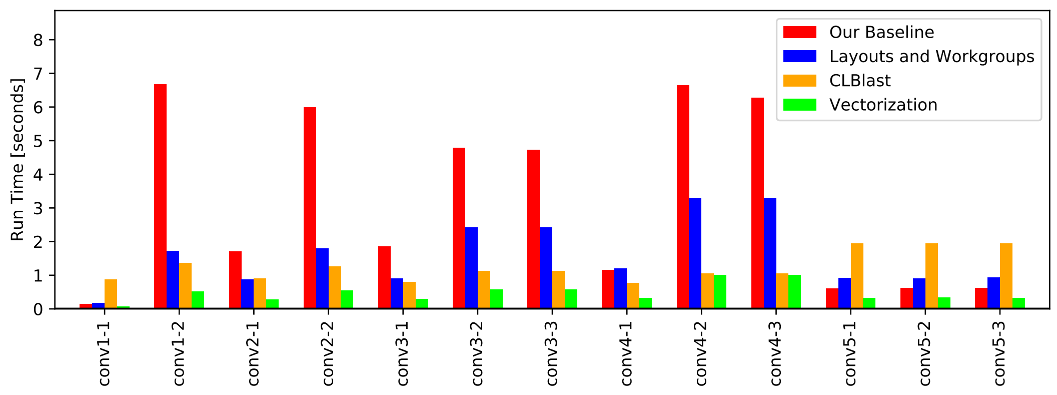


Figure 7.2: Inference time under various optimisations compared to baseline and CLBlast

Layer	CLBlast	Our Baseline	Layouts and Workgroups	Vectorisation
conv1-1	0.87	0.15	0.17	**0.07**
conv1-2	1.36	6.67	1.73	**0.51**
conv2-1	0.89	1.71	0.87	**0.28**
conv2-2	1.26	5.99	1.79	**0.55**
conv3-1	0.80	1.85	0.90	**0.28**
conv3-2	1.12	4.79	2.42	**0.57**
conv3-3	1.12	4.73	2.42	**0.57**
conv4-1	0.77	1.16	1.20	**0.32**
conv4-2	1.05	6.64	3.29	**1.01**
conv4-3	1.05	6.27	3.29	**1.01**
conv5-1	1.95	0.61	0.91	**0.33**
conv5-2	1.95	0.62	0.91	**0.34**
conv5-3	1.95	0.62	0.93	**0.33**
Total	16.14	41.81	20.83	6.16

Table 7.2: Inference time under various optimisations compared to baseline and CLBlast (times in seconds)

7.2.3 Summary

The results in Figure 7.2 and Table 7.2 show the performance differences between the variously optimised systems. Memory layout and workgroup size optimisations were not sufficient for outperforming CLBlast. Vectorisation, however, managed to outperform the library. This could be explained by CLBlast itself making use of vector types as well, an assumption that sees considerable evidence. CLBlast has been observed to perform matrix multiplication on the Mali T-628 at 8 GFLOPS by Nugteren et al. [22], and this same performance was also observed independently by Gronqvist and Lokhmotov [25] as well as Rovder [17] for vectorised GEMM kernels.

The overall inference time of all convolutional layers of **6.16** seconds is 27% faster than the 8.5 second time published by Loukadakis et al. in "Accelerating Deep Neural Networks on Low Power Heterogeneous Architectures". Optimisations we used were similar to those used by Loukadakis et al. and the reason for the observed speed difference is indeterminable due to limited knowledge of their system design.

7.3 Convolution with Morton GEMM

Based on our findings from Chapter 5, we identified the main CLBlast limitations and postulated guideline techniques to mitigate them (Section 6.1). These guidelines dictated the process of designing and optimising our kernels in subsequent sections. In this section, however, we deviate from these guidelines to investigate further potential performance improvements, which may come at the cost of extraneous memory usage.

So far we have implemented variants of convolution that iteratively operated on data directly in either NCHW or NHWC layouts. This was to avoid the memory blowup caused by im2col, which was unavoidable using CLBlast. Our iterative convolution kernels, however, do not fully utilise the computational power of the GPU, which we know from Rovder [17] to be 17 GFLOPS. To push performance up even further, we investigate the im2col-GEMM convolution method [28] and attempt to make it perform better than the CLBlast implementation.

In Section 3.3, we mention the Morton GEMM kernel published by Rovder [17], which was optimised for performing matrix multiplication on the Mali T-628 GPU at 13.5 GFLOPS. We can estimate the amount of floating point operations required to forward propagate one input over VGG-16 convolutional layers as roughly 31 billion. This places the lower bound of VGG-16 inference time using the Morton GEMM kernel with im2col at 2.3 seconds, which is much lower than the 6.16 inference time achieved in the previous section.

Provided we can implement the required infrastructural kernels needed to enable im2col and optimise them to a point where their combined execution time in combination with Morton GEMM execution is less than the minimum 6.16 seconds reached in the previous section, this method may be preferred to our vectorised iterative approach.

7.3.1 Surrounding Infrastructure

In order to execute convolution using the Morton GEMM kernel, we must implement the following operations as OpenCL kernels:

Padding – we need an operation that is capable of adding or removing padding from feature maps in in NCHW layout. This operation will be used to prepare data for undergoing the im2col rearrangement such that the width and height are preserved after the convolution takes place [3].

Zeroing – to avoid the memory blowup im2col introduces, we will reuse a single pair of OpenCL buffers across the entire network. In order to safely reuse buffers, we must have an operation that allows us to wipe the buffer by setting it to zeros.

Im2col – we need an implementation of im2col, much like CLBlast has.

Other operations we need include Morton GEMM, which can be found in Rovder's paper [17], and the max-pooling layer we used in Section 7.1.

While writing zeroing and padding kernels is trivial, im2col is more involved and thus we dedicate a small section to it.

Im2col

Supposing we have a $X_d \times X_h \times X_w$ matrix of feature maps and we intend to perform convolution using filters of size $F_h \times F_w$ by strides of S_h and S_w along the height and width of the feature maps to transform X into a $Y_d \times Y_h \times Y_w$ matrix of feature maps, the dimensions $Y_h \times Y_w$ of the resulting matrix can be computed as follows:

$$Y_h = \frac{X_h - F_h}{S_h} + 1 \quad , \quad Y_w = \frac{X_w - F_w}{S_w} + 1$$

Subsequently, we can use these values to compute the dimensions of the intermediate matrix I (bottom right matrix in Figure 5.1) populated by im2col as:

$$I_h = Y_d \times F_h \times F_w \quad , \quad I_w = Y_h \times Y_w$$

Implementing im2col now becomes a matter of mapping the x, y, z indices of X onto the r, c (row, column) indices of I.

A simple method of mapping x, y, z onto r, c is to map the im2col kernel onto the three dimensions of Y, even though matrix Y is not even a part of this operation. From the coordinates within Y we can compute x, y, z *and* r, c mapping pairs, meaning the mapping can be performed.

Since $I_w = Y_h \times Y_w$, mapping the im2col kernel onto Y_h and Y_w ensures that a single work item populates a single column of I. This can be further subdivided using a third NDRange index of X_d to ensure each work item populates a $F_h \times F_w$ tall subcolumn of a column in I.

[3]Unpadded convolution reduces the size of the feature maps like seen in LeNet in Figure 5.4. To preserve feature map dimensions like VGG-16 does in Figure 5.6, padding needs to be added along the edges of the feature maps before convolution is done.

The kernel for this mapping can be found in Appendix D. Investigating the CLBlast source code we found that CLBlast uses a similar kernel for the `xIM2COL` subroutine.

OpenCL Memory Reuse

Since VGG-16 contains a total of 13 convolutional layers, creating a separate buffer to hold the output of each im2col operation would be extremely redundant. Instead, we opt for an approach that involves buffer reusability.

Once the neural network is constructed, we compute the maximum possible output size of any particular kernel within this network. For VGG-16, the maximum output size is 28,901,376 `floats`, which is the output of im2col preceding conv1-2 (see Figure 5.6).

We construct two buffers of this size to hold any intermediate results and we always zero a buffer before any operation writes to it. This is a redundant operation from the inference perspective, however, it is also very low cost and necessary to keep memory usage at a minimum.

7.3.2 Morton GEMM Integration

In order to perform Morton GEMM we need the input matrices to be in the layouts depicted in Figure 3.1.

Filters used by convolutional layers do not change after the network is initialised, so we can reshape them in main memory and place them into the buffer when they already are in R_2_4_R layout (see Figure 3.1 for layout visualisation).

Ensuring that the propagated data is in C_4_2_C can be done directly in the im2col kernel, which always precedes Morton GEMM, saving us the need to call an extra reshaping kernel. Once our im2col kernel computes the content at the r, c coordinates of matrix I, it can compute the corresponding index within the C_4_2_C layout and place the data there instead.

Another issue to keep in mind is that Rovder's Morton GEMM kernel required work-groups of particular sizes, forcing the rows and columns of the output matrix to be multiples of 8 and 32 respectively. This is not an issue for our memory buffer sizes, since they will necessarily be much larger than the output of Morton GEMM (see previous section). It is, however, an issue for the NCHW layout that should be produced by Morton GEMM, which will now have extraneous terms dangling off the end of every feature map. These extraneous values can be removed by our padding kernel from Section 7.3.1 by passing the full Morton GEMM output to it as if it was a single feature map with C rows and HW columns. This trick allows us to remove the trailing values from each feature map, bringing the data back into proper unpadded NCHW layout.

All padding, zeroing, reshaping and multiplication considered, the convolutional layers become a sequence of 8 calls to 4 different kernels:

1. **Zeroing** – Zeroing one of the two buffers.
2. **Padding** – Add padding around the feature maps of the incoming data in NCHW layout.
3. **Zeroing** – Zeroing one of the two buffers.
4. **Im2col** – Reshape the data into a column matrix.
5. **Zeroing** – Zeroing one of the two buffers.
6. **Morton GEMM** – Perform convolution.
7. **Zeroing** – Zeroing one of the two buffers.
8. **Padding** – Removing the trailing values left at the end of each feature map by Morton GEMM.

This is a completely different approach from the ones in previous chapters, where we restricted our layer implementations to a single kernel. We are relying here on the performance boost from Morton GEMM outweighing the cost of instantiating so many kernels.

We can also attempt to mitigate the amount of zeroing kernels. The zeroing kernels are put in place to ensure that each buffer will be fully overwritten by the kernel that wrote to it last and contains no residual values previously written to it. If we know that an operation will certainly overwrite all data already in a buffer, we may skip the zeroing kernel.

An example of this is the zeroing kernel in step 3, where the output of im2col is always larger than the input data passed to the convolutional layer, guaranteeing a full overwrite of the non-zero values in the buffer. We may similarly skip the zeroing kernel in step 5 provided the output of Morton GEMM is larger than the padded inputs were.

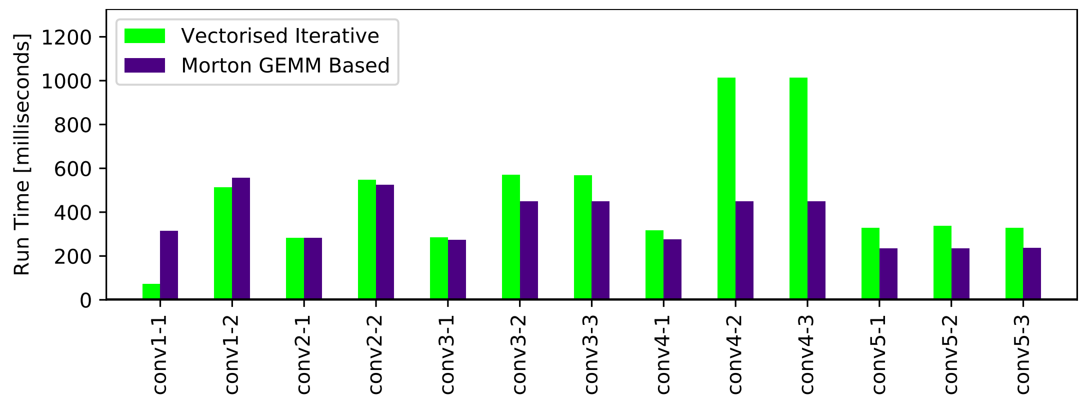


Figure 7.3: Comparing the performance of our vectorised iterative convolution kernel from Section 7.2.2 with our Morton GEMM based convolution

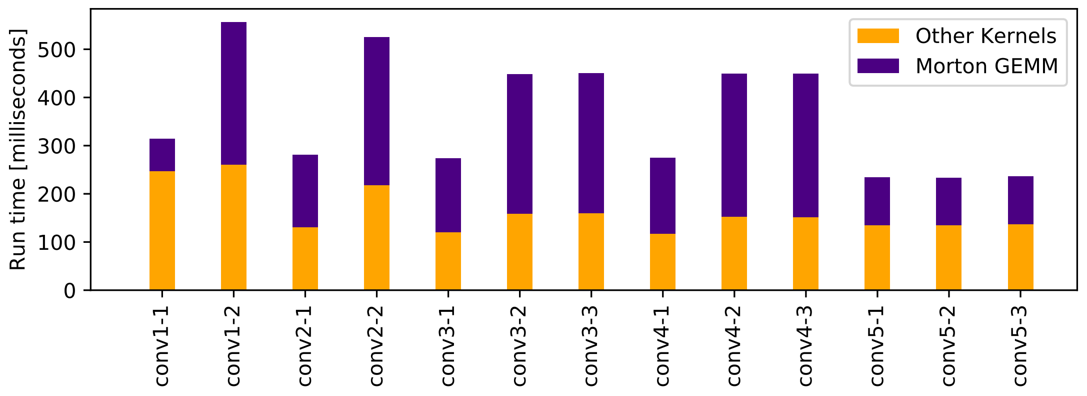


Figure 7.4: A breakdown of how computation time was distributed between Morton GEMM and other infrastructure kernels for our Morton GEMM based convolution

7.3.3 Results

We used the Morton GEMM layer implementations to forward propagate one input over VGG-16 and observed the results in Figure 7.3. Our intuition about the performance boost from Morton GEMM outweighing the overhead accumulated from infrastructure kernels was correct, and the Morton GEMM approach outperformed our iterative vectorised approach from Section 7.2.2 on most layers.

The performance gained by utilising Morton GEMM is most apparent on the deepest convolutional layers with the most feature maps, namely conv4 layers. The robust performance of Morton GEMM results in almost identical run times of the kernel across all convolutional layers, as seen in the breakdown in Figure 7.4.

On the other hand, Figure 7.4 also shows how much time was spent performing the surrounding operations facilitating Morton GEMM, which is for some layers more than the run time of Morton GEMM itself. The overhead is most felt on the conv1-1 layer, which performed extremely poorly compared to the vectorised variant, since 79% of its time was spent on the supporting kernels.

Layer	Morton GEMM	Infrastructure	Total	Useful Time
conv1-1	66.71	247.68	314.39	21%
conv1-2	295.42	260.60	556.02	53%
conv2-1	150.13	131.23	281.36	53%
conv2-2	307.10	217.93	525.03	58%
conv3-1	152.97	120.89	273.86	55%
conv3-2	289.01	158.73	447.74	64%
conv3-3	289.87	159.91	449.78	64%
conv4-1	157.79	117.25	275.05	57%
conv4-2	297.04	151.99	449.03	66%
conv4-3	297.13	152.32	449.45	66%
conv5-1	99.09	135.34	234.43	42%
conv5-2	98.07	135.47	233.53	41%
conv5-3	99.61	136.41	236.01	42%
Total	2599.93	2125.75	4725.69	55%

Table 7.3: A breakdown of how computation time was distributed between Morton GEMM and other infrastructure kernels for our Morton GEMM based convolution. (times in milliseconds)

Table 7.3 contains a detailed breakdown of the execution time and how it was spent across all convolutional layers.

Overall, the time spent performing Morton GEMM across all convolutional layers was 2.6 seconds, which is extremely close to our predicted lower bound of 2.3 seconds from the beginning of this section (the deviation being exactly equal to kernel initialisation time). Hence the execution time of Morton GEMM made up roughly 55% of the overall run time of 4.75 seconds. The Morton GEMM approach is 1.3 times faster than the fastest iterative approach tried in Section 7.2.2 and 3.4 times faster than the CLBlast version.

7.4 Optimising Fully-Connected Layers

We optimised convolutional layers first because they took the most time to compute in our baseline as seen in Figure 7.1. Now convolution has been optimised to the point where our fully-connected layers (mainly fc1) become the network bottleneck, forming 33% of our full VGG-16 inference time. This is a good reflection on our work from the previous sections, since we have optimised convolution to the point where its run time is on par with fully-connected layers.

We shall not delve too deep into optimising fully-connected layers, since we can reduce them from matrix-matrix multiplication to matrix-vector multiplication just as we did with CLBlast in Section 7.1 and add vectorisation. This sequence of optimisations already outperforms CLBlast and was used to attain the final results in the following section.

7.5 Results and Discussion

In this chapter we explored various methods of optimising our custom implementation of neural network layers to minimise VGG-16 inference time on the Mali T-628 GPU.

Primarily focusing on convolution, we implemented and optimised two different approaches to the operation. We optimised an iterative method by configuring work-group sizes and vectorising the kernels, and we optimised an im2col-GEMM approach by making use of the highly optimised Morton GEMM kernel we discovered in our previous work.

Table 7.4 and Figure 7.5 show the inference time breakdown across layers for the two fully-optimised systems, comparing their performance to that of CLBlast as well as to that of the custom OpenCL kernels published by Loukadakis et al. [11] earlier this year.

These results clearly show the superiority of our custom implementations over the other current state-of-the-art systems across all layers. We have successfully demonstrated our hypothesis to be true for VGG-16 by showing that exploiting detailed knowledge of the device architecture will outperform an automatically tuned system on all benchmarks.

The results, however, do *not* conclusively place one of our systems over the other. Each system has some benefits over the other system and we discuss them here now.

There is one important downside to the Morton GEMM approach: we cannot fit the entirety of VGG-16 into memory at one time if we also want to allocate memory for the im2col buffers. This is a peculiar downside of the Morton GEMM approach because the approach does not use much more memory than the iterative vectorised approach, and it would seem that the iterative approach was ever so slightly below the device memory cap all along. This means the network had to be timed in two parts, once to propagate over the convolutional layers and once to propagate over the fully-connected layers. It is worth noting that the largest chunk of the network is the weight matrix used in the fc1 layer, which contains a total of 102,764,544 trainable parameters, forming 74% of the 138,357,544 parameters of VGG-16. The convolutional layers are hence only a small part of the network, and varying the parameter counts there will not help us reduce the memory demands to a consequential degree. It is, nonetheless, a downside of the Morton GEMM approach.

On the other hand, an advantage of the Morton GEMM approach is that it scales very well to deeper layers as can be seen on conv4 layers in Figure 7.5. This means it is more likely to work fast for larger networks in future research. Since the actual matrix multiplication only contributes to 55% of the Morton GEMM convolution, there also remains great amount of optimisation potential to this implementation. Reducing the magnitude of infrastructural overhead could improve the overall performance of this approach significantly. Furthermore, since the approach reuses memory, the memory overhead of im2col becomes negligible with the amount of convolutional layers, since there will only ever be two buffers to store intermediate results in.

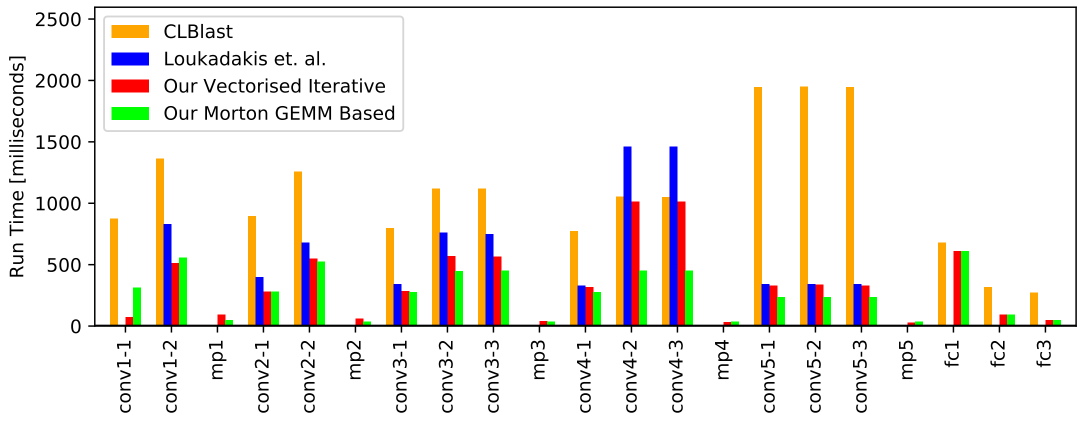


Figure 7.5: Comparison of VGG-16 inference times by layer of our two systems and two state-of-the-art systems (missing or unsupported metrics omitted)

Layer	CLBlast	Loukadakis et al.	Our Work		Our Speedup Over	
			Vectorised	Morton GEMM	CLBlast	Loukadakis et al.
conv1-1	874.732	–	**71.756**	314.390	12.19	–
conv1-2	1362.894	830.000	**513.390**	556.019	2.65	1.62
mp1	–	–	94.202	**49.427**	–	–
conv2-1	894.872	400.000	**281.243**	281.358	3.18	1.42
conv2-2	1257.292	680.000	547.064	**525.032**	2.39	1.30
mp2	–	–	59.221	**37.116**	–	–
conv3-1	796.761	340.000	284.008	**273.863**	2.91	1.24
conv3-2	1117.005	760.000	568.757	**447.739**	2.49	1.70
conv3-3	1117.776	750.000	566.790	**449.776**	2.49	1.67
mp3	–	–	40.145	**34.634**	–	–
conv4-1	772.710	330.000	315.924	**275.046**	2.81	1.20
conv4-2	1052.089	1460.000	1012.383	**449.034**	2.34	3.25
conv4-3	1050.759	1460.000	1011.882	**449.453**	2.34	3.25
mp4	–	–	**31.477**	33.719	–	–
conv5-1	1946.488	340.000	327.361	**234.432**	8.30	1.45
conv5-2	1947.286	340.000	336.554	**233.532**	8.34	1.46
conv5-3	1946.385	340.000	327.602	**236.013**	8.25	1.44
mp5	–	–	**28.797**	33.744	–	–
fc1	677.196	–	**608.650**	**608.650**	1.11	–
fc2	315.579	–	**92.939**	**92.939**	3.40	–
fc3	272.650	–	**46.313**	**46.313**	5.89	–
Total	17402.473	8030.000	7166.458	**5662.229**	3.07	1.42

Table 7.4: Comparison of VGG-16 inference times by layer of our two systems and two state-of-the-art systems in milliseconds (missing or unsupported metrics omitted)

7.6 Correctness

This chapter provides results that are considerably better than state-of-the-art results published by previous research and it is natural to ask what measures were taken to ensure the correctness of our implementation. This section aims to outline the testing method used.

We adopted a continuous testing strategy for this project. Every fundamental operation was first implemented in Python using the numpy library. These operations included equality verification, alignment computation, memory insertion, im2col, convolution, max-pooling, transformations between NCHW and NHWC layouts, as well as transformations to and from the two Hybrid Morton Order Layouts. For these operations we implemented test cases by hand with custom hard-coded values. These can be found in `refs.py`.

All OpenCL operation implementations were tested against these pre-tested numpy implementations automatically using random data, forming a chain of validity which originates at the very bottom numpy implementations and extends all the way to the top level implementations of neural network layers.

As an example of this system in action, we can point to the `ConvolutionalLayer.py` file, which contains the implementation of our 8-stage Morton GEMM based convolution from Section 7.3.2. Each of the 8 stages were imported from their own Python implementation file, each of which contained a test method for testing the stage independently. The `ConvolutionalLayer.py` file also contains a method to test the full stack of the 8 stages. Each of these test cases tests the appropriate stage or layer using the pre-tested numpy implementations, ensuring that the validity is preserved as the implementation stack grows.

Chapter 8

Conclusion and Evaluation

We have successfully shown that a hand-tuned OpenCL neural network implementation can outperform competing state-of-the-art automatically tuned systems on the Mali T-628 GPU. Our custom system showed a speedup of 17x for LeNet and 3x for VGG-16 when benchmarked against the recently published CLBlast library. Our best optimised system also showed a speedup of 1.42x over the most recent research into hand-tuning OpenCL kernels for VGG-16 inference on the same GPU, published by Loukadakis et al. [11].

In Chapter 1 we set out the following goals for our work:

1. Use CLBlast subroutines to implement as many layers specified in Table 1.1 as possible given the library's range of support.
2. Use the CLBlast implementation to forward propagate 100 inputs over LeNet and a single input over VGG-16, investigating performance and behavior.
3. Implement all functionality required for layers specified in Table 1.1 in custom OpenCL kernels.
4. Use the custom implementation to forward propagate 100 inputs over LeNet and a single input over VGG-16, investigating performance and behavior.
5. Optimise the custom implementation to outperform CLBlast on all benchmarks.

The goals we set up in Chapter 1 were fully achieved.

Future research could attempt to minimise the overhead of infrastructural kernels we used to support performing convolution using Morton GEMM. The overheads are significant, as shown in Figure 7.4, and removing the overhead could lead to a further 1.6x speedup over our fastest implementation.

Bibliography

[1] Y. Bengio, A. C. Courville, and P. Vincent, "Unsupervised feature learning and deep learning: A review and new perspectives," *CoRR*, vol. abs/1206.5538, 2012. [Online]. Available: http://arxiv.org/abs/1206.5538

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li, "Imagenet large scale visual recognition challenge," *CoRR*, vol. abs/1409.0575, 2014. [Online]. Available: http://arxiv.org/abs/1409.0575

[3] M. Ravanelli, P. Brakel, M. Omologo, and Y. Bengio, "A network of deep neural networks for distant speech recognition," *CoRR*, vol. abs/1703.08002, 2017. [Online]. Available: http://arxiv.org/abs/1703.08002

[4] 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/

[5] 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.

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

[7] 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

[8] "Cloud vision api," Google. [Online]. Available: https://cloud.google.com/vision/

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, 1998, pp. 2278–2324.

[10] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[11] M. O. Manolis Loukadakis, Jose Cano, "Accelerating deep neural networks on low power heterogeneous architectures," The University of Edinburgh.

[12] 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

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

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

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

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

[17] S. Rovder, "Evaluating neural networks on low powered gpus," 2017. [Online]. Available: http://honours.rovder.com/SimonRovderHonours.pdf

[18] "Cublas library," NVIDIA Corporation. [Online]. Available: https://developer.nvidia.com/cublas

[19] A. Krizhevsky, "Cuda-convnet, 2014." [Online]. Available: code.google.com/p/cuda-convnet/

[20] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[21] *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

[22] C. Nugteren, "Clblast: A tuned opencl BLAS library," *CoRR*, vol. abs/1705.05249, 2017. [Online]. Available: http://arxiv.org/abs/1705.05249

[23] "clblas." [Online]. Available: https://github.com/Yangqing/caffe/wiki/Convolution-in-Caffe:-a-memo

[24] C. Nutegren, "Clblast: A tuned blas library for faster deep learning," 2017. [Online]. Available: https://cnugteren.github.io/downloads/CLBlast_GTC.pdf

[25] 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

[26] K. Chellapilla, S. Puri, and P. Simard, “High Performance Convolutional Neural Networks for Document Processing,” in *Tenth International Workshop on Frontiers in Handwriting Recognition*, G. Lorette, Ed., Université de Rennes 1. La Baule (France): Suvisoft, Oct. 2006, http://www.suvisoft.com. [Online]. Available: https://hal.inria.fr/inria-00112631

[27] P. G. U. hausser, “Convolution in caffe: a memo.” [Online]. Available: https://github.com/clMathLibraries/clBLAS/wiki

[28] A. Vasudevan, A. Anderson, and D. Gregg, “Parallel multi channel convolution using general matrix multiplication,” *CoRR*, vol. abs/1704.04428, 2017. [Online]. Available: http://arxiv.org/abs/1704.04428

Appendix A

CLBlast VGG-16 Run times breakdown

Layer	Subroutine	Run Time	Total Run Time
conv1-1	`xCOPY`	106.85	874.73
	`xIM2COL`	164.29	
	`xGEMM`	603.59	
conv1-2	`xCOPY`	105.69	1362.89
	`xIM2COL`	228.71	
	`xGEMM`	1028.49	
conv2-1	`xCOPY`	98.80	894.87
	`xIM2COL`	128.30	
	`xGEMM`	667.77	
conv2-2	`xCOPY`	98.89	1257.29
	`xIM2COL`	162.25	
	`xGEMM`	996.16	
conv3-1	`xCOPY`	95.16	796.76
	`xIM2COL`	111.61	
	`xGEMM`	589.99	
conv3-2	`xCOPY`	95.11	1117.00
	`xIM2COL`	128.35	
	`xGEMM`	893.54	
conv3-3	`xCOPY`	95.20	1117.78
	`xIM2COL`	128.51	
	`xGEMM`	894.06	
conv4-1	`xCOPY`	92.68	772.71
	`xIM2COL`	103.27	
	`xGEMM`	576.75	
conv4-2	`xCOPY`	92.54	1052.09
	`xIM2COL`	111.50	
	`xGEMM`	848.05	
conv4-3	`xCOPY`	92.74	1050.76
	`xIM2COL`	111.55	

	xGEMM	846.47	
conv5-1	xCOPY	92.11	1946.49
	xIM2COL	98.30	
	xGEMM	1756.07	
conv5-2	xCOPY	92.11	1947.29
	xIM2COL	98.09	
	xGEMM	1757.09	
conv5-3	xCOPY	92.01	1946.38
	xIM2COL	98.09	
	xGEMM	1756.29	
fc1	xCOPY	91.69	677.20
	xGEMV	585.50	
fc2	xCOPY	91.83	315.58
	xGEMV	223.75	
fc3	xCOPY	75.62	272.65
	xGEMV	197.03	
Total			17402.47

Table A.1: Breakdown of forward propagation time of a single input through VGG-16 using CLBlast (times in milliseconds)

Appendix B

Baseline Convolution Kernel

```
__kernel void convolve(
  __global const float* inputs,
  __global const float* filters,
  __global const float* biases,
  __global float* outputs
){

  size_t out_feature = get_global_id(2);
  size_t row = get_global_id(0);
  size_t col = get_global_id(1);

  for(uint image = 0; image < N; image++){
    double cumulative = 0.0;
    for(uint feature = 0; feature < XC; feature++){
      for(uint y = 0; y < FH; y++){
        for(uint x = 0; x < FW; x++){
          cumulative +=
            inputs[
              XW*XH*XC*image +
              XW*XH*feature +
              XW*(row*SH+y +
              (col*SW + x)
            ] * filters[
              FW*FH*XC*out_feature +
              FW*FH*feature +
              FW*y +
              x
            ];
        }
      }
    }
    outputs[
      YW*YH*YC*image +
      YW*YH*out_feature +
      YW*row +
      col
    ] = 1.0 / (1 + exp(-(float)cumulative + biases[out_feature]));
  }
}
```

Source B.1: OpenCL implementation of a convolutional layer

Appendix C

Padding-enabled baseline convolution kernel

```
__kernel void convolve(
  __global const float* inputs,
  __global const float* filters,
  __global const float* biases,
  __global float* outputs
){

  size_t out_feature = get_global_id(2);
  size_t row = get_global_id(0);
  size_t col = get_global_id(1);

  double cumulative = 0.0;
  for(uint feature = 0; feature < XC; feature++){
    for(uint y = 0; y < FH; y++){
      for(uint x = 0; x < FW; x++){
        cumulative +=
          inputs[
            XW*XH*feature +
            XW*(row*SY+y) +
            (col*SX + x)
          ] * filters[
            FW*FH*XC*out_feature +
            FW*FH*feature +
            FW*y +
            x
          ];
      }
    }
  }
  cumulative += biases[out_feature];
  outputs[
    (YW+YWP)*(YH+YHP)*out_feature +
    (YW+YWP)*row + (YHPT*(YW+YWP)) +
    YWPT + col
  ] = cumulative > 0 ? cumulative : 0;
}
```

Source C.1: Padding-enabled baseline convolution kernel

Appendix D

Im2col variants

```
__kernel void im2col(
  __global const float* inputs,
  __global float* outputs)
{
  size_t Yx = get_global_id(0);
  size_t Yy = get_global_id(1);
  size_t Xz = get_global_id(2);
  for(size_t Fy = 0; Fy < FH; Fy++){
    for(size_t Fx = 0; Fx < FW; Fx++){
      size_t Ix = Yy * YW + Yx;
      size_t Iy = Xz * FW * FH + Fy * FW + Fx;

      size_t z = XW * XH * Xz;
      size_t y = XW * (Yy * SH + Fy);
      size_t x = (Yx * SW + Fx);

      outputs[Iy*IW + Ix] = inputs[z + y + x];
    }
  }
}
```

Source D.1: Im2col mapping kernel