Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Deep Neural Network Algorithms on Graphics Processors for Embedded Systems

David Cronin
13319942

10 May 2018

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Magister in Arte Ingeniaria at
The College of the Holy and Undivided
Trinity of Queen Elizabeth near Dublin
with the supervision of
Prof. David Gregg and Dr. Andrew Anderson

Submitted to the University of Dublin, Trinity College, May, 2018

Declaration

I, David Cronin, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: _____ Date: _____

Abstract

Deep Neural Network Algorithms on Graphics Processors for Embedded Systems
by David Cronin

Deep neural networks are becoming an increasingly popular method for tackling a range of problems, particularly those of recognition. A lot of focus is given to improving improving their performance by utilising graphics processors to perform key computations. However this focus is primarily on using computers with very high performance components. This thesis tackles the idea of performing these tasks on low powered embedded devices, often made with components that have comparatively very low performance. As such this thesis describes the implementation of an algorithm that performs a key computation associated with deep neural networks, specifically matrix multiplication.

The algorithm described can target graphics processors using OpenGL on a large range of devices, scaling from small low powered embedded devices with ARM architectures and integrated graphics processors to large high powered desktop workstations with Intel architectures and discrete graphics processors. The implementation is also portable and cross platform, supporting both macOS and Linux. This is achieved by repurposing OpenGL functionality, originally intended for graphics processing, to perform more general purpose computations.

This thesis demonstrates that it is possible to achieve improved execution times when performing matrix multiplication by efficiently using using OpenGL to target the graphics processor. This thesis also demonstrates as a proof on concept that this algorithm can compile and run on a small low powered embedded device for more limited computations. Further work is also described that could be done to achieve improved results. By being able to perform the work of deep neural networks on these embedded devices a whole range of new applications could become possible, as these devices would be more capable of interpreting their environments using image and audio processing.

Acknowledgements

I would like to thank my project supervisors, David Gregg and Andrew Anderson, for their advice and encouragement for the duration of this project. My thanks to Andrew in particular for helping me to integrate my work into the triNNity library and for reliably responding to my emails with helpful pointers and suggestions, especially when they were at the weekend.

Contents

List of Figures

List of Tables

List of Code Samples

Nomenclature

ab-ik	Matrices a and b are untransposed. Input order is image, kernel
abt-ik	Matrix b is transposed. Input order is image, kernel
abt-ki	Matrix b is transposed. Input order is kernel, image
AlexNet	A Convolutional Neural Network developed in 2012 by Alex Krizhevsky
ANN	Artificial Neural Network
API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
C / C++	A general-purpose programming languages
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DNN	Deep Neural Network
EGL	Embedded Systems Graphics Library
Embedded System/Device	A small constrained device designed for low power consumption
Execution Time	Number of elapsed CPU clock cycles
GEMM	GEneral Matrix Multiplication
GLEW	OpenGL Extension Wrangler
GLFW	Graphics Library FrameWork
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
I/O	Input / Output stream
im2col	Arranging patches of an image as rows of a matrix
im2row	Arranging patches of an image as columns of a matrix
Library	A collection of resources used by computer programmes
MCMK	Multi-Channel Multi-Kernel convolution
OpenCL	Open Computing Language, an API for performing general computations on the GPU
OpenGL	Open Graphics Library, an API for 2D and 3D graphics rendering utilising the GPU to provide hardware acceleration
OpenGL ES	An adaptation of OpenGL for Embedded Systems
SIMD	Single Instruction Multiple Data stream
triNNity	A library containing various implementations of techniques for computing neural networks
triNNity-benchmarks	A library for benchmarking the performance of the implementations in the triNNity library
VBO	Vertex Buffer Object

1 Introduction

1.1 Research Questions

Is it possible to utilise the GPU (Graphics Processing Unit) found on a range of low power embedded devices to perform computations associated with DNNs (Deep Neural Networks) using OpenGL? Is it additionally possible that by performing this computation on the GPU with OpenGL that the time taken to perform this computation is less than an equivalent implementation that only uses the CPU (Central Processing Unit)?

1.2 Objectives

- The aim of the research is to develop an efficient implementation of a key DNN layer. The layer that was chosen was the convolution layer.
- The implementation should be written in the programming language C++, specifically version 14.
- The implementation will compile and run on both macOS and Linux operating systems.
- The implementation will perform at least some of the work of the convolution on the GPU.
- The implementation will interface with the GPU using the libraries OpenGL and OpenGL ES, which will be described later in this thesis.
- The implementation will be tested on multiple devices, ranging from a desktop or laptop computer with an Intel CPU to a small low powered embedded device with an ARM system on a chip. All devices must have a GPU with an OpenGL or OpenGL ES driver.
- Ideally the performance of the OpenGL implementation using the GPU should be superior to the equivalent implementation that only utilises the CPU in terms of

execution time.

- This performance improvement would ideally be the case across the range of devices described.
- These implementations will be integrated into the ‘triNNity’ library developed by the 'software-tools-group' (1).

1.3 Purpose of the Research

DNNs are becoming an increasingly popular method for tackling a range of problems, particularly problems in the field of recognition. They have increased the accuracy of prediction and recognition problems significantly, such as image classification or audio recognition. Performing this sort of computation on an embedded system creates a whole range of possibilities where small low powered devices can better interpret their environment. In a scenario where many of these devices are connected to a network it opens the possibility that rather than transmitting all recording input back to some main server for processing, some of this processing could be performed on device at the edge of the network. This would improve latency, which may be desirable if the devices purpose involves human interaction, and arguable be more ethical, if the device is used in a sensitive context where the data can remain on device rather than being transmitted to a server. This would also reduce overall traffic on the network by only transmitting the results of processing rather than all of the data recorded.

To illustrate these benefits take the example of virtual voice assistants, which are becoming more and more popular. A recently published article by the Siri team at Apple describes how the use of DNNs on the iPhone can be used to recognise an utterance of the trigger phrase "Hey Siri" (2). This trigger phrase will then summon the virtual voice assistant, Siri, to listen to the following sentence. This following sentence is then transmitted to a main server to process the request. While it is infeasible to perform the entire request on device, the use of DNNs on the device have enabled the voice assistant to be triggered hands free. If the processing could not be performed on device then the feature would require the phone to constantly stream the microphones input to a server to wait for the trigger phrase. Not only would this have poor latency with a round trip to a server, but it would also consume a lot of bandwidth on the network. Processing the input from so many microphones remotely would likely be an impossible task to perform at scale, whereas offloading the simpler constant trigger processing to device while handling the more complex infrequent requests on the server would be more scalable. The ethics of such a feature would be questionable as transmitting the microphone input constantly over the network, even when the device is not being used, would likely be considered a violation of user privacy.

Another example of the use of DNNs on embedded devices again comes from a feature of the iPhone. DNNs have been used for facial recognition on the iPhone for nearly two years in the Photos App, while in November of 2017 the launch of Face-ID (3) allowed for the device to authenticate users with the front facing camera apparatus and DNNs. Again a feature of this description needs to perform the computation on device, since it will not always be connected to a network and the response time of a network request can be too poor. Meanwhile performing this type of processing with DNNs has yielded promising results, enabling this sort of technology on embedded devices. As such there is clearly a need for computing DNNs on embedded devices. Improvements in performance of these DNNs in terms of execution time would enable more possibilities and more processing on these devices.

There are challenges however. An excerpt taken from the brief for this project: "However, DNNs require extraordinarily large amounts of computation to operate, which stretches the limits of the computation capacity and memory of embedded devices... GPUs provide lots of processing capacity, but require carefully written code to achieve good performance... The designers of embedded systems with GPUs often discourage developers from using the GPUs for general-purpose computation. As a result, many embedded systems with GPUs have limited support for general-purpose GPU (or GPGPU) programming languages such as OpenCL, or disable OpenCL entirely. However, many more embedded boards support OpenGL, which provides GPGPU programming facilities, albeit with fewer features than OpenCL" (4). Embedded boards with GPUs will generally always provide OpenGL support to enable graphics processing, however support for OpenCL, which would be more suitable, is more unreliable. By using OpenGL it is hoped that a wider range of devices could run the developed implementation as there is more support for OpenGL on these types of devices. Thus this choice will enable this implementation to target devices ranging from desktop workstations to small embedded devices. To use OpenGL for general purpose computation on an embedded system it is necessary to target devices with support for OpenGL ES (OpenGL for Embedded Systems) version 3.0 and higher, which provides enough functionality for the task (5). Using OpenGL to perform this type of work is a novel idea, and very little to no research exists as to how this implementation should be approached. Therefore there is very little discussion of existing implementations of convolutional layers that target the GPU, as most of these use OpenCL which is not available. Rather a general approach to convolution implementations is provided and the convolution implementation described was developed from scratch for this thesis.

Therefore the goal of this project is to build a set of primitive functions in OpenGL to implement a key DNN layer, convolution. An iterative approach was taken for the development of this implementation, interfacing with the GPU through OpenGL and making improvements guided by GPU profiling and the execution time of the algorithm. Different

approaches were taken throughout development on how the work was done on the GPU or what aspects of the algorithm were performed on the GPU. The final implementation is discussed in this thesis. This implementation, integrated into the 'software-tools-group/triNNity' repository (1) will be run by the triNNity-benchmarks suite (6) to produce benchmarks of its performance compared with equivalent CPU implementation on a range of devices. This will be the method of collecting results, which will be be analysed to evaluate the implementations developed. With these benchmarks it will also be possible to validate the accuracy of the implementations along with measuring the time they take to complete.

1.4 Layout of the thesis

This thesis tackles the topic as follows.

The objectives, application and parameters of the problem have been established. Next the background to the problem will be explored, with a description of how machine learning works, building up form a basic component, perceptrons, to deep neural networks. This will be accompanied by a description of how these networks are implemented and some background information on GPU programming.

Following on from this will be an in depth breakdown of the implementation put forward by this thesis. Detailing how it takes advantage of OpenGL functionality, originally intended for graphics processing, and uses it for more general computation instead.

Lastly an evaluation of the implementation is laid out, detailing how the implementation is tested, what the results of this testing were for a range of convolutions on different devices, what the results mean, short comings of the implementation, and how the implementation could be improved with further work.

There will then be a conclusion. The appendix will contain the full code developed for the implementation for closer examination.

2 Background

This section will cover the important concepts and background informations needed to understand the field and the algorithm implementation covered in this thesis. First will be an overview of machine learning and some of its basic techniques, followed by a description of neural networks and how they function. This will be wrapped up with an introduction to how GPUs are designed, and what APIs will be developed on to utilise them.

2.1 Machine Learning

The aim of machine learning is to create computer systems that are able to perform a task without being specifically designed for that task (7). A traditional computer system, or expert system, uses a rule base and heuristics to process some input and produce an output. These rule bases and heuristics need to be programmed and tailored for a specific task, and thus cannot generally be reapplied to other tasks without some changes. While machine learning systems still contain these rule bases and heuristics, they are not explicitly programmed but are inferred, or 'learned', from a training dataset of correct outputs for a given set of inputs. As such these systems are essentially being programmed by the data they are trained on, rather than explicitly by a human programmer. Where a traditional system would handle a new edge case scenario by having a new rule being explicitly programmed and added to its rule base, a machine learning system would handle a new edge case by having an example of the edge case added to the training dataset with the correct output. The machine learning system would need to be retrained to incorporate, or 'learn', this new information, but the human programmer would not be required to rewrite any rules of the system, rather the system would rewrite its own rules. This would be a significant advantage for certain problems since maintaining a large programmed rule base can become unwieldy as more edge cases are added. These rules need to also be determined by the programmer, which can be challenging and time consuming.

The following example illustrates this difference between a programmed system and a trained system. Suppose it was desirable for a system to be built that would identify the

colour of cars in images. Two approaches could be taken. The first one where a rule based system is developed that takes the image as an input, examines the pixel values of the vehicle and using a heuristic with some hard coded colour ranges, outputs the colour of the vehicle. The second approach would be to create a dataset of car images of different colours, label each image with the correct colour, and train a machine learning system on the dataset. Now when both systems are presented with new images of cars they are able to correctly identify the colours of the cars. Suppose that in the development of both systems it was forgotten to account for pink cars. To now account for it, the first system would need a new rule added to it, or the existing colour ranges would need to be modified to include a range for pink. A problem is presented here for explicitly programmed systems, it is difficult to determine exactly where one colour transitions to another. In the second system, the training dataset needs to be expanded to include images of pink cars and the system retrained. The exact distinction between two colours is not of interest as the training will determine this boundary based on the labelled training data. The difference between the two systems is that the first requires careful maintenance of rules, whereas the second requires careful maintenance of a training dataset.

Suppose now that it was desirable for both systems to be expanded to determine not only colour, but also the brand of the car. This present serious challenge to the development of the first system, since the programmer needs to develop rules that allow the system to correctly identify brands based on the car badge. It would be challenging to come up with a set of rules that would determine the car brand based on the size and shape of the badge, and the complexity of the problem has grown significantly. To contrast, in the second system the dataset needs to be modified to include examples of labelled cars from a range of brands. How the brand is matched to the car is determined by the system in the training process. As long as the training dataset is large enough to incorporate enough scenarios, the machine learning system should adapt and be able to meet the new specifications of the task.

With this example the advantage of a machine learning system should be clear. Rather than maintaining a system with a rule base for each task, the rules of which can be extremely challenging to determine, the programmer maintains one system and has a training dataset for each task, where the labels of the data should be relatively easy, although time consuming, to determine. This machine learning system hides the complexity of the task from the programmer by determining the rule base itself with training, and can be applied to a range of problems without being redesigned or specifically programmed.

With only one system or code base, it can then be a focus for performance optimisation. By optimising this one system the benefits are utilised for every task that the system handles. Whereas with a rule based system only one task can be optimised at a time. Thus improving a machine learning system provides significantly better returns than optimising one rule based system. This is key to understand for the purpose of this thesis, as the goal is to

implement and improve a key component of machine learning systems, which is beneficial to a range of tasks that machine learning systems are applied to.

2.2 Artificial Neural Networks

Up until this point the focus has been on the difference between a machine learning system and a traditional rule based system. This section will discuss how a machine learning system is implemented to illustrate how the system can 'learn' as opposed to being task specific.

Artificial neural networks (ANNs) are computer systems inspired by biological neural networks in animals brain. They were an attempt to mimic how the brain works to create computer programmes that can perform tasks similar to how the brain would and have since been successfully developed for uses in fields such as computer vision and speech recognition. ANNs are composed of nodes, that resemble a neuron cell, which receive an input, perform a function on that input, and produce an output. A network can be composed of a number of these nodes connected such that the output from one node can be the input for another node. (8)

To examine what is meant by a node here it is appropriate to look at Rosenblatts perceptron (9), see figure 2.1, which was an early algorithm developed to simulate a neuron.

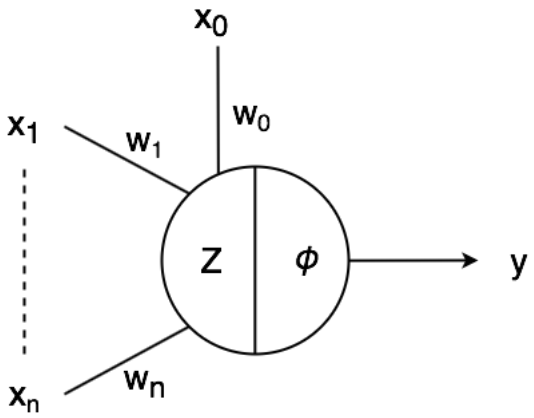

Figure 2.1: Rosenblatts Perceptron

The inputs to the perceptron are labelled x_0 through x_n with x_0 being a constant or bias. Each input has an associated weight, denoted w_0 through w_n, which can be adjusted depending on the task. z represents the sum of the inputs multiplied by their weights, as shown in Equation (1).

$$z = \sum_{i=0}^{n} x_i.w_i \tag{1}$$

Φ represents the activation function or linear threshold unit, which determines whether the output is a 0 (-1), or a 1 depending on whether the value of z is negative or positive.

Perceptrons are simple yet capable. Each perceptron is a classifier. It allows for a single linear decision boundary to be determined in a space. Thus with only one perceptron it is possible to model a logical AND operation or a logical OR operation by adjusting the weights correctly (9). Combining multiple perceptrons into a layer, as seen in figure 2.2, allows for the creation of a multi-output classifier.

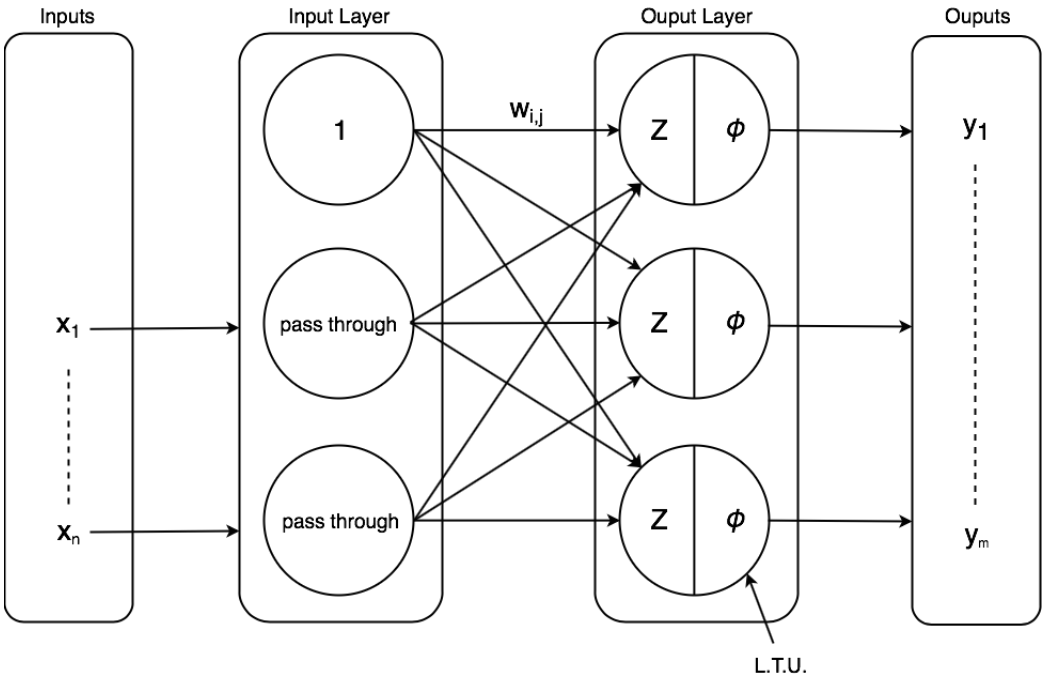

Figure 2.2: Single Layer Perceptron Network

The weights used for these perceptrons are the aspect that need to be adjusted or trained to suit a specific task, such as a logical AND operation or a logical OR operation. Before training these weights have some initial and are iteratively adjusted by inputting data from the training set modifying them according to Equation (2).

$$w_{i,j} = w_{i,j} + n(y_j - \hat{y}_j)x_i \tag{2}$$

where $w_{i,j}$ is the current weight, n is the learning rate y_j is the output $\hat{y}_j$ is the expected output, and x_i is the input.

Training involves running over the data in the training set, probably multiple times, and adjusting the weights until the network converges, that is all elements in the training set get the correct output and the weights no longer need adjusting. To illustrate this example suppose it is desirable to train a single perceptron to perform a logical AND operation. A training set would be composed of the truth table, table 2.1. The training process would use each row of the table as input, calculate the output, compare with the correct output, and if

x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1

Table 2.1: Truth Table for Logical AND Operation

they were not equal adjust the weights according to the previous formula. This would be repeated until all inputs had the expected output, and thus the network had converged, or until some maximum number of training iterations had occurred.

It is also possible to have multiple layers combines into a single network (9). The output from one layer of perceptrons can become the input for another layer, as seen in figure 2.3. A layer that is not connected directly to either the inputs or the outputs is a hidden layer. A network that contains 2 or more hidden layers is generally considered to be a Deep Neural Network (DNN). Now with one additional, hidden, layer it is possible to determine non-linear decision boundaries. This allows a network to be trained to perform a logical XOR operation, or any continuous mathematical model given enough perceptrons. With two hidden layers it is possible to compute any decision boundary, and there is theoretically no need to use more than two hidden layers, performance aside (10).

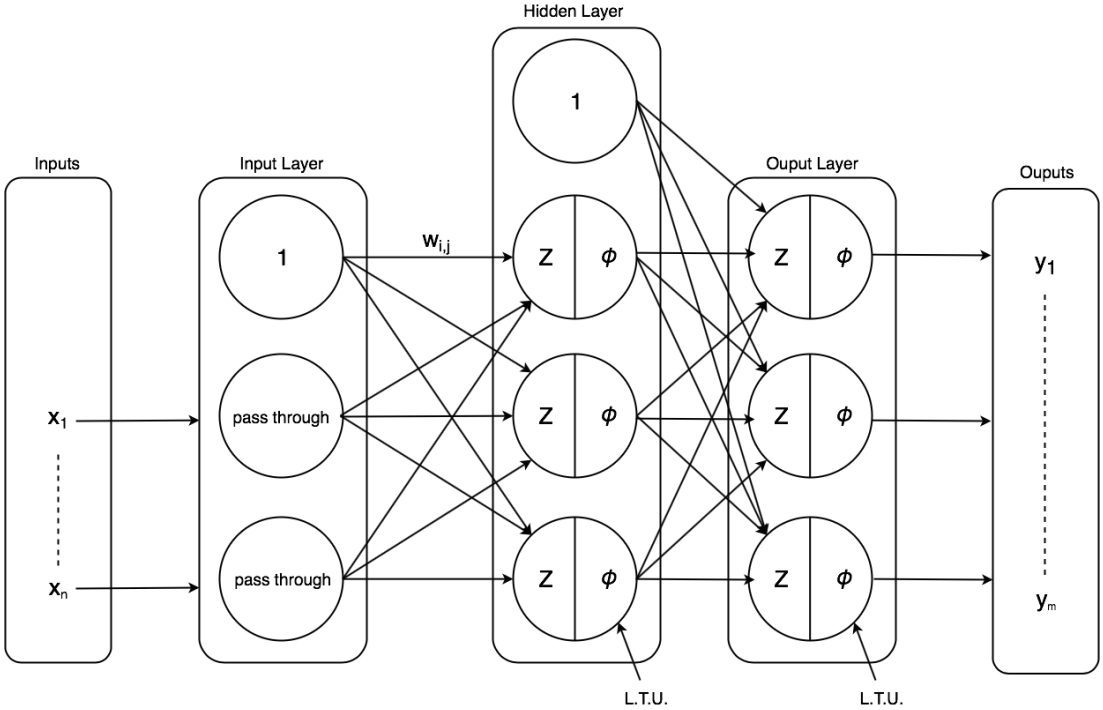

Figure 2.3: Multi-Layer Perceptron Network

The weights associated with the connections between neurons in a DNN are also adjusted through a training process called backpropagation, which is a more advanced version of the weight adjustment formula previously discussed.

2.3 Convolutional Neural Networks

This section will discuss what a Convolutional Neural Network is, and how several of its different layers, based on perceptron layers, are implemented. Convolutional Neural Networks (CNNs) or Convolutional DNNs were popularised by the success experienced in 2012 by Alex Krizhevsky in the development of AlexNet. By achieving good results when classifying images in the ImageNet dataset, they showed that a "deep convolutional neural network is capable of achieving record-breaking results on a highly challenging dataset using purely supervised learning" (11). This created increased interest in using CNNs for tasks such as image processing.

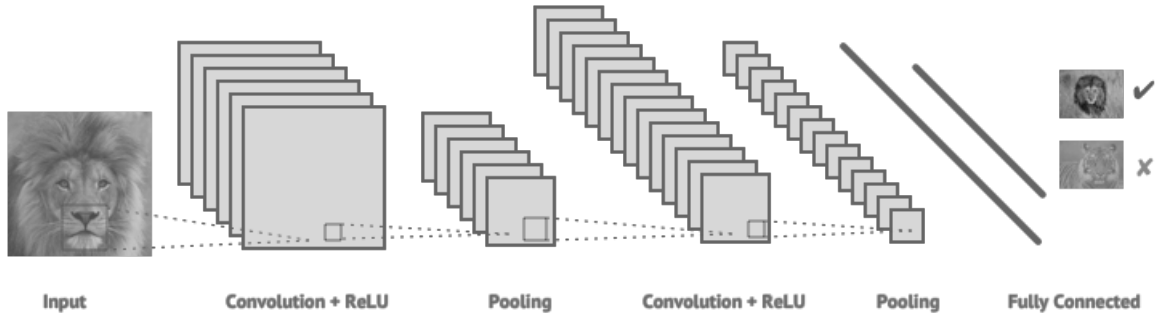

Figure 2.4: An example of a Convolutional Neural Network[1]

CNNs are combinations of several types of layers. Examples of these different types of layers include convolution layers, pooling layers, and fully connected layers (12). These layers consist on perceptrons, as described before, however they differ in the ways they are interconnected or the function that is performed on their inputs. Many different arrangements, or architectures, exist for how these layers should be combined into a network. An example of an architecture can be seen in figure 2.4. An image is used as input for the network, where it is first passed through a convolution layer, followed by a pooling layer, etc. Different architectures are more successful than others, and the previously mentioned AlexNet is an example of one of the first successful architectures developed. However the advantages and disadvantages of different architectures is not of interest to this thesis. The core topic of this thesis is the implementation of specific algorithms used in neural networks, and these algorithms are key implementation details of the different layers mentioned. The layer that was chosen for implementation for this thesis was the convolution layer. Therefore the remainder of this section will be a brief description of the fully connected layer and the pooling layer, and then a more thorough description of the convolution layer including a analogous description of what it is attempting to do.

[1]Image from `https://shafeentejani.github.io/2016-12-20/convolutional-neural-nets/`

2.3.1 Fully Connected Layer

A fully connected layer is the same as what has already been described in the section 2.2 when discussing multi layer perceptrons. It is a layer where all outputs from the previous layer are connected to every perceptron or neuron in the layer (12). Each connection has a weight associated with it, and a sum of the inputs multiplied by the weights is calculated. As such it is the same as the hidden layer described in the multi-layer perceptron network. An example is shown in figure 2.5.

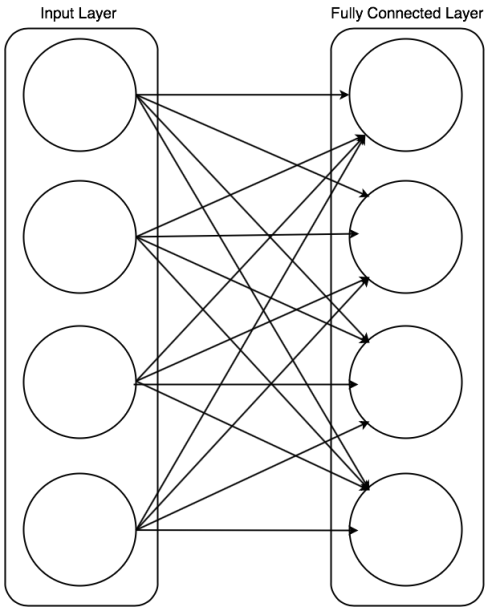

Figure 2.5: A diagram of an Input Layer and a Fully Connected Layer

2.3.2 Pooling Layer

A pooling layer is a layer where each neuron is connected to only a subset of the outputs from the previous layer. There are no weights associated with these inputs. Instead the objective of this layer is to downsize the number of outputs. As such the function performed by the neuron is usually to compute the max, or the mean of its inputs (12). An example can be seen in figure 2.6 where the inputs have been split into four sections and the max of each section is computed. If the input to the layer were an image the effect might be to split the image into squares and take the average or maximum pixel value of that square, thus reducing the image size by a factor of the size of the squares.

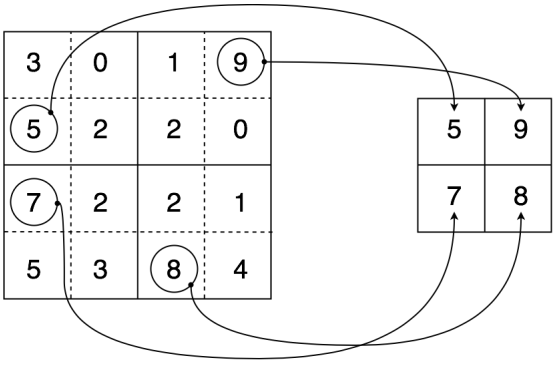

Figure 2.6: An illustration of a Max Pooling Layer

2.3.3 Convolution Layer

A convolution layer is also a layer where each neuron is connected to only a subset of the outputs from the previous layer. Unlike the pooling layer however, these subsets of inputs overlap such that the same input will be handled by multiple neurons. This overlapping can be seen in figure 2.7. Much like the fully connected layer, this layer also has weights associated with each input. The inputs are multiplied by these weights and summed produce the output (12). No activation function is performed to produce the output so the output is not only a 0 or a 1, however an activation layer may follow for this purpose.

The weights that are applied to the overlapping subsections of the input are collectively called the kernel. The kernel is convolved with an input matrix, say an image, such that the centre of the kernel is aligned with each pixel in the image. The surrounding pixel values are multiplied by the corresponding kernel weights and the results are summed. This is performed for every pixel in the image (12). An example can be seen in figure 2.7 where a three by three kernel is being applied to a matrix. To apply the kernel to every point in the input, zero padding is added to the edge of the input such that the kernel can extend over the edge of the input matrix. Notice that this allows the output to be the same size as the input. Stride can also be applied so that the kernel is not applied to every point on the input matrix but only to every second or third point.

The idea behind these kernels is that they allow for pattern recognition in the input. Suppose that the kernels shown in figure 2.8 were convolved with an image. Parts of the image that match the patterns in the kernel would have higher corresponding values in the output. For example if there was an edge in an image, similar to that in the third kernel, then the positive values on the left side and the negative values on the right side, when applied to the an edge in the image, would result in a larger result when applied to that edge. This would mean that the result of the third kernel being applied at that edge would be greater than if it was applied further to the right. This results in an output that has higher values corresponding to areas that matched the pattern in the kernel. This technique

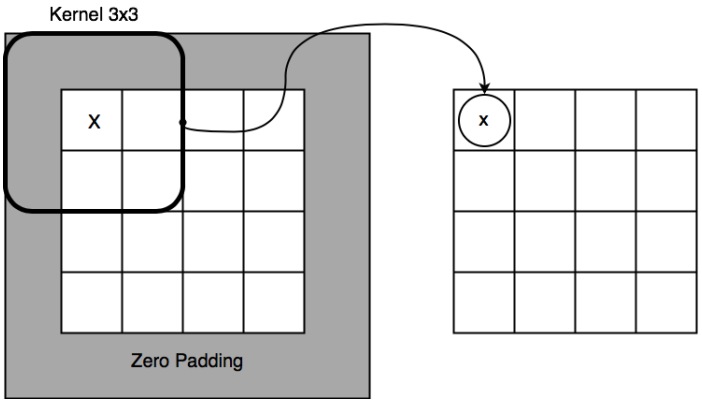

Figure 2.7: An illustration of a Convolution Layer

can be used to detect a range of patterns as well as edges, such as corners, spots or any variation of values in the kernel.

-0.8	-0.7	0.6
-0.7	0.6	1
0.6	1	1

1	1	-0.8
1	1	-0.7
-0.7	-0.8	-0.9

1	1	-1
1	1	-1
1	1	-1

Figure 2.8: Example kernels with values or weights for three different patterns

To illustrate how this works, figure 2.9 shows the results of a kernel with an edge pattern being convolved with a small input matrix. Notice how the high values in the output correspond to points of the input where the surrounding values had the same pattern as those in the kernel used. This allows the pattern in the kernel to be detected in the input.

Input

10	10	2
10	10	2
10	10	2

Kernel

1	1	-1
1	1	-1
1	1	-1

Output

0	36	24
0	54	36
0	36	24

Figure 2.9: Example of a kernel applied to an input. Note how the resulting values at the edge are larger.

This technique is useful in image processing as it enables the detection of features in images, which can then be used to classify the image. For example arrangements of certain patterns may be useful for classifying letters, faces, or other objects.

2.4 GEMM

Up until this point the discussion has been focused on the abstract implementation of neural networks in terms of the layout of neurons and what functions they perform. GEMM stands for GEneral Matrix to Matrix Multiplication. It is a function that multiplies two matrices together and returns the resulting matrix. This section will discuss some linear algebra used in GEMM and how it is used in the implementation of two layers of a convolutional neural network. Specifically these layers are the fully connected layer and the convolutional layer. This section is based on a blog post "Why GEMM is at the heart of deep learning" by Pete Warden (13).

2.4.1 Dot Product

The dot product is a mathematical operation used in linear algebra to multiply two vectors and sum the result, as can be seen in Equation (3). It is the same calculation performed by Rosenblatts perceptron as seen in Equation (1), and is useful for illustrating matrix multiplication. It is also used as part of the GEMM implementation discussed later in this thesis.

$$X = \begin{pmatrix} x_0 & x_1 & x_2 \end{pmatrix} \quad A = \begin{pmatrix} a_0 & a_1 & a_2 \end{pmatrix}$$

$$y = X.A^{\mathsf{T}} = \begin{pmatrix} x_0 & x_1 & x_2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix}$$

$$y = \sum_{i=0}^{n} x_i.a_i = x_0.a_0 + x_1.a_1 + x_2.a_2 \tag{3}$$

2.4.2 Matrix Multiplication

Matrix multiplication can be seen illustrated in figure 2.10 where two matrices, A and B, are being multiplied to produce a matrix C. For the multiplication to be performed the number of columns in matrix A must equal the number of rows in matrix B. For every row in matrix A, the dot product is computed between this row and each column in matrix B. This is why the resulting matrix has the same number of rows as the number of rows in matrix A, and the same number of columns as the number of columns in matrix B.

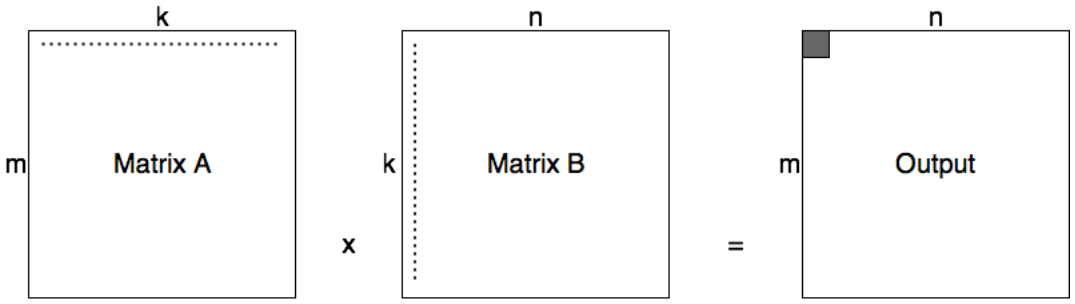

Figure 2.10: An illustration of Matrix Multiplication

2.4.3 Fully Connected Layer

The fully connected layer is implemented by way of a matrix multiplication (13), as seen in figure 2.11. The inputs to the layer are effectively a vector, or a 1 by k matrix where there are k inputs. This can bet multiplied by a k by n matrix, where k is the number of weights and n is the number of neurons. Thus the input is contained in the first matrix and the weights are contained in the second matrix. The result is then a 1 by n matrix with the output of each neuron. Remember each output the result of the dot product of the inputs and the weights for that neuron.

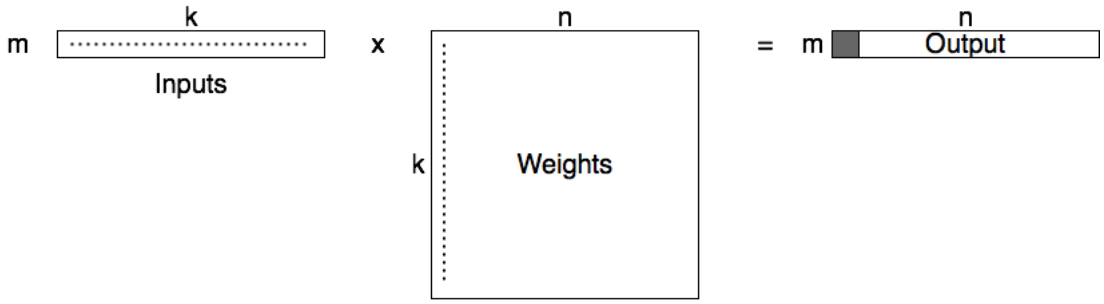

Figure 2.11: A Fully Connected Layer implemented with Matrix Multiplication

2.4.4 Convolution Layer

A convolution is more difficult to perform with a matrix multiplication. Since a different subsection of the inputs is used by each neuron and the same input can be used my multiple neurons. This means that some processing is required to rearrange the input into a matrix with a format suitable for matrix multiplication. A kernel is applied to a particular area or patch of the input matrix. The inputs inside this patch are then used by a single neuron. To arrange the inputs into an appropriate format, the input matrix is expanded into a patch

matrix (13), as seen in figure 2.12. A patch of the inputs, the same shape and size as the kernel, is extracted from the input matrix for every point where a kernel would be applied and becomes a row in the patch matrix. These patches will overlap, as described in section 2.3.3, as long as the stride is less than the kernel size so this patch matrix will be larger than the original input matrix. However the increased size is considered a worth while trade off as "the benefits from the very regular patterns of memory access outweigh the wasteful storage costs" (13). Figure 2.13 shows how this patch matrix is multiplied with a kernel matrix to

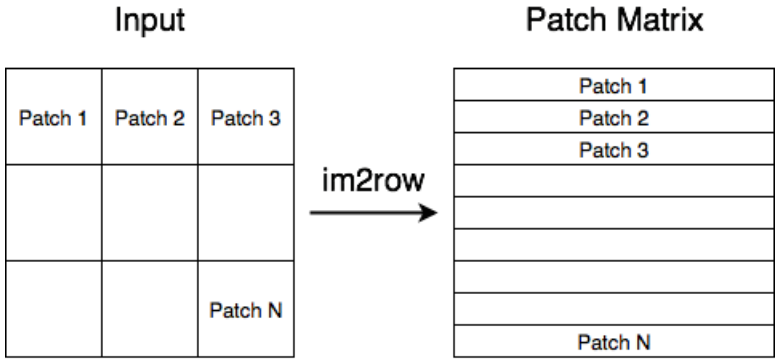

Figure 2.12: The creation of a Patch Matrix

produce the output. This output takes the form of a matrix where each element in a column is the results of a convolution for with that kernel at a certain point in the input. Each column would then contain the result of convolving the input with a particular kernel. Thus a column could be rearranged to have the same dimensions as the input, assuming no stride was used and zero padding was applied. As multiple kernels are usually applied in a convolution layer it is useful to be able to compute the results of that convolution layer in a single function.

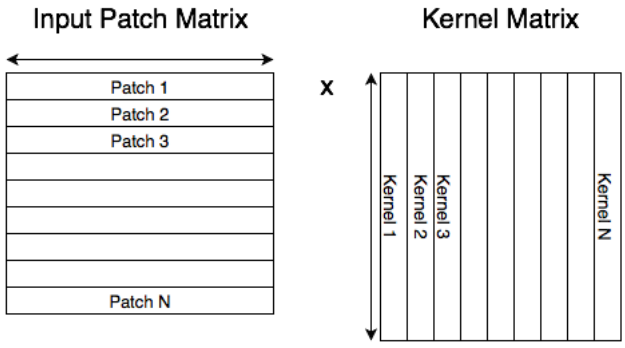

Figure 2.13: Patch Matrix and Kernel Matrix being multiplied to perform convolution

The GEMM function is implemented in the BLAS (Basic Linear Algebra Subprograms)

library (13) as previously said it is used to implement fully connected and convolution layers. The CPU version of AlexNet spends around 89% of its time in these layers, while the GPU version spends 96% of its time processing these layers (13). Thus the GEMM function is clearly a key component of neural network implementation and it a good target for optimisation since it is used so heavily.

2.4.5 Implementations

As discussed GEMM can be used to implement a MCMK (Multi Channel Multi Kernel) convolution. Multiple channels are handled by having the information for each channel together for a single point and can used to represent the different colour channels in an image for example. Multiple kernels are stacked as rows or columns in a kernel matrix.

The information presented as input needs to be rearranged for it to be processed as a matrix multiplication. One technique has already been discussed, where patches of the input are unrolled into rows of the patch matrix. Another implementation would involve the patches becoming columns of the patch matrix. These methods are known as *im2row* and *im2col* (13) and can be seen in figure 2.14

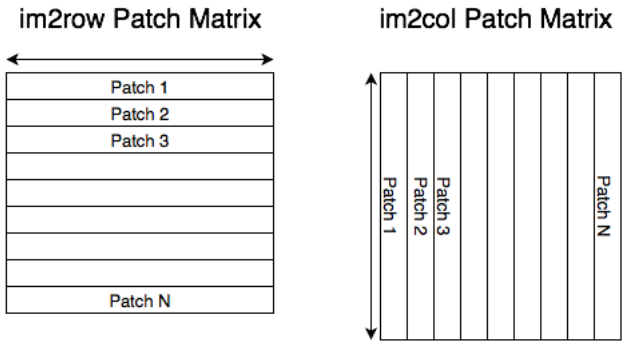

Figure 2.14: im2row and im2col Patch Matrices

As stated before, these methods involve expansion of memory size as patches overlapping result in values being repeated across multiple patches.

A more memory efficient approach is outlined in (**?**). Here the *kn2row* method is described where the kernel is rearranged into rows such that there are *k x k* rows. This can then be multiplied by an input matrix such that every kernel value is multiplied by every input value without memory size increasing. Sections of the output can then be summed or the output can have a *post-pass shift add* applied to get the result of the convolution. This could be very beneficial for performing convolutions on embedded devices which are memory constrained.

2.5 Graphical Processing Unit

A GPU, or graphical processing unit, is a component of modern computing devices that is specialised for processing blocks of memory in order to accelerate the creation of images intended for displaying to the user. GPUs are used in a range of devices from small embedded devices to desktop workstation. This section will briefly outline their architecture and why it is beneficial to the task of this thesis, and also the method used to implement an algorithm on a GPU.

2.5.1 Architecture

A GPU is based on a SIMD (Single Instruction Multiple Data stream) architecture. This is a form of parallelism where multiple processing units, or cores, are performing the same instruction sequence on different blocks of memory at the same time (14, 15). This is illustrated in figure 2.15.

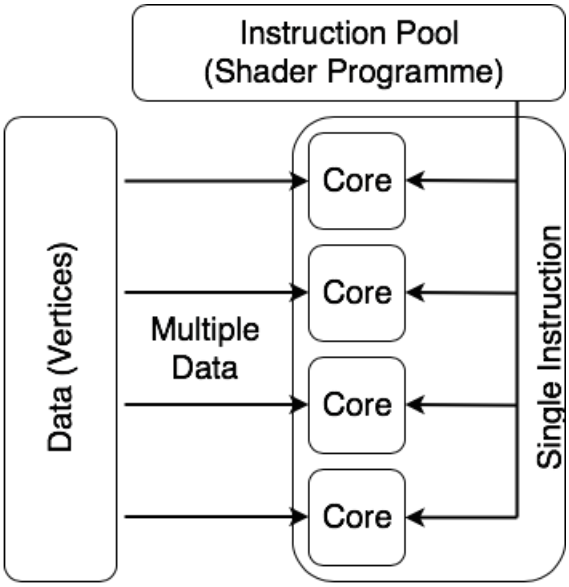

Figure 2.15: Single Instruction Multiple Data Stream

The instruction sequence being performed must be synchronised between the cores such that the same instruction is being performed on all cores at the same time. If there is control flow present in the instruction sequence causing some cores to jump over an instruction then these cores will stall rather than move forward to their next instruction, thus ensuring that all cores remain synchronised (15). While modern GPUs batch execution threads to minimise stalling (15), control flow is still expensive. The number of cores present is generally much higher than on a CPU since it is much simpler to manage more cores when the instructions are the same, and the capabilities or instructions sets for these cores are

much simpler. With this architecture the ideal task for a GPU is one where a large number of threads with the same instruction sequence is working on separate batches of data. This allows the larger number of cores to be utilised on the GPU for a highly parallelised algorithm (16). This is highly suitable for matrix multiplication which involves the same instructions being performed on different data.

2.5.2 OpenGL

OpenGL, or Open Graphics Library, is a API, or application programming interface, for 2D and 3D graphics rendering utilising the GPU to provide hardware acceleration (17). Wile OpenCL (Open Computing Language) would provide more suitable functionality, since the task is a computational one rather than a graphics one, OpenGL was chosen since it has wider support amongst embedded devices that it is aimed to target (4). This section will provide a brief introduction to how the OpenGL Graphics Pipeline works, while the implementation section will discuss specific OpenGL concepts and functions that were utilised for this thesis.

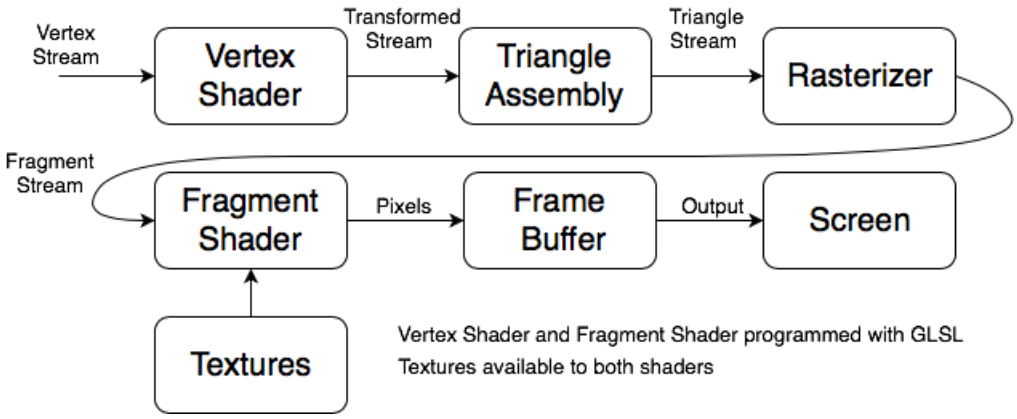

Figure 2.16: Programmable Graphics Pipeline

The programmable graphics pipeline (18) can be seen in figure 2.16. The programmable stages of the pipeline are the vertex and fragment shaders. For these stages a process using OpenGL can compile and link shader programmes at runtime which will be loaded on the GPU. These programmes will then be executed by cores on the GPU as needed (19). The first stage of the pipeline is one of these shader stages, the vertex shader. It receives a stream of vertices as input. These vertices would be those of some shape or object to be drawn to the screen. The vertex shader would transform these vertices, perhaps rotating or scaling them as instructed in the programmed compiled for the shader. These transformed vertices would then be passed on to be formed into shapes, generally triangles, to form the surfaces of an object. These triangles are then rasterized, meaning they are formed into a grid of pixels that would represent them from a particular viewing angle. These pixels, or

fragments, are then passed to the fragment shader which would colour in the pixels appropriately, often using texture or images that can mapped onto the object. These pixels are then pushed into a frame buffer and finally drawn onto the screen.

2.5.3 OpenGL ES

OpenGL ES is a stripped down, or streamlined version of OpenGL that is specifically for embedded systems (5). It has wide support on a range of embedded devices such mobile phones, tablets, or low power development boards. It contains a subset of the features available in the full OpenGL library for desktop sized devices, while also removing a lot of legacy restrictions or constraints. Not all functions available in OpenGL are available in OpenGL ES, either because they have not been added to the specification yet or they are no longer relevant or suitable for a modern graphics API. As such is is important to ensure that the functions utilised by this thesis are available to both versions of the OpenGL library.

3 Implementation

This sections will cover the details of how a GEMM (General Matrix Multiplication) function was implemented for this thesis. This implementation utilising the GPU with OpenGL on a desktop computer and OpenGL ES on an embedded device. The same implementation supports both environments by using a macro definition at compile time to target either OpenGL or OpenGL ES. As these two libraries have slight difference in capabilities and functionality, this macro allows for slightly different code to be compiled in either case to handle these differences. The implementation also supports both MacOS and Linux.

The GEMM function implemented accepts two pointers as input, which allow access to two blocks of memory containing the two matrices, A and B, to be multiplied. One of these matrices will be the input patch matrix and the other will be the kernel matrix, however these details are not relevant to the function. The details of the convolution are abstracted away from the implementation such that the function will just accept two matrices and multiply them. Relevant details, such as the dimensions of the matrices and what transposition the matrices are in, are also provided as parameters of the function. A pointer is also provided to contain the output.

The function was integrated into the 'software-tools-group/triNNity' repository [ref] as a handler for a call to a generic GEMM function. This library provides several functions that perform 'im2row' convolutions by creating a patch matrix and calling a GEMM function to multiply the patch matrix with a kernel matrix. Thus forming the patch matrix was handled by functionality in the repository, leaving this implementation to only handle matrix multiplication. This allows the GEMM implementation described here to be interchanged with an equivalent CPU based GEMM function to compare performance.

This section contains code samples to aid in explaining how the GEMM implementation works. These code samples may differ from the actual implementation for clarity of illustration, and are intended to aid in understanding the full unaltered version of the code contained in the Appendix.

3.1 Setup

In order to call OpenGL function to perform computations on the GPU it is necessary to first perform some initial setup.

Firstly the correct header files need to be included to provide function definitions, type definitions, and other macros representing values that might be needed to pass to functions for certain functionality. On platforms with OpenGL, this is handled by the GLEW (OpenGL Extension Wrangler) library (20), which is a cross platform library that provides OpenGL extensions, or function definitions, that are supported by the platform. GLEW does not support OpenGL ES however and so for these situations the OpenGL ES version header must be included specifically with appropriate macro defined to include the right subset of function definitions.

It is also required that an OpenGL context is established for the process to use OpenGL. An OpenGL context represents all of the OpenGL state and objects required for an instance OpenGL to function, such as details of the display window. As such to make calls to the OpenGL library it is first necessary to create an OpenGL context. As OpenGL is intended for drawing pixels to a screen it is necessary for an OpenGL context to have access to a display and a window. These are not necessary, nor desirable, for performing computations that do not result in graphical outputs. In fact many embedded systems where the implementation would be deployed may not be connected to a display. As such it is necessary to persuade the OpenGL context that a display is present, even if it is absent. This issue can be resolved by setting the appropriate environment variable in the process such that the DISPLAY variable is not NULL. Since the implementation will never try to push anything to the screen the lack of a connected display will not matter. It is also necessary to create a window in the OpenGL context, but the window should be invisible or hidden.

To create the OpenGL context and define invisible windows two solutions were employed. For platforms with OpenGL support the cross platform library GLFW (Graphics Library FrameWork) (21) was used to create an OpenGL context with a hidden window. An illustrative example of how this is achieved is provided in listing 3.1.

Listing 3.1: Outline of GLFW Context and Window Creation

```
glfwInit();
// indicate the window should be invisible
glfwWindowHint(GLFW_VISIBLE, 0);
// Create a window with a GL context
GLFWwindow* window = glfwCreateWindow(640, 480, NULL, NULL);
// Make the context of that window current
glfwMakeContextCurrent(window);
```

For platforms with OpenGL ES support, EGL (Embedded Systems Graphics Library) (22) was used to created contexts and windows for making OpenGL ES calls. This library worked well on a headless device (23), and was called similarly to GLFW.

Since two different libraries were used this meant that the at compile time the correct libraries would need to be linked depending on whether it was targeting OpenGL or OpenGL ES. The creation of contexts and windows was performed once initially. Thus this initialisation process does not factor into the execution time of a GEMM function. This is due to how in the operation of neural networks this function would likely be called a large number of times. Thus the cost of initialisation, in terms of execution time, would be spread out over every call to the function. Therefore the more GEMM calls that are made, the smaller this initialisation process is relative to the overall execution time.

It is also necessary to perform some cleanup after the last GEMM call is made to terminate GLFW or EGL and remove the OpenGL context for the process.

3.2 Transferring Data to the GPU

There are three techniques that were used to load data onto the GPU for computation. These different techniques have an original purpose designed for graphics processing, and have constraints and properties accordingly. These functions can be utilised for computation by restructuring the task to work within these constraints and even take advantage of some of their properties.

The first of these techniques is called a Vertex Buffer, or Vertex Buffer Object (VBO) (24). This is a buffer off memory on the GPU that is used to store an array of vertices used for drawing shapes or objects to the screen. For example if it was desirable to draw a triangle to the screen then the Vertex Buffer would be loaded with three coordinates, each representing a vertex of the triangle, in sequence. More complex shapes could be draw with more coordinates, and 3D objects drawn using coordinates with three dimensions. These vertices can be transformed using a Vertex Shader, outlined in the background to OpenGL (section 2.6.2). This implementation employs a Vertex Buffer to store the contents of the first matrix, mat A, for access by the GPU. This allows the GPU to run over the contents of the first matrix as if they were coordinates. This buffer needs to be generated and bound to the current OpenGL context for drawing. The reason it needs to be bound is that a programme may have multiple objects to draw and these can be stored in separate Vertex Buffers, allowing different objects to be drawn separately. Thus binding is a mechanism for choosing which Vertex Buffer to draw at a given time. An example of how a VBO is created, bound, and filled with data is presented in 3.2

Listing 3.2: Outline of VBO Creation, Binding, and Loading

```
GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, num_of_vertices*sizeof(float),
             vertices_to_load, GL_STATIC_DRAW);
```

The second technique employed was using texture memory (25) to store the second matrix on the GPU. Texture memory is generally a separate block of memory on the GPU that is used to store images. These images can then be mapped onto the object by the Fragment Shader by sampling colour of the pixel in an image, or texture, that corresponds to a point on the object being drawn. Values in a texture can be accessed by their coordinates. This is called mapping a texture onto an object, however is not needed in the scope of this thesis. All that is required of textures is that they store a grid of values which can be accessed in any sequence using a coordinate system. This implementation employs the use of textures to load the second matrix onto the GPU. This means that between a Vertex buffer and texture memory, both matrices can be loaded on the GPU to the fullest extent allowed by the available memory. An example of how this may be done can be seen in listing 3.3

This is very useful for the task at hand since the matrices only need to be loaded onto the GPU once. The first one can be iterated over while the corresponding values in the second matrix can be randomly accessed in texture memory. If both matrices were stored in Vertex Buffers then the matrices would require processing to shift their contents so that the values in each buffer lined up. This buffer manipulation would be expensive and would also involve breaking the computation on the GPU into smaller chunks to perform this processing to reline up the data in the matrices. This would result in a lot of additional time spent not only on processing but also stopping and starting the GPU, which involves synchronising all of the threads on the GPU at the end of the calculation multiple times, another expensive operation. It is more optimal to call the GPU to run over as many vertices as possible in one draw call. This is what textures allow by providing random access to the contents of the second matrix there is no need to line up the values of the two matrices, rather call repeatedly run over the Vertex Buffer the required number of times in one call to the GPU accessing the corresponding texture values.

Textures have particular implementation details with need to be accounted for, and these differ between OpenGL and OpenGL ES. When the implementation targets OpenGL, it uses a type of texture called `GL_TEXTURE_RECTANGLE` (26). For legacy reasons most texture types, with the exception of `GL_TEXTURE_RECTANGLE`, require the dimensions of a texture image to be a power of two. This is to allow for mipmapping of textures, a concept beyond the scope of this thesis. This would put a limitation on the use of texture memory since

most matrices will not have a height and width of a power of two. While it would be possible to pad the matrix with zeros to ensure it had appropriate dimensions this would be potentially very inefficient use of memory since powers of two are not terribly common. For example, a matrix 65 by 65 would need to be padded to become 128 by 128, nearly quadrupling the memory needed. This is not really a feasible approach since GPUs usually have a reasonably small amount of texture memory. However `GL_TEXTURE_RECTANGLE` is available in OpenGL and does not have this limitation, allowing textures to have any size permissible with memory constraints. It also has the benefit of not normalising the texture coordinates between zero and one. This means that to coordinates of the texture are accessed using integers ranging between zero and the height or width of the texture. For this implementation this means one less step is required when accessing values in the texture.

However `GL_TEXTURE_RECTANGLE` is not available in OpenGL ES (27). Since OpenGL ES in newer and has shed some of the legacy of OpenGL, it is not necessary for any texture type to have dimensions that are a power of two. This means that a different texture type can be used for OpenGL ES without a significant increase in memory usage. However there are no texture types available in OpenGL ES which support un-normalised coordinates, meaning that accessing texture memory will involve additional work. Rather than maintain separate code for OpenGL and OpenGL ES macros are used to generate the required code for which is needed at compile time. One last relevant detail of the texture implementation is that it is necessary to set a flag to ensure that when a texture is accessed using coordinates that the returned value is the one nearest that coordinate rather than a linearly interpolated between the nearest points, which may be more desirable in graphics processing.

Listing 3.3: Outline of Texture Creation, Binding, and Loading

```
GLuint tex;
glGenTextures(1, &tex);
glBindTexture(texture_type, tex);
glTexParameteri(texture_type, ..., GL_NEAREST);
glTexImage2D(texture_type, ..., width, height,
             ..., texture_data);
```

The third way that data can be made accessible for the GPU is by creating uniform variables (28). These are variables that can set for the code running on the GPU to access. A way to think of these is that the GPU is calling a function and these are global variables that are immutable, but always available. They are limited to a small range of types and are used for setting integers that are needed for computations on the GPU. An example of how they are set can be seen in listing 3.4.

Listing 3.4: Outline of VBO Creation, Binding, and Loading

```
GLint location_on_GPU = glGetUniformLocation("variable_name");
glProgramUniform1i(..., location_of_variable_on_GPU,
                   variable_value);
```

3.3 Performing the Computation

Now that the two matrices have been loaded onto the GPU it is time to perform some work on them. As described previously OpenGL would use the vertices contained in a Vertex Buffer to draw objects to the screen. These could be the vertices of a triangle for example. To draw the triangle, it is necessary to specify how the vertices are arranged in the Vertex Buffer. For instance these vertices could be coordinates given in two, three, or four dimensions. The vertices for the triangle being drawn would be passed through a Vertex Shader, which would perform transformations on the vertices to scale or rotate the triangle. The Vertex Shader can perform any work that can be programmed in GLSL (OpenGL Shading Language) and this is where the multiplication will be performed. This will be discussed more later in the chapter. Once the Vertex Shader has performed its computation with a vertex it will pass the result to the next stage in the graphics pipeline.

Since the OpenGL is used for graphics processing it is built to handle vertices, or inputs, as vectors of size 1, 2, 3, or 4. This means that the matrices need to be treated as chunks of one of these sizes. This information is defined in a Vertex Attribute Array (24), which is essentially meta data for the Vertex Buffer. How this size is determined and handled is discussed in a later section. This results in the first matrix being split up into independent chunks for processing on multiple GPU cores. These chunks are fed into the graphics pipeline like vertices. Since the values of the first matrix are used multiple times, depending on the dimensions of the second matrix, it is necessary to used 'instanced drawing' (29). This was originally intended for drawing multiple instances of the same object in a Vertex Buffer. For example the Vertex Buffer may contain a tree, and the number of instances is the number of trees to be drawn. Since matrix multiplication involves performing the dot product between every row in the first matrix and every column in the second, the vertices consist of every row of the first matrix while the instances can be used to represent each column of the second matrix. An example of how this 'instanced drawing' would be performed can be seen in listing 3.5

Listing 3.5: Outline of Drawing Procedure in OpenGL

```
glEnableVertexAttribArray(input_values);
glVertexAttribPointer(input_values, vector_size, ...);
glDrawArraysInstanced(GL_POINTS, offset, number_of_vertices,
                      instances);
```

3.4 Retrieving Results from the GPU

Once the computation on the GPU is complete it is necessary to retrieve the results from the GPU. OpenGL was originally intended for processing vertices and pushing the resulting pixels to a screen. There is no stage of the graphics pipeline, shown in figure 2.16, that involves pushing data back to the CPU as this is unnecessary for drawing to the screen and expensive in terms of time spent transferring the data on a bus. For the purposes of performing the computation of matrix multiplication however it is necessary to retrieve the results at some stage in the pipeline.

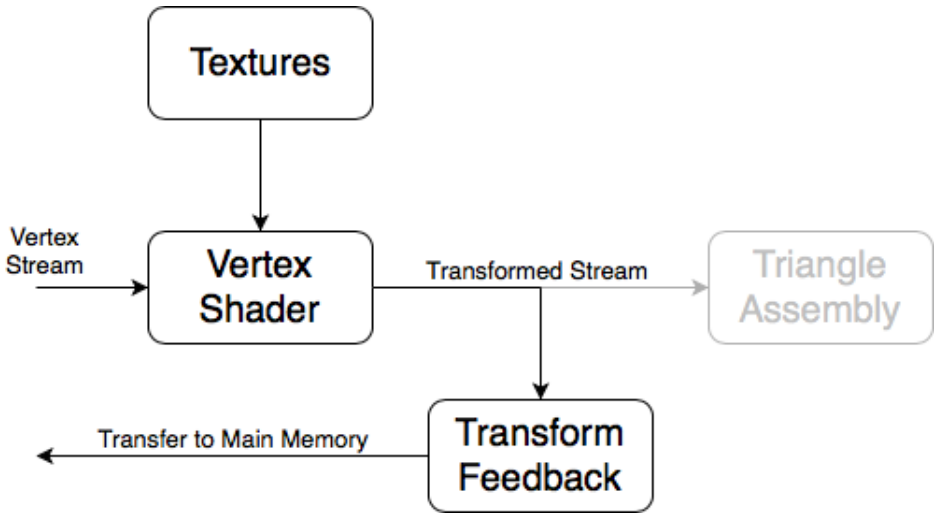

Figure 3.1: How Transform Feedback fits into the Graphics Pipeline

Transform Feedback is method of retrieving the result of the Vertex Shader and making them accessible from the CPU by transferring them back to main memory (30, 31). An illustration of how this fits into the graphics pipeline can be seen in figure 3.1. The original purpose of this method was for processing vertices multiple times for particle effects in video games. Since the rest of the pipeline is not need the following stages of the pipeline should be discarded. How this Transform Feedback technique is employed can be seen in listing 3.6. A buffer object must be created to represent where the results of the Vertex Shader are stored and bound to the current context. This buffer will later allow these results to be accessed from the CPU. glBeginTransformFeedback() will start pushing results of the Vertex

Shader to the buffer, while glEndTransformFeedback() will ensure that the GPU is finished and all of the results are in the buffer ready to be accessed.

Listing 3.6: Outline of Transform Feedback Procedure in OpenGL

```
// Setup buffer to store Vertex Shader output
glBindBuffer(GL_TRANSFORM_FEEDBACK_BUFFER, tbo);
glBufferData(..., tbo_size*sizeof(float), ...);
// Perform any other additional setup
glEnable(GL_RASTERIZER_DISCARD);
glBeginTransformFeedback(GL_POINTS);
// Perfrom drawing
glEndTransformFeedback();
// Results are now accessible in the buffer
```

3.5 Transpositions

The implementation can accept either matrix, or both, in a transposed state. This involves the matrix being inverted such that the top left and bottom right corners of the matrix stay the same, but the top right and bottom left are swapped. This means the top row of the matrix becomes the first column of its transposed version. With two input matrix this means there are four combinations of matrices that the implementation can work with. The four combinations are outlined in the remainder of this section, along with how the second matrix is arranged in texture memory, and how coordinates in the texture are determined.

Texture memory is block of memory on the GPU with a width and a height. When textures are loaded into this memory they do no fill the memory sequentially but as the images shape. For example, if an image was loaded into texture memory then rather than filling the first block of bytes sufficient for storing the image the image would fill the upper left corner of the block of texture memory. This meant that if an image being loaded was wider then the block of texture memory it would fail to load, even if the image had a smaller height then the height of the block and there was enough texture memory to store the image. The image dimensions need to fit within the texture memory block size. This is not the most efficient use of texture memory for the purposes of this implementation since it would potentially require breaking up the second matrix even though it can theoretically fit. Embedded devices would be particularly susceptible to this since texture memory is more constrained there. To avoid this, matrices were loaded to completely fill the width of the texture memory available, meaning that multiple rows, or part of a row, could be in the same row of the texture. Since a texture needs to be square in dimensions some padding was added to the second matrix to fill in the missing area if required.

3.5.1 Matrix A and Matrix B

With both matrices in their untransposed state, the rows of matrix A are multiplied by every column, or instance, of matrix B, shown by dots in figure 3.2. Since a column of matrix B is used, the values located in texture memory are spread out for a single column, also shown in the figure. The formulas used to calculate where in texture memory the required value of

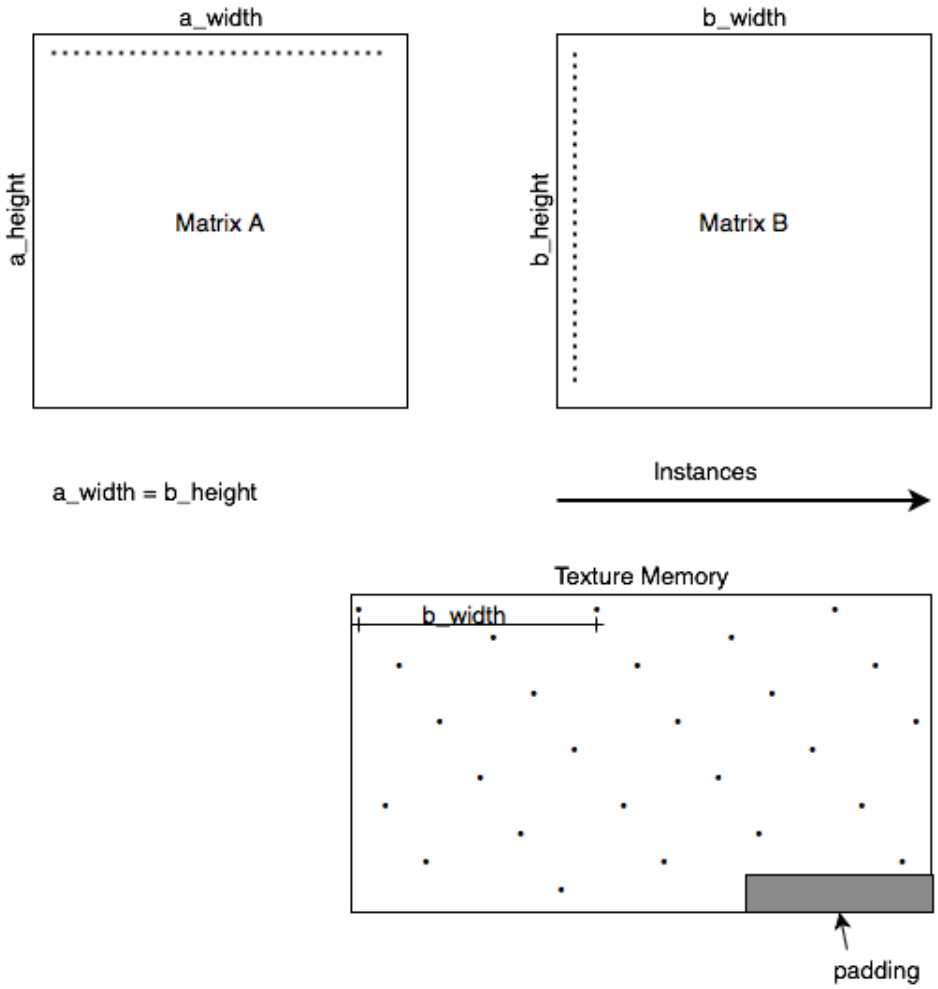

Figure 3.2: Layout of Matrix A and B, and how B is arranged in texture memory

matrix B is can be found below. gl_VertexID is the index of the vertex being processed, or where in matrix A is being accessed. gl_InstanceID is the index of the instance being processed, or which column of matrix B to access.

$$offset = b\_width(gl\_VertexID\%b\_height) + (gl\_InstanceID) \tag{1}$$

$$x\_coord = offset\%texture\_memory\_width \tag{2}$$

$$y\_coord = offset/texture\_memory\_width \tag{3}$$

3.5.2 Matrix A and Matrix B^T

The rows of matrix A are multiplied by every row, or instance, of matrix B, shown by dots in figure 3.3. Since a row of matrix B is used, the values located in texture memory are grouped together for a single row, also shown in the figure. The formulas used to calculate where in

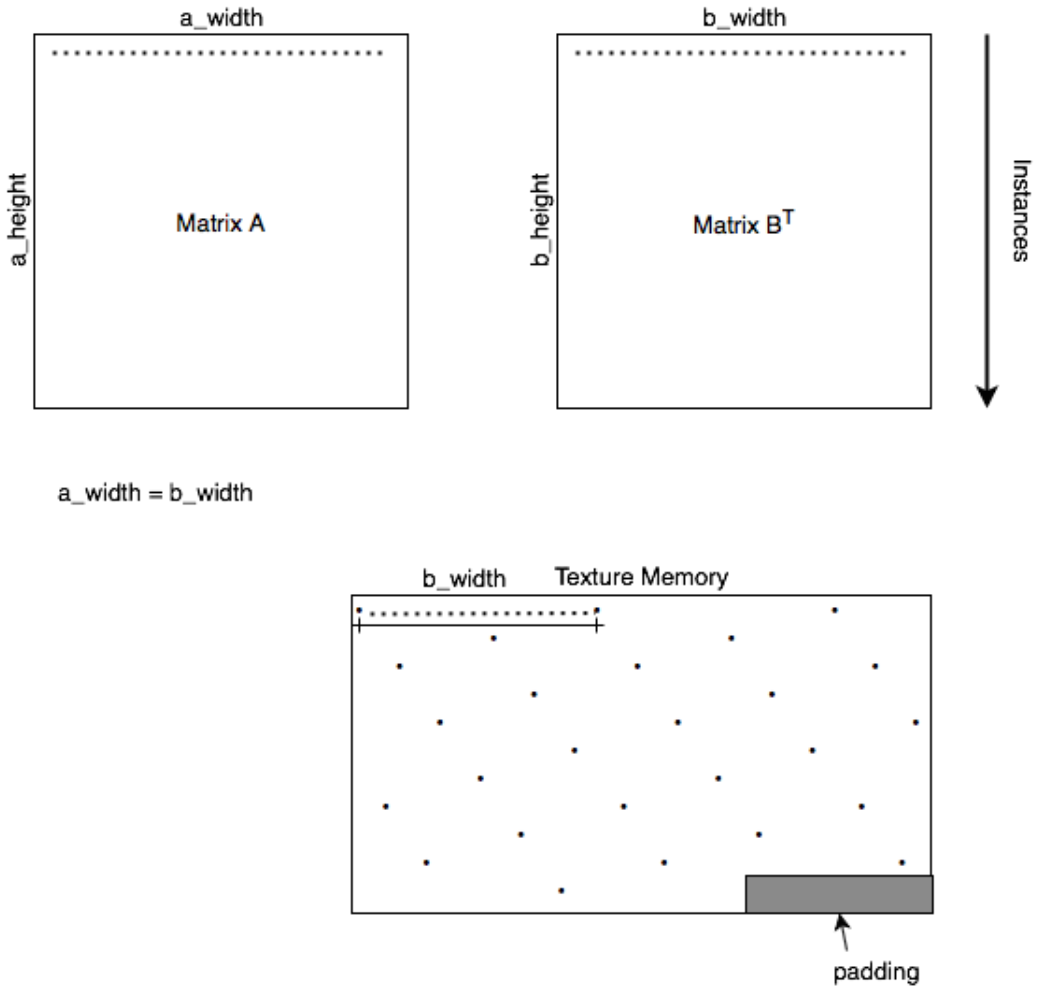

Figure 3.3: Layout of Matrix A and B^T, and how B^T is arranged in texture memory

texture memory the required value of matrix B is can be found below. gl_VertexID is the index of the vertex being processed, or where in matrix A is being accessed. gl_InstanceID is the index of the instance being processed, or which row of matrix B to access.

$$offset = b\_width(gl\_InstanceID) + (gl\_VertexID\%b\_width) \tag{4}$$

$$x\_coord = offset\%texture\_memory\_width \tag{5}$$

$$y\_coord = offset/texture\_memory\_width \tag{6}$$

3.5.3 Matrix A^T and Matrix B

In this case the columns of matrix A are multiplied by every column, or instance, of matrix B, shown by dots in figure 3.4. It is worth noting that due to the way matrix A is grouped into vectors for OpenGL, in cases with matrix A being transposed these vectors must be of size one due to memory layout. This results is a GPU core only being able to access one value for a give computation. The formulas used to calculate where in texture memory the

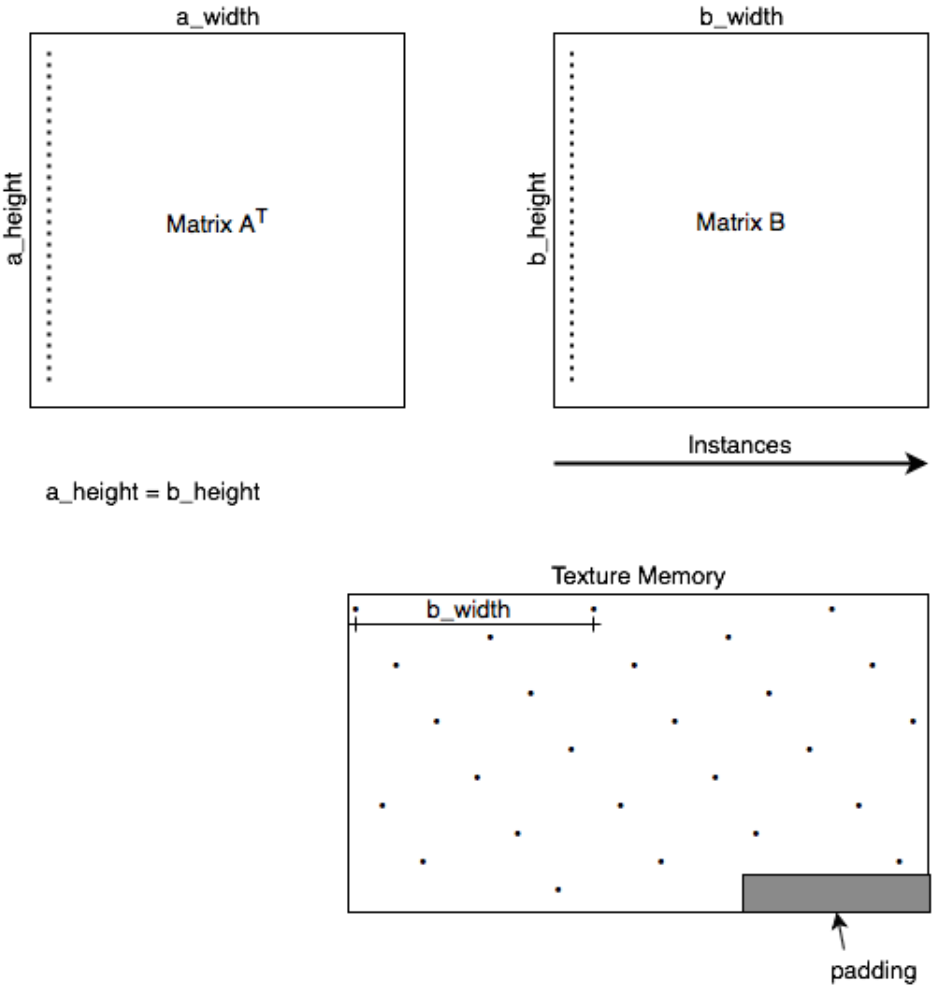

Figure 3.4: Layout of Matrix A^T and B, and how B is arranged in texture memory

required value of matrix B is can be found below. gl_VertexID is the index of the vertex being processed, or where in matrix A is being accessed. gl_InstanceID is the index of the instance being processed, or which column of matrix B to access.

$$offset = b\_width(gl\_VertexID\%b\_height) + (gl\_InstanceID) \tag{7}$$

$$x\_coord = offset\%texture\_memory\_width \tag{8}$$

$$y\_coord = offset/texture\_memory\_width \tag{9}$$

3.5.4 Matrix A^T and Matrix B^T

Finally the columns of matrix A are multiplied by every row, or instance, of matrix B, shown by dots in figure 3.5. It is worth noting again that due to the way matrix A is grouped into vectors for OpenGL, in cases with matrix A being transposed these vectors must be of size one due to memory layout. This results is a GPU core only being able to access one value for a give computation. The formulas used to calculate where in texture memory the

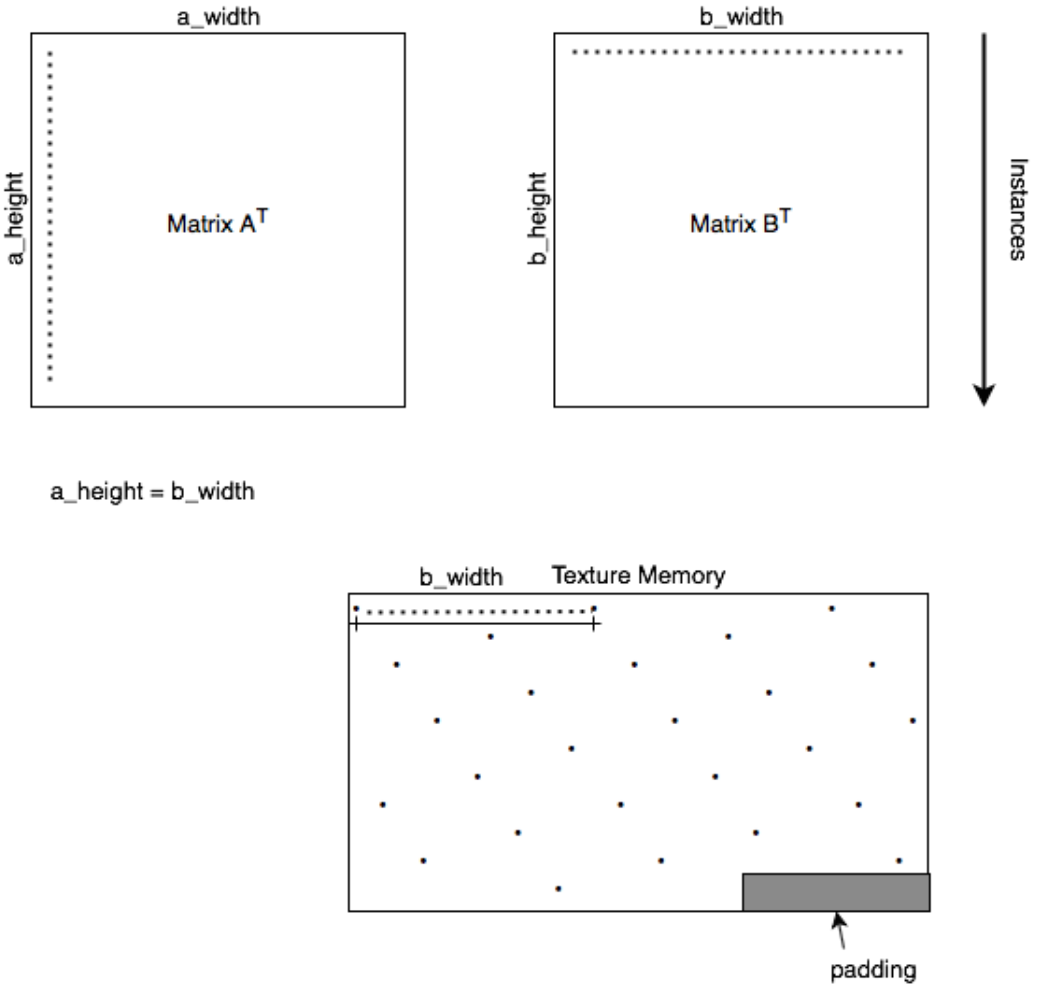

Figure 3.5: Layout of Matrix A^T and B^T, and how B^T is arranged in texture memory

required value of matrix B is can be found below. gl_VertexID is the index of the vertex being processed, or where in matrix A is being accessed. gl_InstanceID is the index of the instance being processed, or which column of matrix B to access.

$$offset = b\_width(gl\_InstanceID) + (gl\_VertexID\%b\_width) \tag{10}$$

$$x\_coord = offset\%texture\_memory\_width \tag{11}$$

$$y\_coord = offset/texture\_memory\_width \tag{12}$$

3.6 Shader Programmes

As discussed previously, the graphics pipeline contains two shader stages, the Vertex Shader and the Fragment Shader. Both stages are programmable using GLSL (OpenGL Shading Language), which is syntactically a C based language. The Vertex Shader receives a stream of vertices as input and perform transformations on these to rotate, scale, or transpose them as required. The Fragment Shader receives a stream of fragments, or pixels, and colours them, either using colours supplied with the vertices or textures.

Since the implementation uses transform feedback to retrieve the results of the Vertex Shader and discards the pipeline after this stage the Fragment Shader is not relevant to this implementation. When targeting OpenGL the implementation can disregard the Fragment Shader completely, however with OpenGL ES a Fragment Shader is required so a basic one is provided in this case even though it will never actually be used. Different versions of GLSL are available between OpenGL and OpenGL ES so slight changes need to be made to the shader at compile time with macros to provide support for both.

The shader programme provided by the implementation runs on each of the cores available on the GPU. The multiple cores run the shader programme in synchronization on different vertices, thus the Single Instruction Multiple Data (SIMD) architecture of the GPU. The shader programme needs to be compiled and linked to the OpenGL context at runtime. The instructions would then be loaded onto the GPU to be executed during a draw call.

The code in listing 3.7 shows the outline of a shader programme. It is necessary to normalise texture coordinates if the implementation is compiled for OpenGL ES, so macros are used, slightly edited in the example for clarity, to provide code for normalising coordinates when needed according to the formula in Equation 4, where x is the coordinate in the matrix from 0 to M, and M is the matrix size in that dimension. In cases where coordinates do not need to be normalised the macro does not have an effect.

$$normalised\_x\_coord = (2x + 1)/(2M) \tag{13}$$

The size of the vector used for the shader, represented as vecN in the example, represents how many vertices or values of matrix A are grouped together. This value could be 1, 2, 3, or 4. It is chosen to be the largest value that divided evenly into a row or column being multiplied of matrix A. This is so that the vector used lines up with the edges of the matrix when it is broken into pieces.

Listing 3.7: Outline of a Vertex Shader

```
// the texture coordinates need to be normalised for OpenGL ES
#if defined(BENCHMARK_OPENGLES)
#define NORMALISED_X(x) float(2*x +1)/float(2*tex_w)
#define NORMALISED_Y(y) float(2*y +1)/float(2*tex_h)
#else
#define NORMALISED_X(x) x
#define NORMALISED_Y(y) y
#endif

// calculate x and y coordinates as described previously
#define BX(j) ((offset+(j*mat_b_w))%tex_w)
#define BY(k) ((offset+(k*mat_b_w))/tex_w)

  VERSION
  SAMPLER
  in vecN input_values;
  UNIFORMS_OUT
  void main()
  {
    // calculate offset in Texture Memory as described
       previously
    int offset = mat_b_w*((gl_VertexID*N)%mat_b_h) +
       gl_InstanceID;
    float tex0 = TEXTURE(tex, vec2(NORMALISED_X(BX(0)),
                                   NORMALISED_Y(BY(0))))[0];
    // repeat for each texture sample
    vecN t = vec3(tex0, ...);
    // multiply the values from matrix A and B
    out_value = dot(input_values, t);
  };
```

When matrix B is not transposed it is necessary to sample the texture separate for each value needed since the values in a column are spread out through texture memory. This can be seen in figures 3.2 and 3.4. This is performed in the shader by the macros BX and BY, which can access further into texture memory using the j and k parameters respectively. When matrix B is transposed then the values needed are all in the same row of the matrix, as seen in figures 3.3 and 3.5, and can be accessed in one texture sample since the values are located together in memory.

3.7 Error Checking

The OpenGL GEMM function also implements the appropriate error checking to ensure correct behaviour. For some OpenGL function, the error can be caught by checking the return value. These functions will return a NULL pointer when they fail while others return void and it is necessary to call glGetError() to determine if there was an error (32). This could happen if an OpenGL function requires memory to be allocated on the GPU, and the implementation is such the function either does not actually allocate the memory when it is called, but rather later on in time, or it does not block on GPU I/O before returning. These errors are generated by OpenGL and placed on a list. glGetError() will then return the first item on this, and as such is called in a loop to retrieve all errors. An example of a case that would produce an OpenGL error would be if there was insufficient memory on the GPU.

Another check that is performed is ensuring that there is enough texture memory on the device to store the contents of the second matrix. It is possible to ascertain the amount of texture memory available with OpenGL. All of texture memory can be utilised by the implementation as described previously, thus if the size of the second matrix is greater than the amount of texture memory on the device then the implementation will return an error.

3.8 Final Steps

The results of the Vertex Shader are stored in the transform feedback buffer on the GPU. This buffer is accessible by copying its contents to main memory, however transferring large amounts of data to main memory from the GPUs memory is expensive. Rather than copy the result the implementation takes advantage of a function glMapBufferRange() to map the contents of the buffer into the processes address space (33). This allows for reading and writing to the buffer from the process running on CPU without an expensive copy being performed between the GPU and the CPU.

Once the results are available to the CPU, some final work must be done to sum up sections of the results to produce the final output. Since the rows of matrix A were broken up into vectors of size N to be processed, this means that not all the addition has been performed. The results are thus broken into `row_length/N` pieces. These are the pieces that need to be summed to compute the final result. This summation is done in place, by the CPU, with two for loops, the outer loop retrieves the value of the first chunk of each row, while the inner loop adds the following `row_length/N` pieces to the first value.

The buffer now contains the final output matrix from the multiplication. This output is finally added to, subtracted from, or replaces, the existing values in the memory buffer passed to the function to contain the output. Any memory or buffers allocated are deallocated. Any shader programmes and textures are destroyed. The function will then return.

4 Evaluation

This section will outline the process of evaluation used for this thesis. First will be a description of the methods used for compiling, running, and measuring the performance of the implementation. This will be followed with an overview of the various algorithms used to call the GEMM implementation. The algorithms would construct a patch matrix and a kernel matrix to perform a convolution with a GEMM function. Depending on the algorithm the matrices will have a different transposition or layouts to compare performance. These algorithms will tested for a range of different sized convolutions that compose the AlexNet neural network, which will also be specified. Following this the results of the evaluation, or benchmarks, will be provided with for a number of devices of varying performance capabilities, running different operating systems. A discussion of the results will follow to summarise and interpret the findings. Finally a description of future work that could be done to potentially improve the implementation is provided.

4.1 Benchmarking

The library 'triNNity-benchmarks', which is a companion library for testing the performance of algorithms in the 'triNNity' library, was used for evaluating the performance of the implementation. Using this library had two main advantages. First since the implementation was being developed as part of the 'triNNity' library, it would be be minimal effort to use its companion benchmarking library. Secondly using an off the shelf benchmarking library, rather than developing evaluation methods from scratch, provided all of the benchmarking functionality upfront with a variety of convolution sizes and algorithms available for comparison. This allowed for more time to be dedicated towards developing and improving the implementation rather than developing benchmarking code.

4.1.1 Compilation

For benchmarking a GEMM implementation a definition is passed to the compiler to specify what GEMM implementation to use. The two GEMM implementations used for evaluation are `GEMM_SIMPLE`, a straightforward GEMM implementation that performs computations entirely on the CPU, and `GEMM_OPENGL`, the GEMM implementation described in this thesis. `GEMM_SIMPLE` is an equivalent CPU implementation that is used to provide a baseline benchmark to compare with the benchmark of OpenGL implementation.

The scenarios used, which are the convolutions of various size, are specified at compile time. Also specified is the selection of algorithms to benchmark, by means of definitions passed to the compiler. For each scenario a different executable is created. This executable will contain code to run each of the algorithms specified to compute the result of the convolution for that scenario. All of these algorithms are run from main in the benchmark library. Whether a particular algorithm is run depends on whether the definition is present.

The code for these algorithms is contained in a the 'triNNity' library, which must be installed on the device. The libraries or frameworks that are needed for compilation must also be specified at compile time. Also specified is what compiler, and what compiler optimisations are used for the benchmark.

All of this information is specified in a Makefile for the benchmark.

4.1.2 Validation and Results

The executable produced by the compiler can then be run on the device. The benchmark serves two purposes.

The first purpose of the benchmark is to validate the results of a convolution. When the benchmark runs an algorithm it will validate the results by printing the error in the result to standard output. This difference between the expected result and the computed result. No results provided were benchmarks of algorithms that produced incorrect results.

The second purpose of the benchmark is to measure the time taken to execute the algorithm. This measurement is the number of CPU clock cycles elapsed between the algorithm being called and returning the result. This is refereed to as the execution time of the algorithm.

The benchmark for an algorithm will run a number of times. For this evaluation all benchmarks were computed with 5 runs. This allows for an average execution time to be established and hopefully mitigates variation, such as variation resulting from CPU load changing due to other processes on the system. The first of these runs is discarded from the results since the execution time of the first run is usually a significantly outlier in the results. The algorithm will usually converge to a stable execution time over after the first or second run.

4.1.3 GEMM Variations

The 'triNNity' contains the following four variations of an im2row algorithm that calls a GEMM function. These variants are refereed to by the following names.

- im2row-scan-ab-ik
- im2row-scan-abt-ik
- im2row-scan-abt-ki

The letters a and b in the third section of the name refer to the matrices a and b that the algorithm creates. Whether there is a t following the letter signifies whether that matrix is transposed. So for example the fourth name contains 'atbt' signifying that both matrices are transposed. The fourth section of the name signifies the order that the matrices are passed to the GEMM function. So for 'ik' the image patch matrix is matrix a and the kernel matrix is matrix b, while for 'ki' the opposite is the case. For algorithms that use OpenGL, the name is prepended with 'opengl'.

4.1.4 Scenarios

There were a range of images and kernels used for evaluating the implementation. These are called 'scenarios'. The four scenarios used are specified in table 4.1, and are four scenarios from the AlexNet convolution network. Another scenario labelled 'test' is specified which is a small scenario used to verify that the GEMM implementation is working on devices where memory is too constrained for the current implementation.

Table 4.1: The specifications of each convolution scenario tested

Scenario	Kernels	Channels	Stride	Image Width	Image Height	Kernel Size
test	1	1	1	4	4	3
2	256	96	1	27	27	5
3	384	256	1	13	13	3
4	384	384	1	13	13	3
5	256	384	1	13	13	3

The dimensions of the image patch matrix and kernel matrix can be calculated for each scenario using the following formulas.

$$image\_patch\_matrix\_width = channels * (kernel\_size^2) \tag{1}$$

$$image\_patch\_matrix\_height = image\_height * image\_width \tag{2}$$

$$kernel\_matrix\_width = kernels \tag{3}$$

$$kernel\_matrix\_height = channels * (kernel\_size^2) \tag{4}$$

Note that these are the dimensions for both matrices in their untransposed state. If a matrix is transposed that the dimensions are swapped. Also note that the image patch matrix width is equal to the kernel matrix height in their untransposed states.

4.2 Results

This section contains the relevant technical specifications of the devices used for evaluating the implementation and the results produced from benchmarks.

4.2.1 MacBook Air

Development of the implementation was done on a MacBook Air, which provided the bulk of the results for this thesis.

Table 4.2: Technical Specifications for MacBook Air (Early 2014) (34)

CPU	1.4 GHz Intel Core i5
Memory	4 GB 1600 MHz DDR3
GPU	Intel HD Graphics 5000 1536 MB of VRAM
Operating System	MacOS 10.13.4
OpenGL Support	Version 4.1
OpenGL ES Support	NA
Compiler used	Apple LLVM version 9.1.0 (clang-902.0.39.1)
Run in 'headless' mode	No

Table 4.3: Benchmarks results for a MacBook Air (10^9 CPU Cycles)

	Scenario			
Algorithm	2	3	4	5
im2row-scan-ab-ik	3.24	0.95	1.99	1.48
opengl-im2row-scan-ab-ik	1.61	0.96	1.44	0.56
	49.84%	101.29%	72.57%	38.17%
im2row-scan-abt-ik	1.05	0.35	0.54	0.38
opengl-im2row-scan-abt-ik	1.54	0.56	0.80	0.56
	146.51%	159.15%	146.74%	149.33%
im2row-scan-abt-ki	1.10	0.34	0.52	0.34
opengl-im2row-scan-abt-ki	1.51	0.59	0.81	0.56
	137.25%	175.13%	155.36%	162.51%

In figure 4.1 the results in table 4.3 are shown in a bar chart. The algorithms are keyed by colour, while the results for a particular scenario are grouped together from 'conv2' to 'conv5'.

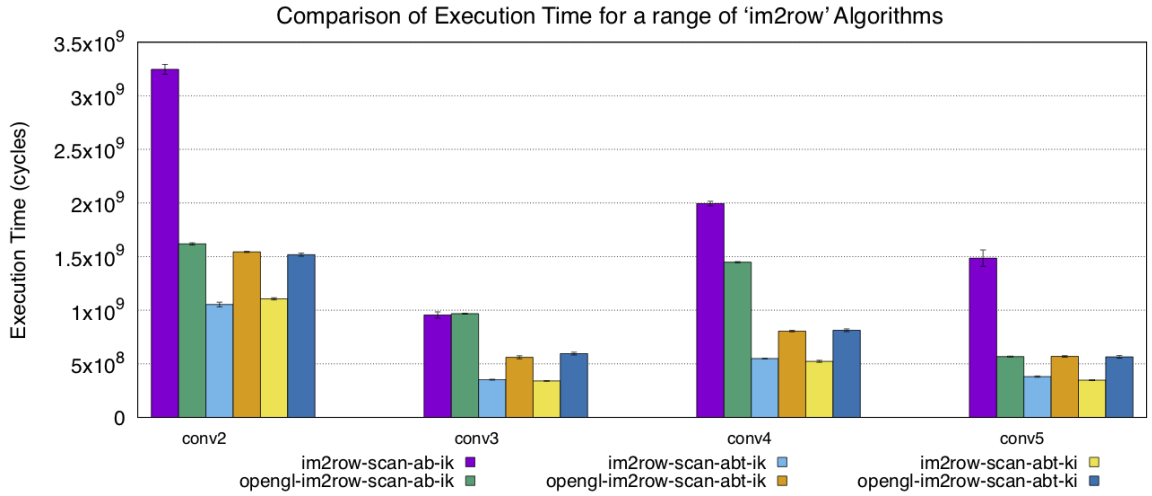

Figure 4.1: Benchmark Results for the GEMM implementation on a MacBook Air

Table 4.4 shows the results obtained by running the scenario 2 benchmark with Apples OpenGL Profiler on the MacBook Air.

Table 4.4: OpenGL Profiler Results on a MacBook Air with times give in micro seconds

GL Function	Calls	Total Time	Avg Time	% GL Time	% App Time
glEndTransformFeedback	15	6009157	400610.48	83.17	21.72
glDeleteBuffers	30	866148	29867.18	11.99	3.13
glDrawArraysInstanced	15	128042	8536.17	1.77	0.46
glBufferData	30	117602	3920.08	1.63	0.42
glCompileShader	15	49571	3304.78	0.69	0.18
glTexImage2D	15	24544	1636.28	0.34	0.09
glLinkProgram	15	7724	514.94	0.11	0.03
glBeginTransformFeedback	15	4900	326.68	0.07	0.02
glFlush	15	4341	289.44	0.06	0.02

4.2.2 ASUS Tinkerboard

Once the implementation was developed it was then ported to support OpenGL ES. The implementation was run on an ASUS Tinkerboard. The results for this device are only from the initial stages due to the time constraints on this thesis. As such only the test scenario was run successfully on the device.

Table 4.5: Technical Specifications for an ASUS Tinkerboard (35)

CPU	1.8GHz ARM Cortex-A17
Memory	2GB LPDDR3
GPU	ARM Mali T764
Operating System	Debian Linux 9
OpenGL Support	NA
OpenGL ES Support	3.0
Compiler used	g++ 6.3.0
Run in 'headless' mode	Yes

Table 4.6: Benchmarks for an ASUS Tinkerboard

Algorithm	test scenario
im2row-scan-ab-ik	Success
opengl-im2row-scan-ab-ik	Success
im2row-scan-abt-ik	Success
opengl-im2row-scan-abt-ik	Success
im2row-scan-abt-ki	Success
opengl-im2row-scan-abt-ki	Success

4.3 Discussion

The results outlined in section 4.2 show significant promise for achieving the goals of the thesis, while also emphasising the further work that would need to be done to fully achieve those goals.

The primary results were achieved on the MacBook Air. These showed that the implementation developed for this thesis with OpenGL achieved better performance than an equivalent CPU implementation when both matrices are created in their untransposed states. For this matrix arrangement, table 4.3 shows that the OpenGL version executed in the worst case a roughly equivalent time to the CPU version, for scenario 3, and in the best case at around 38% of the time of the CPU version, for scenario 5. These two algorithms, 'im2row-scan-ab-ik' and 'opengl-im2row-scan-ab-ik', were also the worst performing of the algorithms tested. With neither matrix transposed the individual kernels were arranged in column of the kernel matrix. As previously described, this meant that the contents of a kernel are spread throughout memory rather than grouped. Having kernel values grouped together would have significant advantages when reading from memory since the values could be accessed together in one read from memory. However when the values are spread out this would require multiple reads. The results would seem to indicate that texture memory on the GPU has significantly better performance for this random accesses than main memory. This advantage means that the OpenGL version of this algorithm has a potential reduction in execution time of around 62%.

The remaining two algorithms have some clear similarities in their results. The CPU versions, 'im2row-scan-abt-ik' and 'im2row-scan-abt-ki', had very close execution times across all scenarios. This is explained by the fact that the difference between them is that the order of the input matrices is swapped between them. As such for the first matrix A is the image patch matrix, while in the second matrix A is the kernel matrix. This difference is slight, but essentially shows that there is virtually no advantage to the matrices being in a particular order. For example the kernel matrix is usually smaller, but there is no advantage to the matrices being ordered by size. The same is true for the results for the OpenGL versions of these algorithms, where there is very little difference when the order of the matrices change. This comparison on the basis of matrix order would become more relevant for the OpenGL version of the algorithm when run on more constrained devices however. Texture memory on GPUs is very limited and on more constrained devices this is even more of an issue, to the point where is may not be possible to fit an entire matrix in texture memory. In these situations it would be necessary to break the matrix into smaller pieces that could fit into texture memory to be processed separately. In cases like this there could be a performance reduction from processing the matrix in multiple pieces, and the comparison of these two

algorithms would demonstrate the extent to which this performance reduction existed.

It is also worth noting that for the second two algorithms the CPU version outperform the OpenGL version, with execution times around a third faster. With these versions of the algorithm the kernel matrix is ordered in rows, meaning that the values for a particular kernel are grouped. These versions of the algorithm therefore benefit from being able to access values in sequence from the kernel matrix, and thus doing so in fewer reads from memory. From this it can be gathered, with the superior performance of the CPU version, that this advantage is greater realised on the CPU in this case. While texture memory is superior for random accesses, main memory has better performance with sequential reads, likely due to CPU caching. As such the fastest algorithms in the results are CPU based, however for matrix transposition that require non-sequential reads from memory the GPU version is superior.

Also provided in are the results of profiling the OpenGL code, in table 4.4. This gives a rough idea of where the benchmark is spending time executing, or waiting on, OpenGL code. Note that this benchmark ran all three algorithms calling the OpenGL GEMM function, and all three algorithms calling the CPU GEMM function. As such the percentage of 'App Time' spent in OpenGL code is compared against the total time to run the benchmark, including CPU versions, setup of the benchmark, validating results, etc. Profiling results like these were used throughout development to pinpoint where OpenGL was taking up a lot of time, and to reduce this if possible. For example this profiling revealed that at one stage the implementation was spending a lot of time copying memory from a buffer on the GPU to main memory, which was then mitigated by using direct mapping of GPU memory to avoid copying amounts of data. This resulted in around a ten-fold increase in performance in some cases. It can be seen that the function 'glEndTransformFeedback' is taking up the vast majority of execution time spent in OpenGL. This is the function that blocks until the GPU has finished drawing, or processing, vertices and the results are in the transform feedback buffer. The next most significant amount of time was spent deleting OpenGL buffers, which is a surprisingly large amount of time to spend on a clean up operation. Ways to reduce the time spent on these functions is further discussed in the next section.

Preliminary results are also provided for running the benchmarks on an embedded device, in this case the ASUS Tinkerboard. These results show that the implementation worked for the small test scenario. This is a proof of concept, that the OpenGL implementation described in this thesis does work on an embedded device. For larger scenarios the implementation fails due to memory errors thrown by OpenGL. How these errors could be resolved is discussed further in the next section as prospective future work, since due to time constraints they could not be resolved for this thesis. However the proof of concept does illustrate that performing this type of work on the GPU of an embedded device is possible, while the results achieved on the MacBook Air demonstrate that improved performance can

be achieved with OpenGL. Thus this thesis describes an implementation that can, with further development, scale from a low powered embedded device to a desktop class device, while supporting two operating systems and two variations of OpenGL, to perform some of the computation involved in neural networks with potentially a performance increase. Even if the performance increase did not materialise with further development it may still be a productive endeavour since offloading the work onto otherwise idle GPUs on headless devices would free the CPU on those devices to perform other work.

4.4 Further Work

Memory errors

The first priority of any future development would be to resolve the memory errors that the implementation encounters on the ASUS Tinkerboard or any other embedded device. From work done up until this point it is likely that these errors result from GPU amount of texture memory available being calculated incorrectly. Even when it is calculated correctly it is necessary for the implementation to support breaking up the input matrices into smaller pieces, or chunks, for processing. These smaller chunks could then be loaded onto the GPU separately for processing or the workload could be shared between the GPU and the CPU.

Evaluate on more devices

It would also be important to evaluate the implementations performance on a wider range of devices in terms of computational power, ranging from more powerful desktop machines with state of the art discrete GPUs, to other low powered devices. Note it was attempted to run the implementation on a Raspberry Pi, however critical OpenGL ES features were not supported.

Vector size

At present the implementation computes the best vector size to use for handling any given matrix. This removes the need to process the matrix such that additional zeros are stuffed into the matrix to line rows up on a boundary according to a fixed sized vector. This avoids a lot of shifting data in memory to line up the rows on the boundary. This could be improved further by generating shaders that could handle multiple vectors simultaneously. These shaders could be generated with macros, and would mean the GPU cores could do more work at once. This would also reduce the amount of processing required by the CPU to sum results.

Asynchronous tasks

It may be feasible to perform some tasks of the implementation concurrently. For example, as previously mentioned, when breaking matrices into smaller pieces it may

be a good idea to share the work between the CPU and the GPU, which would require multiple threads on the CPU to perform the work and to wait for OpenGL to finish. It would also be worth exploring whether some tasks, such as deleting OpenGL buffers which took a not insignificant amount of time according to the profiling results, could be performed in a separate thread while other processing was done on the GPU. The other large amount of time spent in OpenGL is waiting for the results of transform feedback. Rather than waiting for all the results to then perform the additional processing described, it is possible to have another thread monitor how many results have been attained and then batch this post processing to occur as results are ready such that this work can be done while waiting for the rest of the results.

Experiment with other OpenGL functions

Another area that would be of interest would be exploring other OpenGL functionality available. For example with more advanced features it is possible for the GPU to write to images texture memory. This would remove the need for summing the results from the GPU, as this would now be performed by adding the results to the existing valued in the output 'image' (36).

Bug fixing

As with most programming endeavours there are issues with the implementation where it fails for certain scenarios for particular algorithms. These bugs would need to be addressed.

5 Conclusion

This section will present the conclusions that this thesis draws from the results obtained in attempting to complete the objectives of the research.

The first research question posed by this dissertation was "Is it possible to utilise the GPU found on a range of low power embedded devices to perform computations associated with DNNs using OpenGL?". This was achieved in principle, as shown in the results. The implementation described in the thesis does perform computations associated with DNNs, specifically matrix multiplication, using OpenGL to utilise the GPU. While only limited forms of these computations were performed on low power embedded devices, this demonstrated a proof of concept that this approach is feasible with further work.

The second research question was "Is it additionally possible that by performing this computation on the GPU with OpenGL that the time taken to perform this computation is less than an equivalent implementation that only uses the CPU?". This questions was also answered by the results of the thesis, which showed that for one variant of the algorithms tested, the OpenGL version was significantly faster, while for other variants the CPU version is faster. However without testing on a wider variety of hardware a conclusion on how the implementation performs is difficult to reach. The conclusion of this thesis is that, with the further development discussed in the previous section, the performance of the OpenGL version could be better established and improved further.

The objectives of this thesis were also broadly achieved. An implementation of a key DNN layer, the convolution layer, was developed using C++ and using OpenGL to utilise the GPU. It has cross platform support for both MacOS and Linux operating systems. The implementations can function on desktop computers, by targeting the desktop version of OpenGL, and low powered embedded devices, by targeting the version of OpenGL for embedded systems. Thus the same implementation can scale from tiny low powered ARM boards with integrated GPUs to desktop workstations with discrete GPUs. It also provides support across a larger range of devices than other GPU based solutions which rely on OpenCL, which is not available on many low powered devices. Objectives focused on the performance of the implementation were partially achieved, while testing across a broad range of devices was not. The implementation was also developed as part of the 'triNNity'

library.

The results of this thesis demonstrate the it is possible to use OpenGL to perform some of the work associated with DNNs on GPUs of low power embedded devices. By enabling more processing on these low power devices it could help reduce network traffic by allowing devices to only transmit results, which would also reduce latency, while also arguably improving user privacy. With further development, this could help enable a whole range of new applications one low powered devices that are better able interpret their environment, either through image or audio processing.

Bibliography

[1] Andrew Anderson. software-tools-group/trinnity repository. https://gitlab.scss.tcd.ie/software-tools-group/triNNity, . [Online; accessed 8-May-2018].

[2] Siri Team. Hey siri: An on-device dnn-powered voice trigger for apple's personal assistant. https://machinelearning.apple.com/2017/10/01/hey-siri.html. [Online; accessed 8-May-2018].

[3] Apple. Face id security. https://images.apple.com/business/docs/FaceID_Security_Guide.pdf, . [Online; accessed 8-May-2018].

[4] David Gregg. Final year projects. https://www.scss.tcd.ie/David.Gregg/fyp/. [Online; accessed 8-May-2018].

[5] The Khronos Group. Khronos releases opengl es 3.0 specification. https://www.khronos.org/news/press/khronos-releases-opengl-es-3.0-specification, . [Online; accessed 8-May-2018].

[6] A Anderson. software-tools-group/trinnity-benchmarks repository. https://gitlab.scss.tcd.ie/software-tools-group/triNNity-benchmarks, . [Online; accessed 8-May-2018].

[7] Osvaldo Simeone. A brief introduction to machine learning for engineers. *arXiv preprint arXiv:1709.02840*, 2017.

[8] John J Hopfield. Artificial neural networks. *IEEE Circuits and Devices Magazine*, 4(5): 3–10, 1988.

[9] Sanguthevar Rajasekaran and GA Vijayalakshmi Pai. *Neural networks, fuzzy logic and genetic algorithm: synthesis and applications (with cd)*. PHI Learning Pvt. Ltd., 2003.

[10] Jeff Heaton. *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[12] Andrej Karpathy. Convolutional neural networks for visual recognition. `http://cs231n.github.io/convolutional-networks/`. [Online; accessed 8-May-2018].

[13] Pete Warden. Why gemm is at the heart of deep learning. `https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/`. [Online; accessed 8-May-2018].

[14] Onur Mutlu. Computer architecture: Simd and gpus (part i). `https://www.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=onur-740-fall13-module5.1.1-simd-and-gpus-part1.pdf`, . [Online; accessed 8-May-2018].

[15] Onur Mutlu. Computer architecture: Simd and gpus (part ii). `https://pdfs.semanticscholar.org/presentation/7a5f/a4c0fe25f942bf41467c7e5dca56bfe2cdc7.pdf`, . [Online; accessed 8-May-2018].

[16] Jose M Domínguez, Alejandro JC Crespo, and Moncho Gómez-Gesteira. Optimization strategies for parallel cpu and gpu implementations of a meshfree particle method. *arXiv preprint arXiv:1110.3711*, 2011.

[17] The Khronos Group. The opengl graphics system: Specification (version 4.0). . [Online; accessed 8-May-2018].

[18] Microsoft. Graphics pipeline. [Online; accessed 8-May-2018].

[19] Geoff Leach. Lecture: Graphics pipeline and animation. [Online; accessed 8-May-2018].

[20] Glew. `http://glew.sourceforge.net`. [Online; accessed 8-May-2018].

[21] Glfw. `http://www.glfw.org`. [Online; accessed 8-May-2018].

[22] The Khronos Group. Egl. `https://www.khronos.org/egl`, . [Online; accessed 8-May-2018].

[23] Matus Novak. Opengl es 2 rendering without an x server on raspberry pi using egl. `https://github.com/matusnovak/rpi-opengl-without-x`. [Online; accessed 8-May-2018].

[24] The Khronos Group. Vertex specification. `https://www.khronos.org/opengl/wiki/Vertex_Specification`, . [Online; accessed 8-May-2018].

[25] The Khronos Group. Texture specification. https://www.khronos.org/opengl/wiki/Texture, . [Online; accessed 8-May-2018].

[26] The Khronos Group. Rectangle texture. https://www.khronos.org/opengl/wiki/Rectangle_Texture, . [Online; accessed 8-May-2018].

[27] The Khronos Group. glteximage2d. https://www.khronos.org/registry/OpenGL-Refpages/es3.0/html/glTexImage2D.xhtml, . [Online; accessed 8-May-2018].

[28] The Khronos Group. gluniform. https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glUniform.xhtml, . [Online; accessed 8-May-2018].

[29] The Khronos Group. gldrawarraysinstanced. https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glDrawArraysInstanced.xhtml, . [Online; accessed 8-May-2018].

[30] The Khronos Group. Transform feedback. https://www.khronos.org/opengl/wiki/Transform_Feedback, . [Online; accessed 8-May-2018].

[31] The Khronos Group. Transform feedback tutorial. https://open.gl/feedback, . [Online; accessed 8-May-2018].

[32] The Khronos Group. Opengl error. https://www.khronos.org/opengl/wiki/OpenGL_Error, . [Online; accessed 8-May-2018].

[33] The Khronos Group. glmapbufferrange. https://khronos.org/registry/OpenGL-Refpages/es3.0/html/glMapBufferRange.xhtml, . [Online; accessed 8-May-2018].

[34] Apple. Macbook air technical specification. https://support.apple.com/kb/sp700?locale=en_IE, . [Online; accessed 8-May-2018].

[35] ASUS. Tinkerboard technical specification. https://www.asus.com/ie/Single-Board-Computer/Tinker-Board/specifications/. [Online; accessed 8-May-2018].

[36] The Khronos Group. Image load store.
https://www.khronos.org/opengl/wiki/Image_Load_Store, . [Online; accessed 8-May-2018].

A1 Appendix

A1.1 Source code developed for the implementation

Listing A1.1: The GEMM function implemented using OpenGL

```
#if defined(TRINITY_USE_OPENGL_GEMM)
template <gemm_transpose_t transp>
static TRINNITY_INLINE void opengl_gemm_wrap(signed mat_a_w,
                        signed mat_a_h, signed mat_b_w,
                        signed mat_b_h,
                        const float* __restrict__ a,
                        const float* __restrict__ b,
                        gemm_accumulate_t accum,
                        float* __restrict__ c) {

  // mat_a_w == mat_b_h
  // mat_c_w == mat_b_w
  // mat_c_h == mat_a_h

  GLuint shaderProgramID;

  // Vertices need to be of size 1, 2, 3, 4
  // To avoid needing to zero stuff the matrix in the event
  // that the matrix row doesn't divide evenly by
  // a given vertix size, resolve to use the to the largest
     vertix
  // size that divides evenly, incl 1/
  // Ratio should be 33:17:25:25 for 1:2:3:4
  int chunk_size = 0;
  for(int i = 4; i > -1; i--) {
```

```
  if(mat_a_w % i == 0) {
    chunk_size = i;
    break;
  }
}
//  assert(chunk_size); // should be non-zero
int chunks_per_row = mat_a_w/chunk_size;
int tex_components;
int tex_internal_format;
int tex_format;
int instances;
int patches;

#if defined(BENCHMARK_OPENGLES)
int texture_type = GL_TEXTURE_2D;
#else
int texture_type = GL_TEXTURE_RECTANGLE;
#endif
#if defined(CHECK_GL_ERRORS)
triNNity::opengl::glGetErrorString("Before shader compilation")
  ;
#endif
switch (transp) {
  case GEMM_A_B: {
    switch (chunk_size) {
      case 1: {
        shaderProgramID = triNNity::opengl::compileShaders(
                          triNNity::opengl::gemmA_B_vec1);
      } break;
      case 2: {
        shaderProgramID = triNNity::opengl::compileShaders(
                          triNNity::opengl::gemmA_B_vec2);
      } break;
      case 3: {
        shaderProgramID = triNNity::opengl::compileShaders(
                          triNNity::opengl::gemmA_B_vec3);
      } break;
      case 4: {
        shaderProgramID = triNNity::opengl::compileShaders(
```

```
                        triNNity::opengl::gemmA_B_vec4);
    } break;
    default: {
      TRINNITY_ERROR("Not implemented - Shouldn't be
                      possible for this chunk_size");
    } break;
  }
  tex_components = 1;
  instances = mat_b_w;
  patches = mat_a_h;
} break;
case GEMM_A_BT: {
  switch (chunk_size) {
    case 1: {
      shaderProgramID = triNNity::opengl::compileShaders(
                        triNNity::opengl::gemmA_BT_vec1);
      tex_components = 1;
    } break;
    case 2: {
      shaderProgramID = triNNity::opengl::compileShaders(
                        triNNity::opengl::gemmA_BT_vec2);
      tex_components = 2;
    } break;
    case 3: {
      shaderProgramID = triNNity::opengl::compileShaders(
                        triNNity::opengl::gemmA_BT_vec3);
      tex_components = 3;
    } break;
    case 4: {
      shaderProgramID = triNNity::opengl::compileShaders(
                        triNNity::opengl::gemmA_BT_vec4);
      tex_components = 4;
    } break;
    default: {
      TRINNITY_ERROR("Not implemented - Shouldn't be
                      possible for this chunk_size");
    } break;
  }
  instances = mat_b_h;
```

```
    patches = mat_a_h;
  } break;
  case GEMM_AT_B: {
    chunk_size = 1;
    chunks_per_row = mat_a_h/chunk_size;
    shaderProgramID = triNNity::opengl::compileShaders(
                      triNNity::opengl::gemmA_B_vec1);
    tex_components = 1;
    instances = mat_b_w;
    patches = mat_a_w;
  } break;
  case GEMM_AT_BT: {
    chunk_size = 1;
    chunks_per_row = mat_a_h/chunk_size;
    shaderProgramID = triNNity::opengl::compileShaders(
                      triNNity::opengl::gemmA_BT_vec1);
    tex_components = 1;
    instances = mat_b_h;
    patches = mat_a_w;
  } break;
  default: {
    TRINNITY_ERROR("Not implemented");
  } break;
}
if(shaderProgramID == 0) {
  TRINNITY_ERROR("Shader compilation failed");
}
switch (tex_components) {
  case 1: {
    tex_internal_format = GL_R32F;
    tex_format = GL_RED;
  } break;
  case 2: {
    tex_internal_format = GL_RG32F;
    tex_format = GL_RG;
  } break;
  case 3: {
    tex_internal_format = GL_RGB32F;
    tex_format = GL_RGB;
```

```
  } break;
  case 4: {
    tex_internal_format = GL_RGBA32F;
    tex_format = GL_RGBA;
  } break;
  default: {
    TRINNITY_ERROR("Not implemented - this tex_components size
                    shouldn't be possible");
  } break;
}

#if defined(CHECK_GL_ERRORS)
triNNity::opengl::glGetErrorString("After shader compilation");
#endif

// Create VAO
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

// Create input VBO and vertex format
GLuint vbo;
glGenBuffers(1, &vbo);

// Create transform feedback buffer
int tbo_size = patches*instances*chunks_per_row;
GLuint tbo;
glGenBuffers(1, &tbo);
glBindBuffer(GL_TRANSFORM_FEEDBACK_BUFFER, tbo);
glBufferData(GL_TRANSFORM_FEEDBACK_BUFFER,
             tbo_size*sizeof(float), 0, GL_STATIC_READ);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, tbo);
#if defined(CHECK_GL_ERRORS)
triNNity::opengl::glGetErrorString("After binding tbo");
#endif

// For Intel HD 5000 result is 16384, which should
// be 16384*16384 (2048*8) on a MacBook Air
int max_texture_size = 0;
```

```
int max_texture_width = 0;
glGetIntegerv(GL_MAX_TEXTURE_SIZE, &max_texture_width);
max_texture_width /=8;
max_texture_size = max_texture_width*max_texture_width;
// check mat_b fits in texture memory
// write code to chunk otherwise
bool needs_chunks = false;
if (mat_b_h*mat_b_w > max_texture_size) {
  needs_chunks = true; // handle this!
  TRINNITY_ERROR("Not implemented - chunking");
  return;
}

int mat_b_size = mat_b_w*mat_b_h/tex_components;
int tex_w = (mat_b_size<max_texture_width) ? mat_b_size :
                                             max_texture_width;
int tex_h = mat_b_size/tex_w;
int remainder = (mat_b_size%tex_w)?tex_w-(mat_b_size%tex_w):0;
if(remainder) {
  tex_h++;
}

float* b_copy = new float[mat_b_size+remainder];
// need to do a memcpy as b is const restricted
memcpy(b_copy, b, mat_b_size*sizeof(float));

GLint input_values = glGetAttribLocation(shaderProgramID,
                                         "input_values");
GLint shader_mat_b_w = glGetUniformLocation(shaderProgramID,
                                            "mat_b_w");
glProgramUniform1i(shaderProgramID, shader_mat_b_w, mat_b_w);
GLint shader_mat_b_h = glGetUniformLocation(shaderProgramID,
                                            "mat_b_h");
glProgramUniform1i(shaderProgramID, shader_mat_b_h, mat_b_h);
GLint shader_tex_w = glGetUniformLocation(shaderProgramID,
                                          "tex_w");
glProgramUniform1i(shaderProgramID, shader_tex_w, tex_w);
GLint shader_tex_h = glGetUniformLocation(shaderProgramID,
                                          "tex_h");
```

```
glProgramUniform1i(shaderProgramID, shader_tex_h, tex_h);
#if defined(CHECK_GL_ERRORS)
triNNity::opengl::glGetErrorString("After setting uniforms");
#endif

// Create texture
GLuint tex;
glGenTextures(1, &tex);
glBindTexture(texture_type, tex);
//GL_NEAREST ensures actual kernal values are used,
// not linearly interpolated/weighted ones
glTexParameteri(texture_type, GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);
glTexParameteri(texture_type, GL_TEXTURE_MAG_FILTER,
                GL_NEAREST);

// load b into texture memory
#if defined(CHECK_GL_ERRORS)
triNNity::opengl::glGetErrorString("Before binding texture");
#endif
glTexImage2D(texture_type, 0, tex_internal_format, tex_w,
             tex_h, 0, tex_format, GL_FLOAT, b);
delete[] b_copy;
#if defined(CHECK_GL_ERRORS)
triNNity::opengl::glGetErrorString("After binding texture");
#endif

// Load a into VBO
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, mat_a_h*mat_a_w*sizeof(float), a,
             GL_STATIC_DRAW);
#if defined(CHECK_GL_ERRORS)
triNNity::opengl::glGetErrorString("After binding vbo");
#endif

glEnableVertexAttribArray(input_values);
glVertexAttribPointer(input_values, chunk_size, GL_FLOAT,
                      GL_FALSE, 0, NULL);
glEnable(GL_RASTERIZER_DISCARD);
```

```
// Perform feedback transform
glBeginTransformFeedback(GL_POINTS);

// Instances allow mat_a to be worked on repeatedly
glDrawArraysInstanced(GL_POINTS, 0, (tbo_size/instances),
   instances);
glEndTransformFeedback();
glDisable(GL_RASTERIZER_DISCARD);
glFlush();
#if defined(CHECK_GL_ERRORS)
triNNity::opengl::glGetErrorString("After drawing");
#endif

// cleanup early to avoid GL_OUT_OF_MEMORY when mapping
   feedback
glDeleteBuffers(1, &vbo);
glDeleteTextures(1, &tex);
glDeleteVertexArrays(1, &vao);
glDeleteProgram(shaderProgramID);

GLfloat* feedback = reinterpret_cast<GLfloat*>(glMapBufferRange
                    (GL_TRANSFORM_FEEDBACK_BUFFER, 0,
                     tbo_size*sizeof(GLfloat),
                     GL_MAP_READ_BIT | GL_MAP_WRITE_BIT));

#if defined(CHECK_GL_ERRORS)
triNNity::opengl::glGetErrorString("After mapping feedback");
#endif
if(feedback == NULL) {
  TRINNITY_ERROR("Transform feedback buffer is empty");
}

for(unsigned l=0; l < patches*instances; l+=1) {
  feedback[l] = feedback[l*chunks_per_row];
  for(unsigned k=1; k < chunks_per_row; k+=1) { // for each
     chunk
    feedback[l] += feedback[l*chunks_per_row + k];
  }
```

```
 }

 for (unsigned i = 0; i < patches; i+=1) { // for each row of a
   for(unsigned j = 0; j < instances; j+=1) { // for each column
       of b
     switch (accum) {
       case GEMM_ACCUMULATE: {
         c[i*instances + j] += feedback[j*patches+i];
       } break;
       case GEMM_SUBTRACT_ACCUMULATE: {
         c[i*instances + j] -= feedback[j*patches+i];
       } break;
       case GEMM_NO_ACCUMULATE: {
         c[i*instances + j] = feedback[j*patches+i];
       } break;
     }
   }
 }

 glUnmapBuffer(GL_TRANSFORM_FEEDBACK_BUFFER);
 glDeleteBuffers(1, &tbo);
}
#endif // #if defined(TRINITY_USE_OPENGL_GEMM)
```

Listing A1.2: Collection of shader programmes developed

```
namespace triNNity {

namespace opengl {

// gl_VertexID to get vertex index

#if defined(BENCHMARK_OPENGLES)
#define NORMALISED_X(x) "float(2*" x "+1)/float(2*tex_w)"
#define NORMALISED_Y(y) "float(2*" y "+1)/float(2*tex_h)"
#define VERSION "#version 310 es\n"
```

```
#define SAMPLER "uniform sampler2D tex;\n"
#define TEXTURE "texture"
#else
#define NORMALISED_X(x) x
#define NORMALISED_Y(y) y
#define VERSION "#version 400\n"
#define SAMPLER "uniform sampler2DRect tex;\n"
#define TEXTURE "texture"
#endif

#define UNIFORMS_OUT   "uniform int mat_b_w;\n \
                        uniform int mat_b_h;\n \
                        uniform int tex_w;\n \
                        uniform int tex_h;\n \
                        out float out_value;\n"

// A_B
#define BX(j) "((offset+(" #j "*mat_b_w))%tex_w)"
#define BY(k) "((offset+(" #k "*mat_b_w))/tex_w)"
static const char* gemmA_B_vec1 =
  VERSION
  "\n"
  SAMPLER
  "in float input_values;\n"
  UNIFORMS_OUT
  "\n"
  "void main()\n"
  "{\n"
  "   int offset = mat_b_w*((gl_VertexID*1)%mat_b_h) +
     gl_InstanceID;"
  "   float t = " TEXTURE "(tex, vec2(" NORMALISED_X(BX(0)) "
                                     ," NORMALISED_Y(BY(0)) "))
                                        [0];\n"
  "   out_value = input_values*t;\n"
  "}";

static const char* gemmA_B_vec2 =
  VERSION
  "\n"
```

```
  SAMPLER
  "in vec2 input_values;\n"
  UNIFORMS_OUT
  "\n"
  "void main()\n"
  "{\n"
  "    int offset = mat_b_w*((gl_VertexID*2)%mat_b_h) +
     gl_InstanceID;"
  "    float tex0 = " TEXTURE "(tex, vec2(" NORMALISED_X(BX(0)) "
                                          ," NORMALISED_Y(BY(0)) "
                                            ))[0];"
  "    float tex1 = " TEXTURE "(tex, vec2(" NORMALISED_X(BX(1)) "
                                          ," NORMALISED_Y(BY(1)) "
                                            ))[0];"
  "    vec2 t = vec2(tex0, tex1);\n"
  "    out_value = dot(input_values, t);\n"
  "}";

static const char* gemmA_B_vec3 =
  VERSION
  "\n"
  SAMPLER
  "in vec3 input_values;\n"
  UNIFORMS_OUT
  "\n"
  "void main()\n"
  "{\n"
  "    int offset = mat_b_w*((gl_VertexID*3)%mat_b_h) +
     gl_InstanceID;"
  "    float tex0 = " TEXTURE "(tex, vec2(" NORMALISED_X(BX(0)) "
                                          ," NORMALISED_Y(BY(0)) "
                                            ))[0];"
  "    float tex1 = " TEXTURE "(tex, vec2(" NORMALISED_X(BX(1)) "
                                          ," NORMALISED_Y(BY(1)) "
                                            ))[0];"
  "    float tex2 = " TEXTURE "(tex, vec2(" NORMALISED_X(BX(2)) "
                                          ," NORMALISED_Y(BY(2)) "
                                            ))[0];"
  "    vec3 t = vec3(tex0, tex1, tex2);\n"
```

```
  "   out_value = dot(input_values, t);\n"
  "}";

static const char* gemmA_B_vec4 =
  VERSION
  "\n"
  SAMPLER
  "in vec4 input_values;\n"
  UNIFORMS_OUT
  "\n"
  "void main()\n"
  "{\n"
  "   int offset = mat_b_w*((gl_VertexID*4)%mat_b_h) +
     gl_InstanceID;"
  "   float tex0 = " TEXTURE "(tex, vec2(" NORMALISED_X(BX(0)) "
                                        ," NORMALISED_Y(BY(0)) "
                                          ))[0];"
  "   float tex1 = " TEXTURE "(tex, vec2(" NORMALISED_X(BX(1)) "
                                        ," NORMALISED_Y(BY(1)) "
                                          ))[0];"
  "   float tex2 = " TEXTURE "(tex, vec2(" NORMALISED_X(BX(2)) "
                                        ," NORMALISED_Y(BY(2)) "
                                          ))[0];"
  "   float tex3 = " TEXTURE "(tex, vec2(" NORMALISED_X(BX(3)) "
                                        ," NORMALISED_Y(BY(3)) "
                                          ))[0];"
  "   vec4 t = vec4(tex0, tex1, tex2, tex3);\n"
  "   out_value = dot(input_values, t);\n"
  "}";

// A_BT
#define BTX "(offset%tex_w)"
#define BTY "(offset/tex_w)"
static const char* gemmA_BT_vec1 =
  VERSION
  "\n"
  SAMPLER
  "in float input_values;\n"
  UNIFORMS_OUT
```

```
  "\n"
  "void main()\n"
  "{\n"
  "   int offset = ((mat_b_w/1)*gl_InstanceID) +
                   (gl_VertexID%(mat_b_w/1));"
  "   float t = " TEXTURE "(tex, vec2(" NORMALISED_X(BTX) ",
                                        " NORMALISED_Y(BTY) "))
                                          [0];\n"
  "   out_value = input_values*t;\n"
  "}";

static const char* gemmA_BT_vec2 =
  VERSION
  "\n"
  SAMPLER
  "in vec2 input_values;\n"
  UNIFORMS_OUT
  "\n"
  "void main()\n"
  "{\n"
  "   int offset = ((mat_b_w/2)*gl_InstanceID) +
                   (gl_VertexID%(mat_b_w/2));"
  "   vec2 t = vec2(" TEXTURE "(tex, vec2(" NORMALISED_X(BTX) ",
                                           " NORMALISED_Y(BTY) ")
                                            ));"
  "   out_value = dot(input_values, t);\n"
  "}";

static const char* gemmA_BT_vec3 =
  VERSION
  "\n"
  SAMPLER
  "in vec3 input_values;\n"
  UNIFORMS_OUT
  "\n"
  "void main()\n"
  "{\n"
  "   int offset = ((mat_b_w/3)*gl_InstanceID) +
                   (gl_VertexID%(mat_b_w/3));"
```

```
  "    vec3 t = vec3(" TEXTURE "(tex, vec2(" NORMALISED_X(BTX) ",
                                            " NORMALISED_Y(BTY) ")
                                              ));"
  "    out_value = dot(input_values, t);\n"
  "}";

static const char* gemmA_BT_vec4 =
  VERSION
  "\n"
  SAMPLER
  "in vec4 input_values;\n"
  UNIFORMS_OUT
  "\n"
  "void main()\n"
  "{\n"
  "    int offset = ((mat_b_w/4)*gl_InstanceID) +
                    (gl_VertexID%(mat_b_w/4));"
  "    vec4 t = vec4(" TEXTURE "(tex, vec2(" NORMALISED_X(BTX) ",
                                            " NORMALISED_Y(BTY) ")
                                              ));"
  "    out_value = dot(input_values, t);\n"
  "}";

static const char* dummy_frag_shader =
  VERSION
  "out int o_value;\n"
  "void main()\n"
  "{\n"
  "    o_value = 0;\n"
  "}";
}
}
```

Listing A1.3: Some OpenGL, EGL, and GLFW helper functions developed

```
#if defined(BENCHMARK_OPENGLES)
```

```
#include <EGL/egl.h>
#define GL_GLEXT_PROTOTYPES
#include <GLES3/gl31.h>
#else
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#endif

namespace triNNity {

namespace opengl {
#if defined(BENCHMARK_OPENGLES)
static const EGLint configAttribs[] = {
  EGL_SURFACE_TYPE, EGL_PBUFFER_BIT,
  EGL_BLUE_SIZE, 8,
  EGL_GREEN_SIZE, 8,
  EGL_RED_SIZE, 8,
  EGL_DEPTH_SIZE, 8,

  EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
  EGL_NONE
};

// Width and height of the desired framebuffer
static const EGLint pbufferAttribs[] = {
  EGL_WIDTH, 800,
  EGL_HEIGHT, 600,
  EGL_NONE,
};

static const EGLint contextAttribs[] = {
  EGL_CONTEXT_CLIENT_VERSION, 2,
  EGL_NONE
};

static const char* eglGetErrorStr(){
  switch(eglGetError()){
  case EGL_SUCCESS:
    return "The last function succeeded without error.";
```

```
case EGL_NOT_INITIALIZED:
  return "EGL is not initialized, or could not be initialized,
          for the specified EGL display connection.";
case EGL_BAD_ACCESS:
  return "EGL cannot access a requested resource (for example
          a context is bound in another thread).";
case EGL_BAD_ALLOC:
  return "EGL failed to allocate resources for the requested
          operation.";
case EGL_BAD_ATTRIBUTE:
  return "An unrecognized attribute or attribute value was
          passed in the attribute list.";
case EGL_BAD_CONTEXT:
  return "An EGLContext argument does not name a valid EGL
          rendering context.";
case EGL_BAD_CONFIG:
  return "An EGLConfig argument does not name a valid EGL
          frame buffer configuration.";
case EGL_BAD_CURRENT_SURFACE:
  return "The current surface of the calling thread is a
          window, pixel buffer or pixmap that is no longer
             valid.";
case EGL_BAD_DISPLAY:
  return "An EGLDisplay argument does not name a valid EGL
          display connection.";
case EGL_BAD_SURFACE:
  return "An EGLSurface argument does not name a valid surface
          (window, pixel buffer or pixmap) configured for
          GL rendering.";
case EGL_BAD_MATCH:
  return "Arguments are inconsistent (for example, a
          valid context requires buffers not supplied
          by a valid surface).";
case EGL_BAD_PARAMETER:
  return "One or more argument values are invalid.";
case EGL_BAD_NATIVE_PIXMAP:
  return "A NativePixmapType argument does not refer
          to a valid native pixmap.";
case EGL_BAD_NATIVE_WINDOW:
```

```
    return "A NativeWindowType argument does not refer
            to a valid native window.";
  case EGL_CONTEXT_LOST:
    return "A power management event has occurred. The
            application must destroy all contexts and
            reinitialise OpenGL ES state and objects to
            continue rendering.";
  default: break;
  }
  return "Unknown error!";
}

int initEGL(EGLDisplay* display, EGLSurface* surface,
            EGLContext* context) {

  int major, minor;
  int desiredWidth, desiredHeight;
  GLint posLoc, colorLoc, result;

  if((*display = eglGetDisplay(EGL_DEFAULT_DISPLAY))
     == EGL_NO_DISPLAY)
  {
    fprintf(stderr, "Failed to get EGL display! Error: %s\n",
            eglGetErrorStr);
    return EXIT_FAILURE;
  }

  if(eglInitialize(*display, &major, &minor) == EGL_FALSE)
  {
    fprintf(stderr, "Failed to get EGL version! Error: %s\n",
            eglGetErrorStr);
    eglTerminate(*display);
    return EXIT_FAILURE;
  }

  printf("Initialized EGL version: %d.%d\n", major, minor);

  EGLint numConfigs;
  EGLConfig config;
```

```
if(!eglChooseConfig(*display, configAttribs, &config, 1,
                    &numConfigs))
{
  fprintf(stderr, "Failed to get EGL config! Error: %s\n",
          eglGetErrorStr);
  eglTerminate(*display);
  return EXIT_FAILURE;
}

*surface = eglCreatePbufferSurface(*display, config,
                                   pbufferAttribs);
if(*surface == EGL_NO_SURFACE){
  fprintf(stderr, "Failed to create EGL surface! Error: %s\n",
          eglGetErrorStr);
  eglTerminate(*display);
  return EXIT_FAILURE;
}

eglBindAPI(EGL_OPENGL_API);

*context = eglCreateContext(*display, config, EGL_NO_CONTEXT,
                            contextAttribs);
if(*context == EGL_NO_CONTEXT) {
  fprintf(stderr, "Failed to create EGL context! Error: %s\n",
          eglGetErrorStr);
  eglDestroySurface(*display, *surface);
  eglTerminate(*display);
  return EXIT_FAILURE;
}

eglMakeCurrent(*display, *surface, *surface, *context);

// The desired width and height is defined inside of
   pbufferAttribs
// Check top of this file for EGL_WIDTH and EGL_HEIGHT
desiredWidth = pbufferAttribs[1]; // 800
desiredHeight = pbufferAttribs[3]; // 600

// Set GL Viewport size, always needed!
```

```
  glViewport(0, 0, desiredWidth, desiredHeight);

  // Get GL Viewport size and test if it is correct.
  // The following piece of code checks if the gl functions
  // are working as intended!
  GLint viewport[4];
  glGetIntegerv(GL_VIEWPORT, viewport);

  // viewport[2] and viewport[3] are viewport width and height
  // respectively
  printf("GL Viewport size: %dx%d\n", viewport[2], viewport[3]);

  // Test if the desired width and height match the one returned
  // by glGetIntegerv
  if(desiredWidth!=viewport[2] || desiredHeight!=viewport[3])
  {
    fprintf(stderr, "Error! The glViewport/glGetIntegerv are not
                     working! EGL might be faulty!\n");
  }

  return 0;
}
#else
int initGLFW() {
    // start GL context and O/S window using the GLFW helper
       library
  if (!glfwInit()) {
    fprintf(stderr, "ERROR: could not start GLFW3\n");
    return 1;
  }

  // for some reason GLFW_FALSE was not in scope on pi?
  glfwWindowHint(GLFW_VISIBLE, 0);
  // the following four are macOS specific
  // code from http://antongerdelan.net/opengl/hellotriangle.
     html
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
  glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
```

```
  glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
  GLFWwindow* window = glfwCreateWindow(640, 480, "", NULL,
                                        NULL);
  if (!window) {
    fprintf(stderr, "ERROR: couldn't open window with GLFW3\n");
    glfwTerminate();
    return 1;
  }
  glfwMakeContextCurrent(window);
  return 0;
}
#endif

// Shader Functions

GLuint addShader(GLuint shaderProgram, const char* shaderText,
                 GLenum shaderType)
{
  // create a shader object
  GLuint shaderObj = glCreateShader(shaderType);

  if (shaderObj == 0) {
    fprintf(stderr, "Error creating shader type %d\n",
            shaderType);
    return 0;
  }
  // Bind the source code to the shader,
  // this happens before compilation
  glShaderSource(shaderObj, 1, &shaderText, NULL);
  // compile the shader and check for errors
  glCompileShader(shaderObj);
  GLint success;
  // check for shader related errors using glGetShaderiv
  glGetShaderiv(shaderObj, GL_COMPILE_STATUS, &success);
  if (!success) {
    GLchar infoLog[1024];
    glGetShaderInfoLog(shaderObj, 1024, NULL, infoLog);
    fprintf(stderr, "Error compiling shader type %d: '%s'\n",
            shaderType, infoLog);
```

```
    return 0;
  }
  // Attach the compiled shader object to the program object
  glAttachShader(shaderProgram, shaderObj);
  return shaderObj;
}

GLuint compileShaders(const char* shaderSrc)
{
  //Start the process of setting up our shaders by creating
  // a program ID
  //Note: we will link all the shaders together into this ID
  GLuint shaderProgramID = glCreateProgram();
  if (shaderProgramID == 0) {
    fprintf(stderr, "Error creating shader program\n");
    return 0;
  }

  // Create two shader objects, one for the vertex, and one for
  // the fragment shader
  GLuint shaderObj = addShader(shaderProgramID, shaderSrc,
                               GL_VERTEX_SHADER);
  if(shaderObj == 0) {
    fprintf(stderr, "Error adding shader program\n");
    return 0;
  }

  #if defined(BENCHMARK_OPENGLES)
  GLuint shaderObjFrag = addShader(shaderProgramID,
                          triNNity::opengl::dummy_frag_shader,
                          GL_FRAGMENT_SHADER);
  if(shaderObjFrag == 0) {
    fprintf(stderr, "Error adding frag shader program\n");
    return 0;
  }
  #endif

  const GLchar* feedbackVaryings[] = { "out_value" };
  glTransformFeedbackVaryings(shaderProgramID, 1,
```

```
                                feedbackVaryings,
                                GL_INTERLEAVED_ATTRIBS);

  GLint success = 0;
  GLchar errorLog[1024] = { 0 };

  // After compiling all shader objects and attaching them
  // to the program, we can finally link it
  glLinkProgram(shaderProgramID);
  // check for program related errors using glGetProgramiv
  glGetProgramiv(shaderProgramID, GL_LINK_STATUS, &success);
  if (success == 0) {
    glGetProgramInfoLog(shaderProgramID, sizeof(errorLog),
                        NULL, errorLog);
    fprintf(stderr, "Error linking shader program: '%s'\n",
            errorLog);
    return 0;
  }

  glDeleteShader(shaderObj);
  #if defined(BENCHMARK_OPENGLES)
  glDeleteShader(shaderObjFrag);
  #endif

  // Finally, use the linked shader program
  // Note: this program will stay in effect for all draw calls
  // until you replace it with
  // another or explicitly disable its use
  glUseProgram(shaderProgramID);
  return shaderProgramID;
}

void glGetErrorString(const char* description) {
  int err = glGetError();
  while(err != GL_NO_ERROR) {
    switch(err) {
      case GL_INVALID_ENUM: {
        fprintf(stderr, "%s, GL_INVALID_ENUM\n", description);
      } break;
```

```
      case GL_INVALID_VALUE: {
        fprintf(stderr, "%s, GL_INVALID_VALUE\n", description);
      } break;
      case GL_INVALID_OPERATION: {
        fprintf(stderr, "%s, GL_INVALID_OPERATION\n",
                description);
      } break;
      case GL_OUT_OF_MEMORY: {
        fprintf(stderr, "%s, GL_OUT_OF_MEMORY\n", description);
      } break;
      default: {
        fprintf(stderr, "%s, Unknow error\n", description);
      } break;
    }
    err = glGetError();
  }
}



}

}
```