

**NANYANG  
TECHNOLOGICAL  
UNIVERSITY**

## **Image Recognition on an Embedded GPU Board**

**Submitted by: Thomas Stephen Felix  
Matriculation Number: U1722308D**

**Supervisor: A/P Vun Chan Hua, Nicholas**

School of Computer Science and Engineering

A final year project report presented to the Nanyang Technological University  
in partial fulfilment of the requirements of the degree of  
Bachelor of Engineering

**2021**

## Table of Contents

Abstract .....	iv
List of Figures .....	v
List of Tables .....	vii
Chapter 1 Introduction.....	1
1.1 Motivations .....	1
1.2 Objectives and Scope .....	2
Chapter 2 Literature Review.....	4
2.1 Computer vision and image recognition .....	4
2.1.1 Convolution layer.....	6
2.1.2 Sub-Sampling Layer .....	7
2.1.3 Fully Connected Layer.....	7
2.1.4 Forward & Backward Propagation .....	8
2.2 Embedded Board.....	8
2.2.1 NVIDIA .....	8
2.2.2 Jetson Nano .....	9
2.2.3 Add-On capabilities .....	10
2.3 GPU.....	11
2.3.1 Architecture.....	11
2.3.1 CUDA C.....	13
Chapter 3 Materials & Methodology .....	22
3.1 Jetson nano board set up. ....	22
3.1.1 Interfacing .....	23
3.2 CUDA Implementation .....	26

3.2.1	Convolution using CPU (Serial) .....	26
3.2.2	Convolution using GPU (Parallel) .....	27
3.2.3	Convolution in Parallel using Shared Memory.....	29
3.2.4	Convolution in Parallel using Constant Memory.....	32
3.2.5	Convolution in Parallel using Shared & Constant Memory. ....	33
3.2.6	Convolution in Parallel using Texture Memory. ....	33
3.2.7	Convolution in Parallel using 2D Texture Memory. ....	34
3.3	OpenCV Implementation .....	34
Chapter 4.....		36
4.1	CUDA Results .....	36
4.1.1	Effect of kernel size on convolution algorithms .....	36
4.1.2	Effect of image size on convolution algorithms .....	43
4.1.3	Effect of block size on convolution algorithms .....	45
4.2	OpenCV Results.....	46
Chapter 5.....		48
5.1	Conclusions.....	48
5.2	Recommendation in Future Work.....	48
References.....		49
Appendix A.....		51
Appendix B .....		52
Appendix C .....		53
Appendix D.....		54
Appendix E .....		55
Appendix F.....		56

# Abstract

The onset of artificial intelligence (AI) led to an inconceivable boom in the technology industry. Furthermore, the dawn of AI and machine learning led to technological giants pushing for systems capable of handling highly intensive mathematical calculations to accommodate for possible machine learning implementations. Today, we have mobile chips with machine learning engines optimized especially for machine learning algorithms. Such advancements have revolutionized the face of the industry. In my project, I intend to understand the various aspects of an embedded system with an on-board graphical processing unit (GPU).

One such application of artificial intelligence is computer vision, a machine learning technique used to provide machines with cognitive abilities. A wide variety of techniques and methods have been explored and examined to improve this process. However, the most relevant and commonly used technique is convolutional neural networks (CNNs). My project takes advantage of the on-board GPU with the use of CUDA C to improve the performance of convolutional kernels used in CNNs. Therefore, improving the efficiency of image recognition algorithms.

# List of Figures

Figure 1 - Mapping features from input space to feature space.....	5
Figure 2 - Image recognition architecture.....	5
Figure 3 - Convolution operation breakdown.....	6
Figure 4 - Max Pooling & Average Pooling.....	7
Figure 5 - Jetson Nano Developer Kit [12].....	9
Figure 6 - Jtop overview page.....	10
Figure 7- Jtop GPU Usage - .....	11
Figure 8 - Jtop CPU usage .....	11
Figure 9 - GPU specifications.....	11
Figure 10 - Maxwell Architecture SMM .....	12
Figure 11 - CUDA C grid .....	13
Figure 12 - Simple CUDA kernel call .....	15
Figure 13 - Thread & Block implementation.....	16
Figure 14 - Memory allocation .....	17
Figure 15 - Memory hierarchy .....	19
Figure 16- Etcher Interface .....	22
Figure 17 - set IPv4 properties.....	23
Figure 18 - choose network range.....	24
Figure 19 - choose network card.....	24
Figure 20 - Run DHCP Server .....	24
Figure 21 - Get IP address given by DHCP .....	24
Figure 22 - SSH into machine.....	25
Figure 23 - ifconfig for Jetson Nano with WIFI.....	25
Figure 24 - Serial convolutional implementation .....	26
Figure 25 - Parallelized calls for convolution.....	28
Figure 26 - Code snippet for getting image pixel coordinate. ....	28
Figure 27 - Shared memory implementation .....	29
Figure 28 - Shared memory implementation kernel .....	30
Figure 29 - Shared memory working.....	31
Figure 30- Constant memory implementation .....	32
Figure 31 - Texture memory implementation.....	33
Figure 32 - 2D Texture memory implementation .....	34
Figure 33 - OpenCV serial convolution implementation.....	34
Figure 34 - OpenCV Parallel convolution implementation .....	35
Figure 35 – Kernel size comparison for serial & parallel.....	37
Figure 36 – Kernel size comparison for different parallel methods .....	37
Figure 37 - Kernel size comparison for different parallel methods (4096x4096) .....	38
Figure 38 - Parallel implementation comparisons (4096x4096) with adjusted block width for shared memory implementation.....	39
Figure 39 - New shared memory working .....	40
Figure 40 - Parallel implementation comparisons (4096x4096) adjusted shared memory implementation. ....	41
Figure 41 - Image size comparison for serial & parallel .....	43

Figure 42 - Image size comparison for different parallel methods .....	44
Figure 43 - Block size comparison for different parallel methods .....	45
Figure 44 - OpenCV CUDA convolution .....	47

# List of Tables

Table 1 - Calculation of thread ID .....	15
Table 2 - Function Type Qualifiers.....	18
Table 3 - Variable Type Qualifiers .....	18
Table 4 - Parallel vs Shared memory implementation.....	31
Table 5 - Execution time comparison based on kernel size.....	36
Table 6 - Execution time comparison based on image size. ....	43
Table 7- Execution time comparison based on block size. ....	45
Table 8 - Serial Vs Parallel OpenCV convolution comparison .....	46
Table 9 - Table with execution time for different convolution implementation styles ....	51
Table 10 - Table with speedup of different convolution implementation styles .....	52
Table 11 - Table with execution time for shared memory implementations with updated block size value.....	53
Table 12 - Table with Speedup for shared memory implementations with updated block size value.....	54
Table 13 - Table with execution time for share memory implementations with no overlapping between blocks.....	55
Table 14 - Table with Speedup for share memory implementations with no overlapping between blocks.....	56

# Chapter 1

## Introduction

### 1.1 Motivations

The rising popularity of image processing and image recognition algorithms has left a need for implementations of more efficient algorithms. CUDA (Compute Unified Device Architecture) [7] as a heterogeneous architecture of CPU & GPU (Graphical Processing Unit) has made a wide impact on a multitude of industries and has consistently outperformed various other systems and implementations. Its ability to implement and control multi-threading in the GPGPU (General Purpose Graphical Processing Unit) has been used to speedup otherwise slower image processing algorithm.

In [1], Sajin Choi & Kwangyeob Lee implement a CUDA-based convolutional network from scratch. Their implementation of convolutional and pooling kernels involved the assignment of kernel to individual blocks which would then generate a corresponding output feature map in parallel. They employed parallel threading for computation of the fully connected layer and weight update to improve the throughput of the system. They make use of the NVIDIA CUDA architecture to introduce parallelization and hence improve the throughput of a CNN by approximately 3.5 times when compared to a CPU core and 2.5 times when compared to an OpenMP implementation. Thus, highlighting the computational power and efficiency obtained when using CUDA C.

CUDA C, has been used extensively especially in image processing. [2] & [6], shows how the use of CUDA C to implement a CT image reconstruction and transmission



algorithm respectively has shown considerable improvement in throughput. The speed up introduced highlight the computational power possible when using CUDA C. This is further established in [3]. A comparison being made between GPU, FPGA & MATLAB implementations of convolutions establishes the dominance of CUDA C's parallel threading capabilities. Nevertheless, the implementation of CUDA C is based on spatial separable convolutional kernels which consists of a small subset of possible kernels. Therefore, it may prove beneficial to compare a more generic implementation of CUDA C based convolution to obtain more significant results.

In [4], several CUDA C convolutional algorithms are used. Of these several algorithms, convolution done with the help of multiple blocks seems to produce the best performance. The implementation takes advantage of the max number of blocks along one direction of the CUDA grid which is 65535 and uses a 2D grid implementation to calculate the values of a pixel per block. This is done by using the shared memory within a block. The shared memory will be assigned multiplied values provided by threads and then giving one thread the responsibility of summing up all the values within a block. While the experimental results indicate significant improvement, the results obtained were for an image size of 31x31 and a kernel size of 16x16. These sizes may not provide accurate representations of images with sizes up to 2048x2048. Furthermore, this limits the size of the kernel to 32x32, which is the limit on the number of threads per block.

## 1.2 Objectives and Scope

I aim to implement methodologies using the CUDA architecture that improve the performance of the most used image recognition algorithm i.e., convolution. This is done to improve the convolution operation with the means of threading on a GPU. This is done by making use of the resources on board such as constant, shared and texture memory. Furthermore, a detailed comparison will be done with regards to the effects of

kernel size, image size and block size on computational cost.

In addition, I aim to compare my CUDA implementations with an open-source library known as OpenCV. OpenCV is used for image processing as well as implementation of image recognition models. We use OpenCV's CUDA interface to compare our results with a pre-existing library to gauge the performance improvement attained. We will discuss other interfaces that use CUDA to improve performances as well.

# Chapter 2

## Literature Review

### 2.1 Computer vision and image recognition

Computer vision (CV) refers to the methodologies used to allow a machine to see [10]. Ever since the boom of Artificial Intelligence, computer vision has been a major field of interest with multiple applications across fields such as robotics and signal processing. The objective is to better understand the content of images and videos, then use this information to perform human like tasks such as identifying human emotions.

Early concepts of computer vision included image enhancements and edge processing. With the onset of neural networks, the field of computer vision was greatly impacted. The use of Neural Networks allowed for more information to be extracted from images.

Image and Object recognition are two such applications of computer vision that experienced great advancements with the onset of neural networks. The working of neural networks involves the generation of feature vectors. These feature vectors help the computer interpret various features of the image which are then used to classify images. We observe that when an  $n$ -dimensional feature vector is mapped to feature space, images with similar features are grouped together which allows for better classification of images.

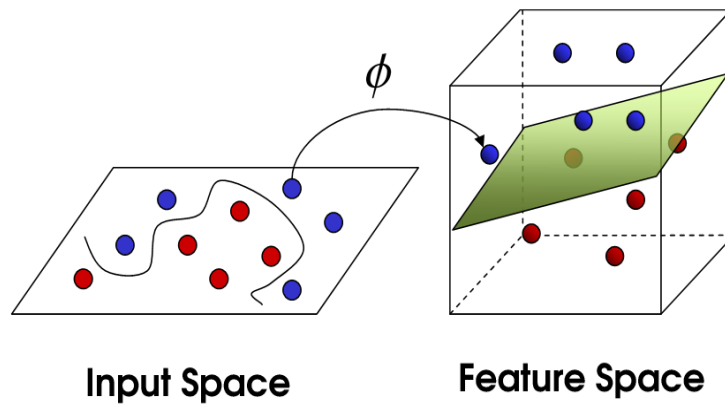


Figure 1 - Mapping features from input space to feature space.

The use of deep neural networks allowed for the learning of hidden features observed in data. These hidden features helped a machine to better distinguish between classes of images. However, if we were to use a simple deep neural network in order to implement an image recognition algorithm, each pixel would be used as an input and the information of the neighboring pixels would be lost [11].

To overcome the shortcomings of a typical deep neural network, multiple convolutional kernels were used in groups and in succession on images to extract or summarize information. This in turn led to the development of Convolutional Neural Networks (CNN). The image below shows how image recognition is typically implemented with use of a CNN.

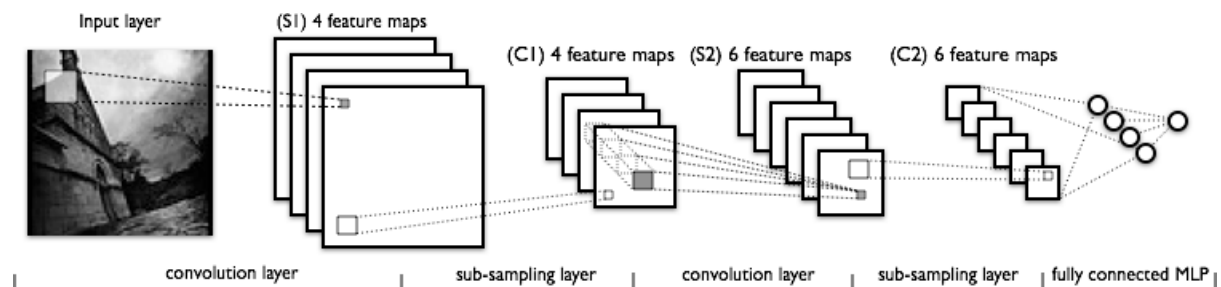


Figure 2 - Image recognition architecture

## 2.1.1 Convolution layer

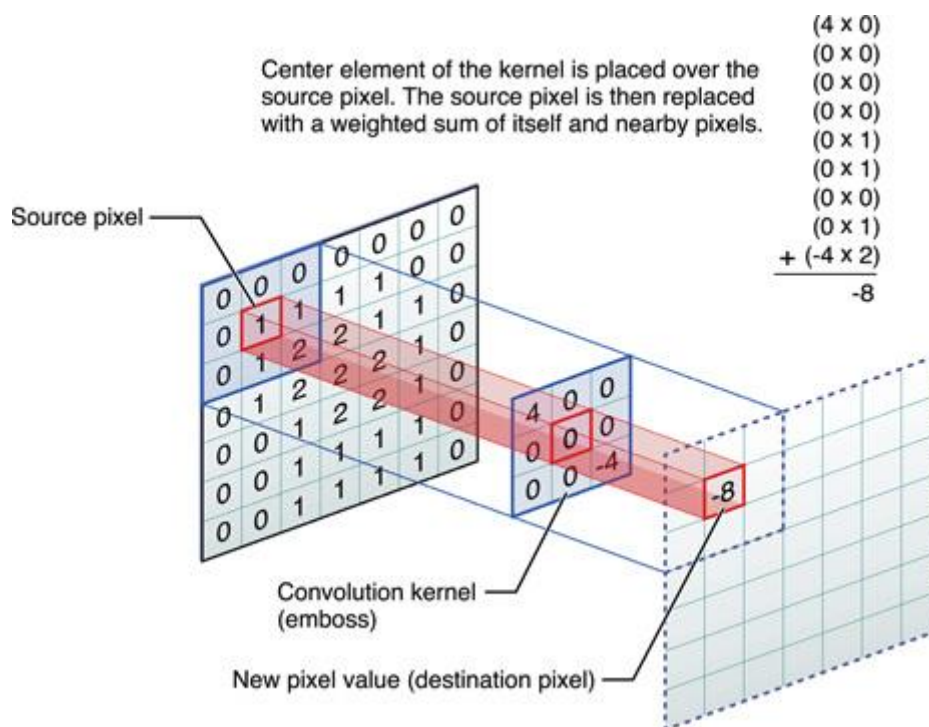


Figure 3 - Convolution operation breakdown

The first step of CNNs involves the use  $N \times M$  kernels to create many small pieces called features [11]. These kernels assign different weights to values around the point of interest. The values of the image are then multiplied with those of the kernel to generate summarized information around an area of interest. The figure above helps visualize the process of a convolutional kernel. The point of interest refers to the source pixel in the image.

Convolutional kernels can be of various sizes depending on the size of the output required and the stride of the kernel (stride refers to the distance between current point of interest to the next). To attain complete information from the image, generally images are GENERALLY zero-padded (adding zeros to the border of images) to perform convolution along the borders of images as well.

Convolution is the most important and time-consuming aspect of a CNN algorithm. It involves a series of multiplications and additions for each pixel. This makes it quite burdensome on a simple CPU architecture. Therefore, we make use of a GPGPU to improve the computation time of such algorithms with the help of multi-threading.

### 2.1.2 Sub-Sampling Layer

The sub-sampling layer helps limit the information by reducing the number of pixels and highlighting the most relevant information. In the sub-sampling layer, we either make use of max-pooling or average-pooling.

- **Max-Pooling:** Maximum value from a window is used to represent all the information within the window.
- **Average-Pooling:** Average value of all elements within a window is used to represent the information within the window.

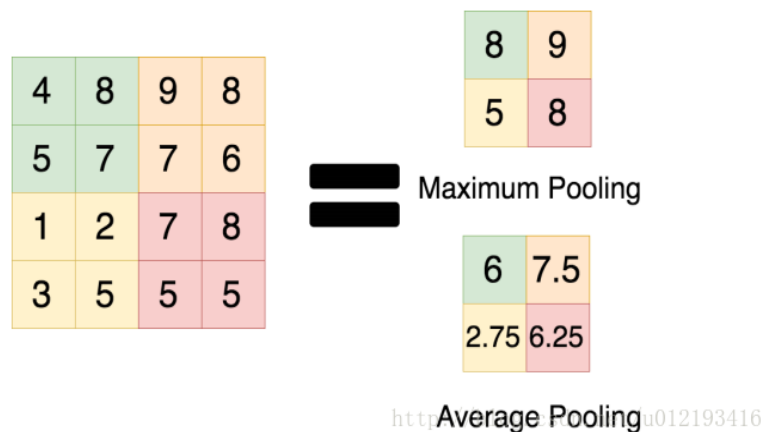


Figure 4 - Max Pooling & Average Pooling

### 2.1.3 Fully Connected Layer

The final layers involve connections from every output to every possible label. This allows the calculations of a probability score for every class possible. Once each score is calculated, the category with the highest score is used as the category for the image.

### **2.1.4 Forward & Backward Propagation**

In neural networks, forward propagation refers to how data is processed by the neural network to obtain a result. Whereas backward propagation refers to how the difference between obtained result and required result can be used to update the weights of the various layers in a neural network.

In CNNs, during back propagation we update the values of convolutional kernels for better representation of hidden features. Thus, improving the accuracy of the neural network.

## **2.2 Embedded Board**

Embedded boards consist of a variety of processors, ICs (Integrated circuits), storage and other essential components to serve a function. Embedded board have applications in automotive, industrial, and audio video. [17]

### **2.2.1 NVIDIA**

The Nvidia Corporation is a MNC who's primary objective is the design of graphical processing units. They provide frameworks that allows researchers to exploit the parallel processing capabilities of a GPU to run computationally expensive algorithms. There are a multitude of NVIDIA Jetson systems that allow the development of autonomous machines. They each have a complete SOM (System-on-Module), inclusive of CPU, GPU & storage. The products include Jetson Nano, the Jetson TX2 Series, Jetson Xavier NX, Jetson AGX Xavier Series.

## 2.2.2 Jetson Nano



Figure 5 - Jetson Nano Developer Kit [12]

For this project we make use of an NVIDIA Jetson Nano Developer Kit. The Jetson Nano is basically a small and compact computer system with an in-built CPU & GPU. The kit is used for the implementation of neural networks that require parallel threading capabilities. Its applications include image classification, object detection and speech processing [12]. The Jetson Nano is ideal for implementing small-scale projects such as mini robots that can sense and traverse complex environments.

The kits specifications are as follows:

CPU	Quad-core ARM A57 @ 1.43 GHz
GPU	128-core Maxwell
Memory	4 GB 64-bit LPDDR4 25.6 GB/s
USB	4x USB 3.0, USB 2.0 Micro-B
Display	HDMI and display port

### 2.2.2.1 Built-in capabilities

The Jetpack SDK for the Jetson Nano includes CUDA for GPU accelerated applications across multiple domains inclusive of NVCC which is a CUDA compiler. It forwards all non CUDA compilation to a C++ host compiler. Additionally, jetpack also includes TensorRT and cuDNN which are used for high-performance deep learning applications.



## 2.2.3 Add-On capabilities

### 2.2.3.1 OpenCV

A computer vision library used for image and video processing. Its applications include edge detection, histogram generation and image recognition.

### 2.2.3.2 Tegrastats

Jetson device utility used for memory usage and processor usage.

### 2.2.3.3 Jtop

We use Jtop as a profiling resource. It helps provide statistics on CPU, GPU and memory usage.

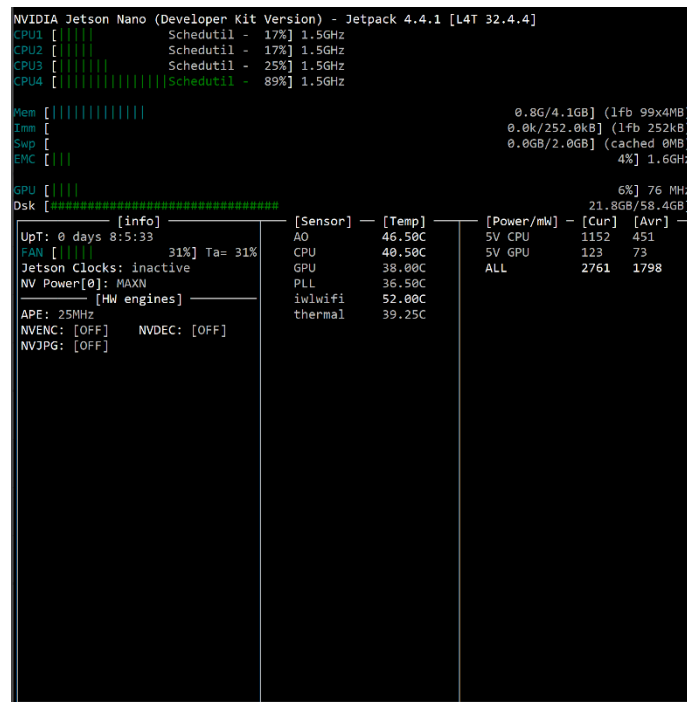


Figure 6 - Jtop overview page

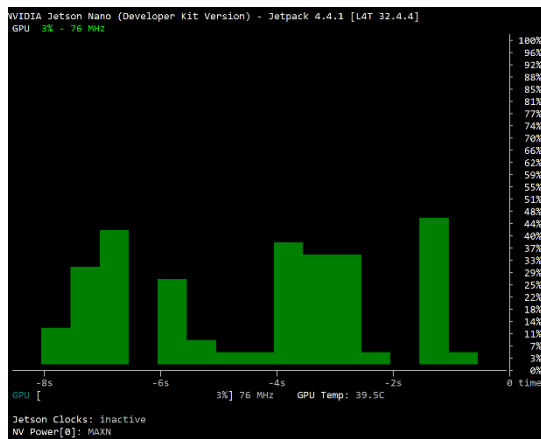


Figure 7- Jtop GPU Usage

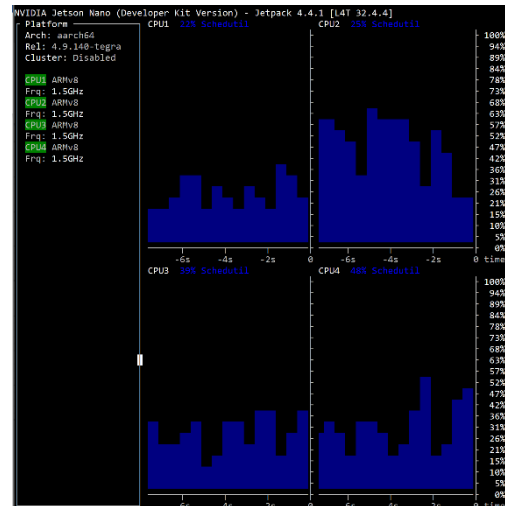


Figure 8 - Jtop CPU usage

## 2.3 GPU

### 2.3.1 Architecture

```

Device 0: "NVIDIA Tegra X1"
  CUDA Driver Version / Runtime Version      10.2 / 10.2
  CUDA Capability Major/Minor version number: 5.3
  Total amount of global memory:              3964 MBytes (4156682240 bytes)
  ( 1) Multiprocessors, (128) CUDA Cores/MP: 128 CUDA Cores
  GPU Max Clock rate:                        922 MHz (0.92 GHz)
  Memory Clock rate:                         13 Mhz
  Memory Bus Width:                          64-bit
  L2 Cache Size:                             262144 bytes
  Maximum Texture Dimension Size (x,y,z)      1D=(65536, 65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
  Total amount of constant memory:             65536 bytes
  Total amount of shared memory per block:     49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                  32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block:         1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z):   (2147483647, 65535, 65535)
  Maximum memory pitch:                       2147483647 bytes
  Texture alignment:                           512 bytes
  Concurrent copy and kernel execution:        Yes with 1 copy engine(s)
  Run time limit on kernels:                   Yes
  Integrated GPU sharing Host Memory:           Yes
  Support host page-locked memory mapping:     Yes
  Alignment requirement for Surfaces:          Yes
  Device has ECC support:                      Disabled
  Device supports Unified Addressing (UVA):     Yes
  Device supports Compute Preemption:          No
  Supports Cooperative Kernel Launch:          No
  Supports MultiDevice Co-op Kernel Launch:    No
  Device PCI Domain ID / Bus ID / location ID: 0 / 0 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

```

Figure 9 - GPU specifications

The statistics above refers to a specification sheet for the on-board GPU. The **NVIDIA Tegra X1** was released in 2015 using the **Maxwell** GPU architecture. Originally, It was intended to improve the gaming experience and deliver 4K video quality on mobile devices when it was released. It then went onto star in the NVIDIA SHIELD Android TV [15]. The Tegra X1 is also being used in the Jetson Nano as it includes capabilities that help advance computer vision and deep learning.

The Tegra X1 is equipped with features such as OpenGL, CUDA 6.0 and DirectX 12 API. The Tegra X1 used in mobile devices consisted of 256 CUDA cores. However, the Tegra X1 on the Jetson Nano consists of only 128 CUDA cores.

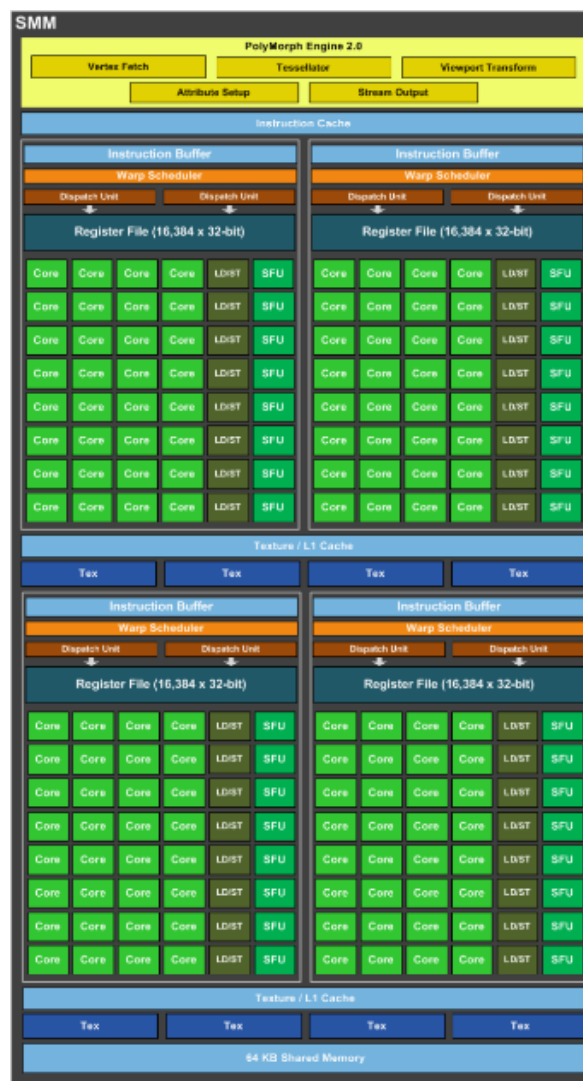


Figure 10 - Maxwell Architecture SMM

The NVIDIA Jetson Nano makes use of a GM20B GPU with CUDA compute capability 5.3. It has the **Maxwell 2.0** architecture. It inherits several features from its predecessor, the Kepler microarchitecture. The objective of the Maxwell architecture was to introduce improved Streaming Multiprocessor (SM). The image above shows the diagram of a single Streaming Multiprocessor.

The Maxwell SM has four 32-CUDA core processing blocks called SMP with a total of 128 CUDA cores in a single SM. Each partition has its own dedicated resources used for scheduling and instruction buffering. The architecture aligns with the warp size with the number of CUDA cores per partition (i.e., 32) which improves efficiency. CUDA makes use of SIMT (Single Instruction Multiple Threads) model. This is where threads that perform the same instruction are grouped together by a mechanism called warp. Each warp is then mapped to an SMP and instructions are sent to the 32 CUDA cores. Additionally, the architecture used a larger dedicated 64 kb shared memory for all partitions.

### 2.3.1 CUDA C

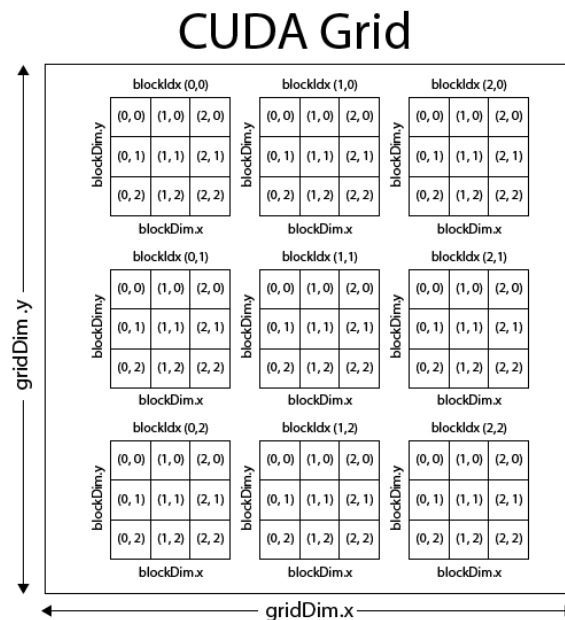


Figure 11 - CUDA C grid

### 2.3.1.1 Multi-Threading

The CUDA Grid is the basic knowledge required to understand the thread hierarchy. The compute capability of the Jetson Nano is 5.3 and its specifications can be found in [16].

- A CUDA grid consists of multiple CUDA blocks. Each block is executed consecutively by a single SM.
- A single SM can have up to 32 (or 2048 threads) blocks reside in it once depending on available resources. Once all threads within a block finished execution, its resources are freed, and another block can take its place [16].
- The maximum number of blocks is only limited by the maximum grid dimension which is equal to (2147483647, 65535, 65535) for the Jetson Nano.
- A maximum number of 1024 threads are allowed within a block.

CUDA C is an extension of C. One of the major differences come using kernels. Kernels are a function executed in parallel by different threads. The number of threads is defined by the user. To define a kernel, we use the `__global__` declaration. This declaration implies that the function is to be executed by the device (the GPU) and can be called only by the host only (the CPU). While making a call to a kernel function, we must establish an execution configuration. The execution configuration takes the format `<<< Dg, Db, Ns, S >>>`. **Dg** refers to the dimensions and size of the grid or the number of blocks that is to be used. **Db** refers to the dimensions and size of the blocks. **Ns** indicates the number of bytes in the shared memory that is to be dynamically allocated. Finally, **S** is an optional parameter that specifies the type of stream [7].

```

1 // Kernel definition
2 __global__ void VecAdd(float* A, float* B, float* C)
3 {
4     int i = threadIdx.x;
5     C[i] = A[i] + B[i];
6 }
7
8 int main()
9 {
10     int N = 10;
11     float A[10], B[10], C[10];
12
13     ...
14     // Kernel invocation with N threads
15     VecAdd<<<1, N>>>(A, B, C);
16     ...
17 }

```

Figure 12 - Simple CUDA kernel call

In line 4 we observe the use of ***threadIdx***. The ***threadIdx*** is a 3-component vector (***threadIdx.x***, ***threadIdx.y***, ***threadIdx.z***) used for identification of threads. Depending on the number of dimensions in the blocks, we can use the ***threadIdx*** to find a unique 3-dimensional ID to control the function of the thread. If the three-dimensional size of a block is given by ( $D_x$ ,  $D_y$ ,  $D_z$ ), the thread ID can be calculated as follows.

*Note: Id refers to a unique value associated to every thread whereas index is 3-dimensional coordinate (x, y, z)*

Table 1 - Calculation of thread ID

Dimensions	Thread ID
1 dimension - (x)	$x$
2 dimensions - (x, y)	$x + y * D_x$
3 dimensions - (x, y, z)	$x + y * D_x + z * D_x * D_y$

The total number of threads in a block is limited to 1024 as shown in the figure 10. However, to accommodate for this, we can make use of multiple blocks containing equal number of threads.

$$\text{total number of threads} = \text{number of blocks} * \text{number of threads per block}$$

To calculate thread index for threads across multiple blocks, we make use of **blockIdx** & **blockDim**. **blockIdx** is a 3-component vector (**blockIdx.x**, **blockIdx.y**, **blockIdx.z**) used to access the block index of the block in which the thread is contained. Whereas **blockDim** is a 3-component vector (**blockDim.x**, **blockDim.y**, **blockDim.z**) used to access the dimensions of the block. When dealing with multiple blocks, we use the following formulas to access the thread index.

$$x = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

$$y = \text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y}$$

$$z = \text{blockIdx.z} * \text{blockDim.z} + \text{threadIdx.z}$$

```

1  #define N 32
2
3  // Kernel definition
4  __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
5  {
6      int i = blockIdx.x * blockDim.x + threadIdx.x;
7      int j = blockIdx.y * blockDim.y + threadIdx.y;
8
9      if (i < N && j < N) C[i][j] = A[i][j] + B[i][j];
10 }
11
12 int main()
13 {
14     float A[N][N], B[N][N], C[N][N];
15     ...
16     // Kernel invocation
17     dim3 threadsPerBlock(16, 16); // 16x16 threads per block
18     dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y); // 2x2 blocks
19     MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
20     ...
21 }
22

```

Figure 13 - Thread & Block implementation

However, the above implementation would fail to compile. In CUDA C, the GPU is a secondary device and the system on which the programs run are considered the **host**. Therefore, the GPU has its own separate memory and to access it, we must make use of function calls that help transfer the data from host to device. To assign linear memory, we make use of the following function calls.

- **cudaMalloc**: allocate linear memory on the device.
- **cudaMemcpy**: transfer data from host to device memory.
- **cudaFree**: free memory allocated on device.

```

1  #include <iostream>
2  #include <stdio.h>
3  #include <cuda.h>
4
5  #define N 32
6
7  // Kernel definition
8  __global__ void MatAdd(float* A, float* B, float* C)
9  {
10     int i = blockIdx.x * blockDim.x + threadIdx.x;
11     int j = blockIdx.y * blockDim.y + threadIdx.y;
12
13     if (i < N && j < N) C[i*N + j] = A[i*N + j] + B[i*N + j];
14 }
15
16 int main()
17 {
18     size_t size = N * N * sizeof(float);
19
20     // Allocate input vectors h_A and h_B in host memory
21     float* h_A = (float*)malloc(size);
22     float* h_B = (float*)malloc(size);
23     float* h_C = (float*)malloc(size);
24
25     // Initialize input vectors
26     for (int i = 0; i < N; i++)
27     {
28         for (int j = 0; j < N; j++) {
29             h_A[i*N + j] = i+j;
30             h_B[i*N + j] = i-j;
31         }
32     }
33
34     // Allocate vectors in device memory
35     float* d_A;
36     cudaMalloc(&d_A, size);
37     float* d_B;
38     cudaMalloc(&d_B, size);
39     float* d_C;
40     cudaMalloc(&d_C, size);
41
42     // Copy vectors from host memory to device memory
43     cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
44     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
45
46     // Kernel invocation
47     dim3 threadsPerBlock(16, 16);
48     dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
49     MatAdd<<<numBlocks, threadsPerBlock>>>>(d_A, d_B, d_C);
50
51     // Copy result from device memory to host memory
52     // h_C contains the result in host memory
53     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
54
55     // Free device memory
56     cudaFree(d_A);
57     cudaFree(d_B);
58     cudaFree(d_C);
59 }

```

Figure 14 - Memory allocation

The above code is a complete implementation of a two-dimensional matrix addition using the parallel threading with multiple blocks.



### 2.3.1.2 Memory Hierarchy

Table 2 - Function Type Qualifiers

<i>Function Type Qualifiers</i>	<code>__device__</code>	<code>__global__</code>	<code>__host__</code>
<b>Executed on</b>	Device	Device	Host
<b>Callable from</b>	Device	Host	Host

Function type qualifiers help indicate whether a function will be executed by the device or the host and who has access to call the function. As seen earlier, kernels used `__global__` qualifier to indicate functions that are callable by the host but are to be executed on the device. For functions that can either be called by host or device, we can use `__host__` & `__device__` together while declaring a function.

Table 3 - Variable Type Qualifiers

<i>Variable Type Qualifiers</i>	<code>__device__</code>	<code>__constant__</code>	<code>__shared__</code>
<b>Location</b>	global memory space	constant memory space	shared memory space of thread block
<b>Lifetime</b>	application	application	thread block
<b>Accessibility</b>	all thread within grid	all thread within grid	all threads within block

Variable type qualifiers are more important as we will be constantly working with them in this project. The variable type qualifiers establishes where the data will be stored in device memory and how long it will be stored in the specified location. This understanding helps us navigate various methods of improving the throughput of the system by taking advantage of these available memory features.

CUDA threads may access data from multiple memory spaces during their execution.

- **Each thread** has **private local memory**.
- **Each thread** block has **shared memory** visible to all **threads of the block** and with the same lifetime as the block.
- **All threads** have access to the same **global memory**.
- There are also two additional read-only memory spaces accessible **by all threads**: the **constant** and **texture memory** spaces.

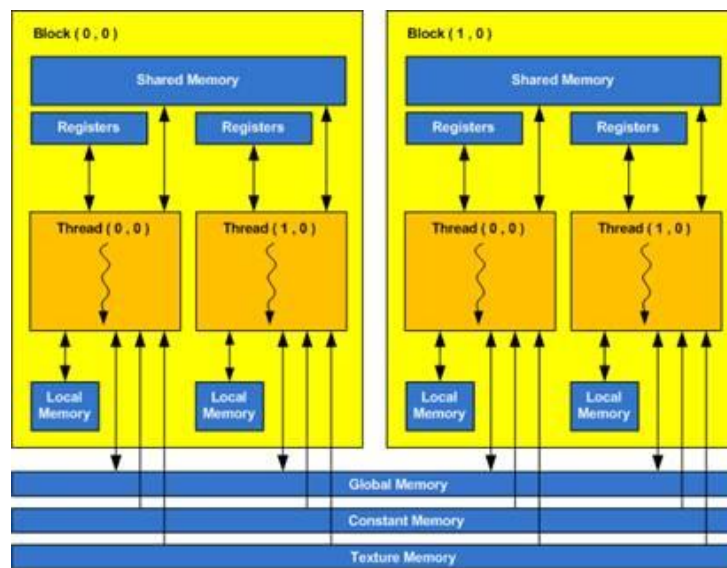




Figure 15 - Memory hierarchy

**Global Memory:** Global memory is accessible and shared by all threads as it resides on the device. The memory transactions with global memory generally take place as 32-, 64- or 128- byte sized transactions. The warp executes access to the global memory, and it does so by grouping memory accesses of multiple threads. This is done as each transaction to the memory brings along with it more words than necessary. Therefore, a greater number of transactions leads to a reduction in throughput.

**Shared Memory:** This memory is shared between threads of the same block. When compared to global memory, it has higher bandwidth and lower latency. Shared memory makes use of the concept of banks. This is where the memory is equally partitioned into modules referred to as banks. These banks allow simultaneous read and write and hence improves its bandwidth. The hardware tries to optimize the process by splitting memory accesses to avoid or reduce the number bank conflicts. Bank conflicts occur when two memory requests are made to the same memory bank.

Shared memory is an important concept to improve the throughput of any CUDA system. It behaves as a user defined cache. It works such that we define a shared memory location in the block (using the `__shared__` type qualifier as mentioned before). Then each thread makes an assignment based on the thread ID. These memory locations can then be used by any thread within the same block, regardless of which thread assigned it.

Eg. Consider a summation of 3 elements in a 1D array done with a block size of 7.

Global Memory	1	3	2	4	3	5	4	6	5
									
Shared Memory		3	2	4	3	5	4	6	
									
Result			9	9	12	12	15		

An important CUDA function extensively used with shared memory is the `__syncthreads()` function. Its objective is to wait for all threads within a block to reach the same point to avoid conflicts. This is important when using shared memory, so as to avoid accessing shared memory locations that have not been assigned by a thread yet.

**Constant Memory:** Constant memory resides in a read-only device memory and is cached in constant cache memory. A read from constant memory takes one read cycle from constant cache or one read cycle from constant memory in the case of a cache miss. Warp requests are first broken down into two requests, one for each warp. Once this is done, for each half warp, the requests are further broken into read requests for distinct memory locations and decreases throughput by the number of requests per memory location. The throughput increases linearly with the number of distinct requests. Constant memory is used when there is a need to have access to data that will not change during the lifetime of execution.

**Texture Memory:** Texture memory is like global memory, with an added feature of another read-only cache located on the device. It is designed to perform fetching with similar latency. Texture should be used when threads of a block accesses different areas of the bound-global memory in a non-orderly fashion. Texture memory is optimized for 2D spatial locality and provides better performance when threads of the same warp try to access memory locations that are neighboring each other in 2D space.

# Chapter 3

## Materials & Methodology

### 3.1 Jetson nano board set up.

The NVIDIA Jetson Nano can host an Operating System (OS). For our initial setup we make use of Jetson Nano Developer kit more commonly known as JetPack (latest version 4.5.1) [13]. This is available as an open resource on the Jetson Nano home page. Once we download the kit, we load it into a SD card using a software known as Etcher. The software allows us to write the image of the OS into the SD.

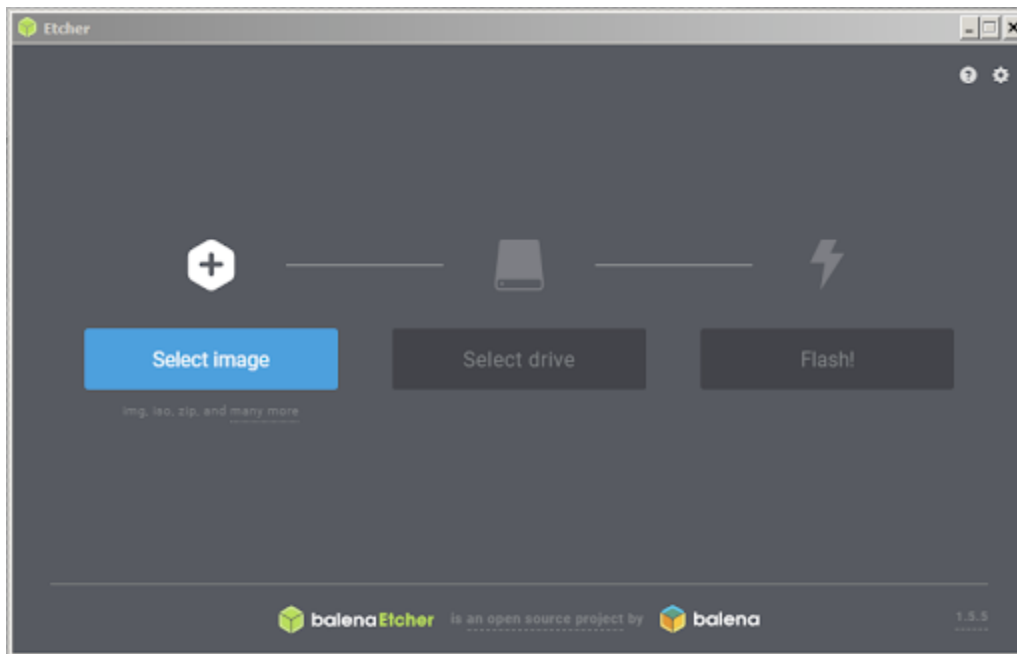


Figure 16- Etcher Interface

We then insert the microSD card into the Jetson Nano, after which we connect a power cable to power on the machine. In addition, we connect a monitor via the HDMI port, a mouse, and a keyboard to set-up the system configurations.

This step can also be accomplished headless via the use of PuTTY. However, this requires a barrel jack connector to power the device as the micro-USB slot is required to access initial configuration prompts. Ubuntu is the OS included in JetPack.

### 3.1.1 Interfacing

To access the Jetson Nano, we use SSH (Secure Shell). The use of SSH allows us to remotely login and use issue commands via the terminal on the system. However, to do so, the IP address of the server machine must be known.

We first connect the board via the ethernet cable to a computer. We observe that the SSH is not possible as a static IP address has not been provided to the device. We then configure the connection (Ethernet 2) by changing the IPv4 properties under network connections by setting the IP address to 192.168.1.1 with a subnet mask of 255.255.255.0.

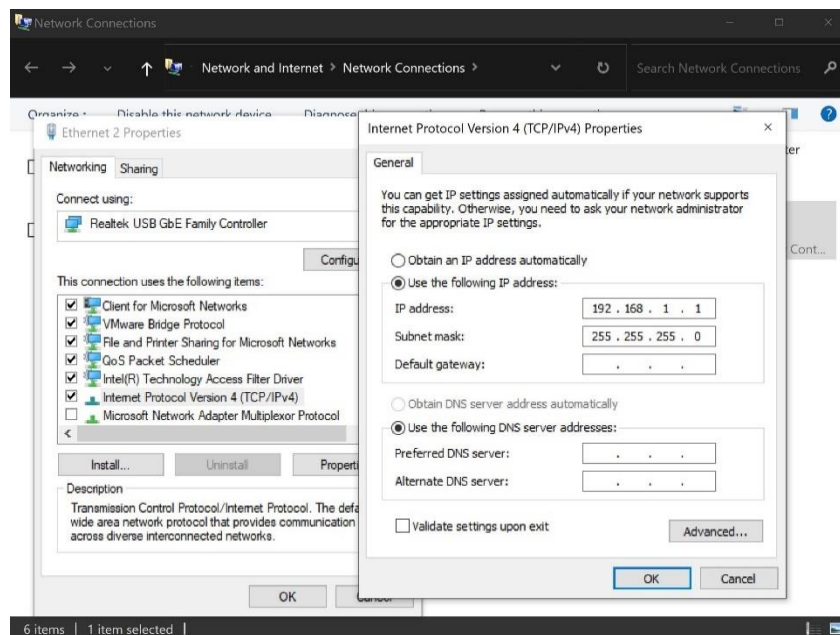


Figure 17 - set IPv4 properties.

We then download a DHCP (Dynamic Host Control Protocol) server that can be used to run locally on a windows machine from [www.dhcpserver.de](http://www.dhcpserver.de). We then configure the DHCP Server using the DHCP wizard as follows.

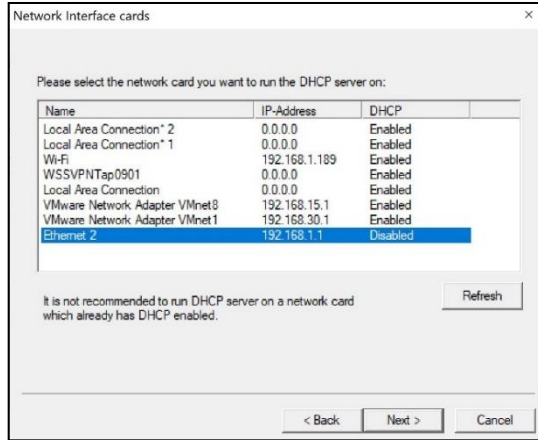


Figure 18 - choose network range.

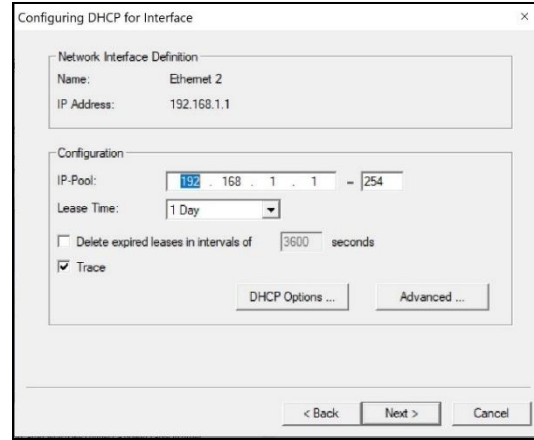


Figure 19 - choose network card

We first set the DHCP server to run on the network card of the connection for which the DHCP server is disabled. We then configure the DHCP server to range from 192.168.1.1 – 192.168.1.254. We save these configurations.

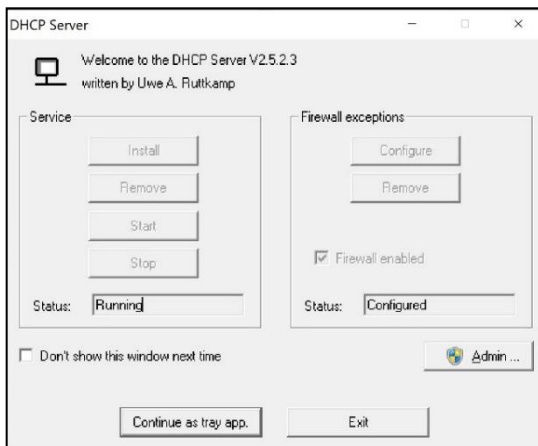


Figure 20 - Run DHCP Server

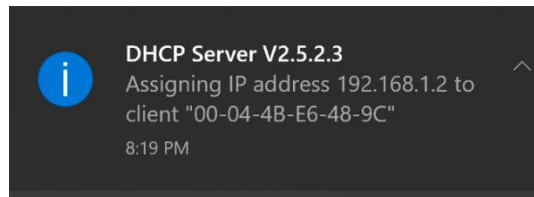


Figure 21 - Get IP address given by DHCP

We then start the DHCP server with administrator privileges. Once the DHCP server is up and running, it assigns IP addresses to devices on the network. In a few minutes, a notification pops up stating that Jetson Nano has been assigned the IP address 192.168.1.2. We then use this address to SSH into the machine and access it.

```

PS C:\Users\steph> ssh stephen@192.168.1.2
stephen@192.168.1.2's password:
Welcome to Ubuntu 18.04.5 LTS (GNU/Linux 4.9.140-tegra aarch64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage
This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

22 packages can be updated.
21 of these updates are security updates.
To see these additional updates run: apt list --upgradable

Last login: Sun Mar 14 19:54:28 2021 from 192.168.1.189
stephen@stephen-desktop:~$

```

Figure 22 - SSH into machine

Once we have access into the system, with a few simple steps we can set up a WIFI connection [14].

- `ifconfig wlan0 up`
- `iwconfig wlan0 essid WIFI_NETWORK_HERE key PASSWORD_HERE`
- `dhclient wlan0`

```

wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.1.45 netmask 255.255.255.0 broadcast 192.168.1.255
    inet6 fe80::e834:2ccf:be9a:3ad5 prefixlen 64 scopeid 0x20<link>
    ether 4c:1d:96:c8:3b:97 txqueuelen 1000 (Ethernet)
    RX packets 4993 bytes 5096982 (5.0 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1854 bytes 193127 (193.1 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

Figure 23 - ifconfig for Jetson Nano with WIFI

Once connected to the WIFI, we can use *ifconfig* to find the IP address using the WIFI module. This would allow us to SSH into the machine using the WIFI rather than ethernet, thus allowing mobility. Furthermore, when connected via ethernet, the system diverts all internet requests via the ethernet cable, and hence is unable to connect to the internet. This method allows us to be connected to Jetson Nano while at the same time allowing the Jetson board to be connected to the internet.



## 3.2 CUDA Implementation

As discussed, convolution is the most important and time-consuming function in CNNs. Thus, we investigate different methods to improve the efficiency of convolution algorithm with the help of a GPU and use of different types of memory for faster retrieval and processing of data.

### 3.2.1 Convolution using CPU (Serial)

We process the convolution operation by first reading the convolutional kernel and image in a 1-dimensional array. This is easier to handle and transfer to device memory (GPU) during execution. We then repeat the convolution process on every pixel in the image to get the corresponding output pixel. The pixels are then used to form the convoluted image.

```

1 float applyKernelPerPixel(int y, int x, int kernelX, int kernelY, int imageWidth, int imageHeight, float *kernel, float *image)
2 {
3     /*
4     * This function applies the kernel on the pixel @ (x,y) of the image.
5     */
6     parameters:
7     int y : Y-coordinate of pixel
8     int x : X-coordinate of pixel
9     int kernelX : Kernel width
10    int kernelY : Kernel height
11    int imageWidth : Image width
12    int imageHeight : Image Height
13    float *kernel : 2D float matrix containing kernel values
14    float *image : 2D float matrix containing image values
15
16    return : value of convoluted image pixel @ (x,y)
17    */
18    float sum = 0;
19    int offsetX = (kernelX - 1) / 2;
20    int offsetY = (kernelY - 1) / 2;
21
22    for (int j = 0; j < kernelY; j++)
23    {
24        //Ignore out of bounds
25        if (y + j < offsetY || y + j - offsetY >= imageHeight)
26            continue;
27
28        for (int i = 0; i < kernelX; i++)
29        {
30            //Ignore out of bounds
31            if (x + i < offsetX || x + i - offsetX >= imageWidth)
32                continue;
33
34            float k = kernel[i + j * kernelY];
35            float imageElement = image[y * imageWidth + imageWidth * (j - offsetY) + x + i - offsetX]; //Get value at kernel positions (i,j)
36            float value = k * imageElement; //Get value at image positions (x - offset + i, y - offset + j)
37            sum = sum + value;
38        }
39    }
40    return sum;
41 }

```

Figure 24 - Serial convolutional implementation

The illustration below is used to demonstrate the working of the code.

We first load kernel values from an external file. Once the kernels are loaded, we normalize the kernel by dividing all values in it by the sum of the values. This helps us generate an image with a blurred effect. This is to properly analyze and compare results. However, the normalization step can be skipped or replaced to produce images with higher degrees of convolution.

The image is also loaded into a 1-dimensional array that is equal to the size of the product of image width and the image height. We then use the index location of the center element to find the surrounding elements with which convolution is to be done. In the example below, we use an image of size 4x5 and the pixel 2x2 is the center of the kernel with which convolution is to take place.

*Note: 1st row: Pixel value, 2nd row: 1D-index, 3rd row: Image position (RxC)*

0	255	34	26	78	34	1	245	219	2	3	99	101	111	200	78	33	56	4	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0x0	0x1	0x2	0x3	0x4	1x0	1x1	1x2	1x3	1x4	2x0	2x1	2x2	2x3	2x4	3x0	3x1	3x2	3x2	3x4

1	2	1
3	2	1
1	2	3

 $*$ 

1	245	219
99	101	111
33	56	4

 $=$ 

1477
------

### 3.2.2 Convolution using GPU (Parallel)

To improve the efficiency of the algorithm, we farm out the convolution operation to the GPGPU with the help of CUDA C. In such a scenario, multiple threads are used to run the convolutional kernel on each pixel in parallel.

We use the x and y coordinate of the thread to perform convolution on the corresponding pixel of the image. The kernel and image data are copied from host memory to the global memory of the device from which all threads in all blocks of the GPU can access.

*Note: Further code will emphasize on changes done on the above code and not the entire code itself*

```

2 float *applyKernelToImageParallelNaive(float *image, int imageWidth, int imageHeight, kernel *kernel, char *imagePath, int blockWidth)
3 {
4     /*
5     This function makes use of GPU threading capability by assign each thread a pixel to apply the convolutional kernel on
6     */
7     float *image : 1D float matrix containing image values
8     int imageWidth: Image width
9     int imageHeight: Image Height
10    kernel* kernel: Kernel to be applied on image
11    char *imagePath: Path to image being convoluted
12    int blockWidth: Width and height of a single block
13
14    return : Resultant image as a 1D float matrix
15    */
16
17    // Variables that will point to values on the device
18    int *d_kernelDimensionX, *d_kernelDimensionY, *d_imageWidth, *d_imageHeight;
19    float *d_kernel, *d_image, *d_sumArray;
20
21    int sizeInt = sizeof(int);
22    int sizeFloat = sizeof(float);
23    int sizeImageArray = imageWidth * imageHeight * sizeFloat;
24
25    // Resultant convoluted image as a 1D float matrix
26    float *sumArray = (float *)malloc(sizeImageArray);
27
28    // CUDA create variables
29    cudaMalloc((void **)&d_image, sizeImageArray);
30    ...
31
32    // CUDA copy from host to device
33    cudaMemcpy(d_image, image, sizeImageArray, cudaMemcpyHostToDevice);
34    ...
35
36    int numHorBlocks = imageWidth / blockWidth;
37    int numVerBlocks = imageHeight / blockWidth;
38
39    if (imageWidth % blockWidth)
40        numHorBlocks++;
41    if (imageHeight % blockWidth)
42        numVerBlocks++;
43
44    dim3 dimGrid(numVerBlocks, numHorBlocks, 1);
45    dim3 dimBlock(blockWidth, blockWidth, 1);
46
47    applyKernelPerPixelParallel<<<dimGrid, dimBlock>>>(d_kernelDimensionX, d_kernelDimensionY, d_imageWidth, d_imageHeight, d_kernel, d_image, d_sumArray);
48    cudaMemcpy(sumArray, d_sumArray, sizeImageArray, cudaMemcpyDeviceToHost); // Copy image from resultant array on device to host
49
50    // CUDA free variables
51    cudaFree(d_image);
52    ...
53
54    return sumArray;
55 }

```

Figure 25 - Parallelized calls for convolution

Since most of the change occurs in how the function to apply the kernel on each pixel is called, the above code uses basics discussed in (CUDA C) to implement parallel threading. The above code snippet shows how we break down an image into individual blocks of size BLOCKWIDTHxBLOCKWIDTH. We then use the kernel index to access the pixel that is to be convoluted by the thread.

```

2 _global_ void applyKernelPerPixelParallel(int *d_kernelDimensionX, int *d_kernelDimensionY, int *d_imageWidth,
3                                           int *d_imageHeight, float *d_kernel, float *d_image, float *d_sumArray)
4 {
5     ...
6     int y = blockIdx.y * blockDim.y + threadIdx.y;
7     int x = blockIdx.x * blockDim.x + threadIdx.x;
8     ...
9
10    ...
11 }

```

Figure 26 - Code snippet for getting image pixel coordinate.

### 3.2.3 Convolution in Parallel using Shared Memory.

In comparison to local or global memory, shared memory has a higher bandwidth and lower latency since it is on-chip. Shared memory is accessible to all threads within a block of threads.

We make use of the shared memory to save a section of the image that is relevant to the threads within a block. Thus, reducing access to the global memory and reducing latency for memory access of image information.

Shared memory is implemented during the execution of kernels as discussed before. Therefore, we take advantage of this process by loading image values corresponding to a block by making use of the thread index. E.g., If we have a block size of 16x16, we load a region of the image of size 16x16 in the shared memory. However, as convolution depends not just on the pixel value itself but the surrounding pixels as well, we must make use of blocks that overlap in image space.

```

2 float *applyKernelToImageParallelSharedMemory( ... )
3 {
4     ...
5     ...
6     ...
7     int overlapX = (kernel.dimension - 1);
8     int overlapY = (kernel.dimension - 1);
9     ...
10    int numHorBlocks = (imageWidth) / (blockWidth - overlapX);
11    int numVerBlocks = (imageHeight) / (blockWidth - overlapY);
12    ...
13    if (imageWidth % (blockWidth - overlapX))
14        numHorBlocks++;
15    if (imageHeight % (blockWidth - overlapY))
16        numVerBlocks++;
17    ...
18    dim3 dimGrid(numVerBlocks, numHorBlocks, 1);
19    dim3 dimBlock(blockWidth, blockWidth, 1);
20    applyKernelPerPixelParallelSharedMemory<<<dimGrid, dimBlock, blockWidth*blockWidth*sizeFloat>>>
21        (d_kernelDimensionX, d_kernelDimensionY, d_imageWidth, d_imageHeight, d_kernel, d_image, d_sumArray);
22    ...
23    ...
24    ...
25 }

```

Figure 27 - Shared memory implementation

```

27  global__ void applyKernelPerPixelParallelSharedMemory( ... )
28  {
29      ...
30      ...
31      extern __shared__ float local_imageSection[];
32      local_imageSection[row*(blockDim.x) + col] = d_image[y * (*d_imageWidth) + x];
33      ...
34      __syncthreads();
35      ...
36      // If blockDim.x is 0 do not start from offset otherwise start from offset (overlapping region calculated by previous block)
37      if ((blockIdx.x == 0 || (blockIdx.x != 0 && threadIdx.x >= offsetX))
38          // End as blockDim - offset (overlapping region calculated by next block)
39          && threadIdx.x < blockDim.x - offsetX
40          // If blockDim.y is 0 do not start from offset otherwise start from offset (overlapping region calculated by previous block)
41          && (blockIdx.y == 0 || (blockIdx.y != 0 && threadIdx.y >= offsetY))
42          // End as blockDim - offset (overlapping region calculated by next block)
43          && threadIdx.y < blockDim.y - offsetY)
44      {
45          ...
46          ...
47          ...
48          ...
49      }
50  }

```

Figure 28 - Shared memory implementation kernel

Take note in line 20 of figure 27, we make use of the shared memory size parameter to dynamically allocate shared memory and declaring the shared variable on line 32 as an **extern** variable. This is to avoid unnecessary allocation of shared memory as such memory allocations are time consuming and lead to a waste of memory.

Furthermore, with reference to figure 9, we observed that shared memory is allocated only 49152 bytes per block. This therefore sets an upper limit to block width as larger block widths would require more memory per block.

$$\frac{\text{shared memory per block}}{\text{sizeof(float)}} = \frac{49152}{4} = 12288$$

$$\text{max block width} = \sqrt{12288} \sim 110$$

However, this number is inconsequential, as the max number of threads per block as shown in figure 9 is 1024. Therefore, max block width is constrained to 32x32.

The figure below helps us understand our shared memory implementation with depth (not drawn to scale). The overlap region indicated are regions partially calculated by two blocks: block (0,0) and block (0,1). This is because to convolve the values of pixels on the left of

the overlapping region, we require values to the right of the pixel as well. Similarly, in block (0,1) we find the convoluted values of pixels to the right of the overlapping region by considering pixels to the left of the overlapping region as well.



Figure 29 - Shared memory working

However, the use of shared memory in such a manner may lead to a significant decrease in throughput. This is because the effective width of a block is reduced by a factor of kernel size – 1. Therefore, requiring a greater number of blocks and threads to process the entire image.

Table 4 - Parallel vs Shared memory implementation

	<i>Image Size</i>	<i>Block Size</i>	<i>Kernel Size</i>	<i>Effective Blocks Size</i>	<i>Grid Size</i>	<i># of blocks</i>
<b>Naïve Parallel</b>	512x512	16x16	5x5	16x16	32x32	1024
<b>Parallel – Shared Memory</b>	512x512	16x16	5x5	12x12	43x43	1849

### 3.2.4 Convolution in Parallel using Constant Memory.

Constant memory refers to a read only memory that resides on the device and is cached in the constant cache. Since the kernel, kernel size and image size remain constant for all threads, we make use of the constant memory to store this information. Since this information is stored in cache memory, access time is reduced and hence performance is increased.

```

1  ....
2
3  _constant_ float kernelConstant[64 * 64];
4  _constant_ int  imageWidthConstant;
5  _constant_ int  imageHeightConstant;
6  _constant_ int  kernelDimensionXConstant;
7  _constant_ int  kernelDimensionYConstant;
8
9  float *applyKernelToImageParallelConstantMemory(...)
10 {
11     ...
12
13     //CUDA constants
14     cudaMemcpyToSymbol(kernelConstant, kernel.matrix, sizeof(float) * kernel.dimension * kernel.dimension);
15     cudaMemcpyToSymbol(imageWidthConstant, &imageWidth, sizeof(int));
16     cudaMemcpyToSymbol(imageHeightConstant, &imageHeight, sizeof(int));
17     cudaMemcpyToSymbol(kernelDimensionXConstant, &kernel.dimension, sizeof(int));
18     cudaMemcpyToSymbol(kernelDimensionYConstant, &kernel.dimension, sizeof(int));
19
20     ...
21 }

```

Figure 30- Constant memory implementation

We first declare the constant variables outside all function definitions as global variables. These variables sizes do not affect the throughput of the convolution function. This is because the variables are created and allocated at the beginning of program execution. They are only assigned values when the function is called. We have assigned an arbitrary size of 64x64 for the kernel in our implementation. However, max size of constant memory with reference to figure 19 is 64 kb and we can have a kernel of size 128x128 if no other constant variables are declared.

### 3.2.5 Convolution in Parallel using Shared & Constant Memory.

We use a combination of shared and constant memory to see how much we can better our convolution algorithm. This is done by declaring as constants, values that do not change regardless of which thread accesses them, such as the kernel, kernel height and width, and image height and width. Furthermore, we break the image into blocks which is then passed on to the shared memory of a block and can be accessed by threads depending on the thread index.

### 3.2.6 Convolution in Parallel using Texture Memory.

Texture memory is also a read-only memory that resides on the device but are cached in texture cache.

```

3  ...
4  texture<float, 1, cudaReadModeElementType> texRefId;
5
6  float *applyKernelToImageParallelTextureMemory(...)
7  {
8      ...
9
10     cudaBindTexture(0, texRefId, d_image, sizeImageArray);
11     ...
12 }
13
14 __global__ void applyKernelPerPixelParallelTextureMemory(...)
15 {
16     ...
17
18     float imageElement = tex1Dfetch(texRefId, y * (*d_imageWidth) + x + i - offsetX + (*d_imageWidth) * (j - 1));
19     ...
20 }
21

```

Figure 31 - Texture memory implementation



### 3.2.7 Convolution in Parallel using 2D Texture Memory.

Texture cache is optimized for 2D spatial locality and hence threads of the same warp that close together in 2D will achieve the best result.

```

3  ...
4
5  //2d texref
6  texture<float, 2, cudaReadModeElementType> texRef;
7
8
9  float *applyKernelToImageParallel2DTextureMemory(...)
10 {
11     ...
12
13     // Texture memory - 2d
14     // cudaArray *cuArray;
15     cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
16     cudaBindTexture2D( NULL, texRef,
17         d_image,
18         channelDesc, imageWidth, imageHeight,
19         sizeof(float) * imageWidth );
20
21     ....
22 }
23
24 __global__ void applyKernelPerPixelParallel2DTextureMemory(...)
25 {
26     ...
27     float imageElement = tex2D(texRef, x + i - offsetX , y + j - offsetY);
28
29     ...
30 }

```

Figure 32 - 2D Texture memory implementation

## 3.3 OpenCV Implementation

The code below refers to a basic implementation of convolution both in serial and OpenCV's interface for CUDA C.

```

2  ...
3  cv::Mat image = imread(file_path); // Read image file
4  cv::cvtColor(image, result, cv::COLOR_BGR2GRAY); // Convert to greyscale
5  ...
6
7  cv::normalize(k, k, 1.0, 0.0, NORM_L1); // Normalize between 0 to 1
8  ...
9
10 cv::filter2D(result, result, -1, k, Point(-1, -1), 5.0, BORDER_REPLICATE); // Apply serial filter
11 ...

```

Figure 33 - OpenCV serial convolution implementation

```

14 cv::Mat image = imread(file_path); // Read image file
15 cv::cuda::GpuMat gpu_image, gpu_result, gpu_kernel; // Declare GPU matrices
16 ...
17
18 cv::cvtColor(image, image, cv::COLOR_BGR2GRAY); // Convert to greyscale
19 gpu_image.upload(image); // Upload to GPU Matrix
20 gpu_image.convertTo(gpu_image, CV_32FC1); // Convert to 32_FC2 Type
21 ...
22
23 gpu_kernel.upload(k); // Upload to GPU Matrix
24 gpu_kernel.convertTo(gpu_kernel, CV_32FC1); // Convert to 32_FC2 Type
25 cv::normalize(k, k, 1.0, 0.0, NORM_L1); // Normalize between 0 to 1
26
27 ...
28
29 Ptr<cuda::Convolution> convolver = cuda::createConvolution(cv::Size(dim, dim)); // Create CUDA convolution ptr
30 convolver->convolve(gpu_image, gpu_kernel, gpu_result); // Apply convolution
31 ...
32
33 gpu_result.download(result); // Download result from GPU

```

Figure 34 - OpenCV Parallel convolution implementation

# Chapter 4

## Results and Discussion

### 4.1 CUDA Results

The execution time and speedup of the different methods used for convolution have been recorded in tables given in appendix A and appendix B respectively. The table contrasts the convolution methods for different sizes of kernel, images, and block sizes. We shall now look at trends to compare and analyze the different methods. These results are calculated after performing the functions for 100 iterations or after running for 5 mins, whichever comes first.

#### 4.1.1 Effect of kernel size on convolution algorithms

Table 5 - Execution time comparison based on kernel size.

<b>Kernel Size</b>	<b><i>Serial</i></b>	<b><i>Naïve Parallel</i></b>	<b><i>Shared Memory Parallel</i></b>	<b><i>Constant Memory Parallel</i></b>	<b><i>Shared + Constant Memory Parallel</i></b>	<b><i>Texture Memory Parallel</i></b>	<b><i>2D Texture Memory Parallel</i></b>
3x3	75.408	19.588	12.78	7.8	9.991	11.521	10.495
5x5	197.797	16.741	20.109	14.477	16.147	24.238	24.787
7x7	363.994	24.448	28.459	23.014	24.674	27.209	27.182
9x9	588.575	22.005	25.262	26.221	25.703	29.397	32.191
11x11	864.658	24.921	28.228	27.175	30.296	31.767	29.996

The above table refers to average time taken to complete a function in **milliseconds**. The images used was of size 512x512 and the size of blocks used in the grid were of size 16x16.

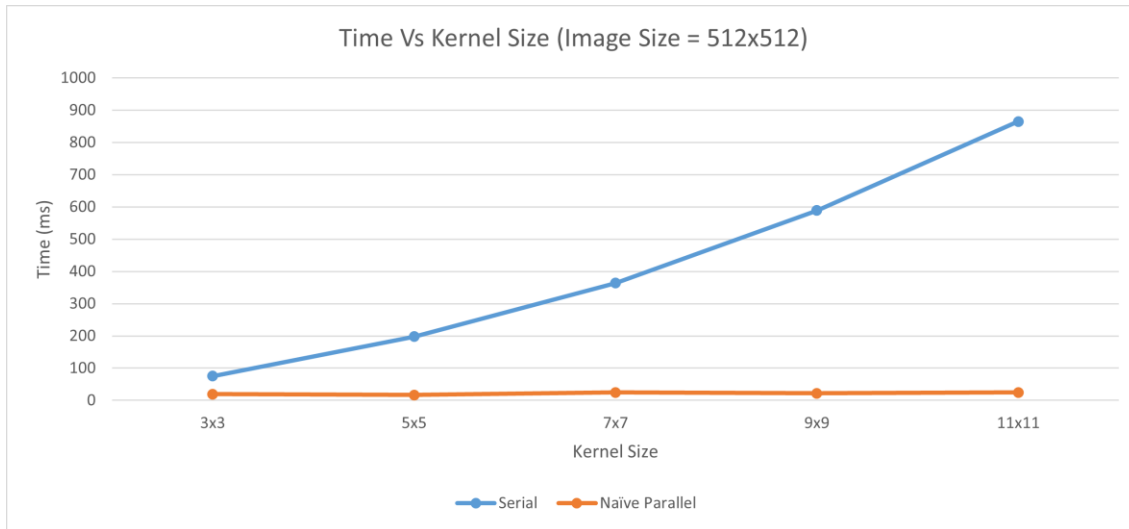


Figure 35 – Kernel size comparison for serial &amp; parallel

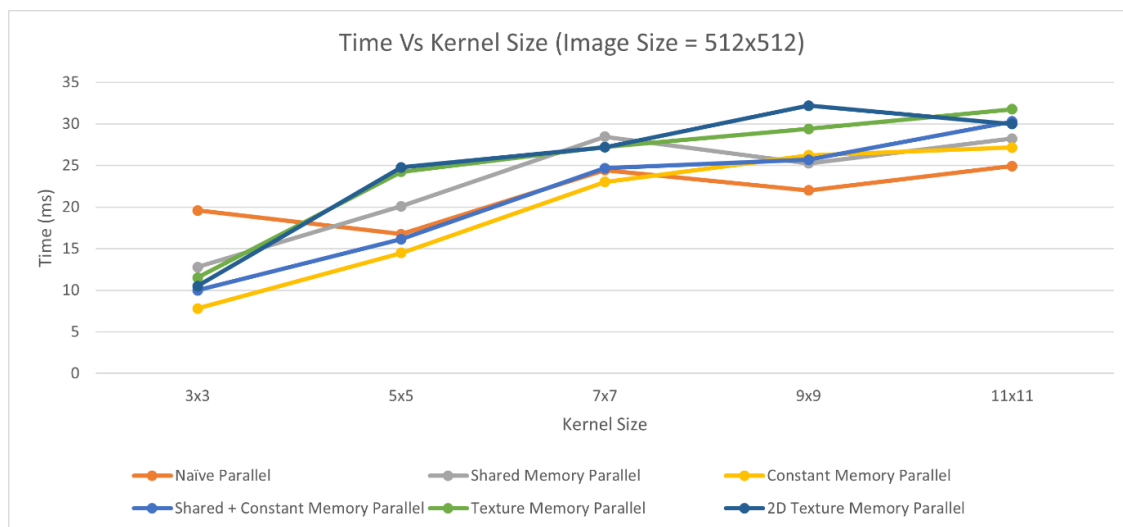


Figure 36 – Kernel size comparison for different parallel methods

Our major observation arises from the improvement from serial to parallel implementation. The time taken to complete a convolution using a serial CPU increases almost exponentially with an increase in kernel size. In comparison, our parallel GPU implementation maintains almost constant time.

On closer analysis of GPU implementations, we make a variety of different and important observations. Firstly, for a kernel of size 3x3, we observe that a Naïve parallel method takes

the most time when compared to other implementations. This trend however is not observed as the kernel size grows larger. Secondly, constant memory shows the best performance, over runs multiple runs and image sizes. Constant memory consistently provides performance equal or better than a naïve parallel implementation. However contrary to expectations, shared memory and shared with constant memory implementation fails to provide adequate performance. Finally, texture and texture 2D provides similar performance but both show the worst performance of all implementations. These observations are more pronounced, as we increase the image size as follows.

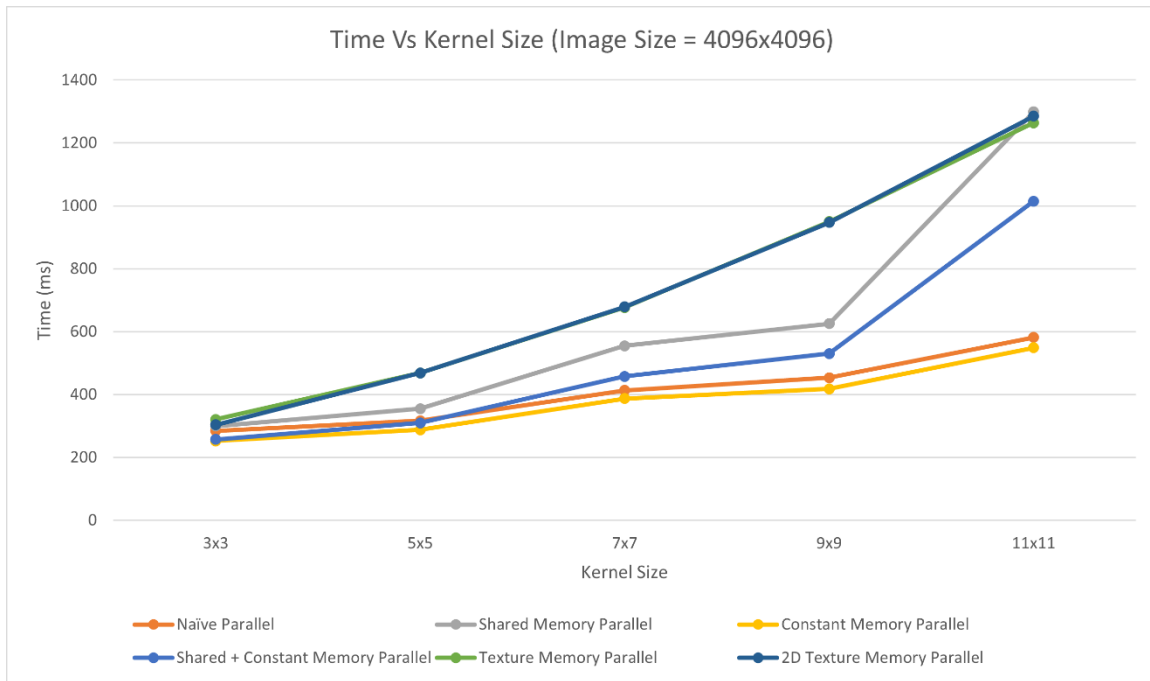


Figure 37 - Kernel size comparison for different parallel methods (4096x4096)

*Why is the shared memory implementation under-performing?*

The answer to this question is 3-fold.

- Firstly, our implementation of shared memory makes use of blocks of smaller sizes (block width – kernel width + 1). This increases the number of blocks used during the convolution process and hence leads to an increase in overall time and reduction in throughput.

- Second, Additional latency to create and transfer data from the global memory to shared memory.
- Finally, due to the requirement of convolutional kernels to access neighboring data, there are circumstances when two threads may try to access the same shared memory banks. As the kernel size increases, the overlapping regions of kernels increases and hence, bank conflicts increase, reducing throughput.

We can tackle the use of reduced width of blocks by considering kernel size. Therefore, we implement an algorithm that generates blocks of size that is inclusive of the kernel width ( $\text{block width} + \text{kernel width} - 1$ ). Therefore, when we consider the overlapping region, our block size returns to its original size. The recalculated values of execution time and speedup that have been observed and represented in appendix C and appendix D. The following graphs represents the recalculated values of shared memory implementations and its impact.

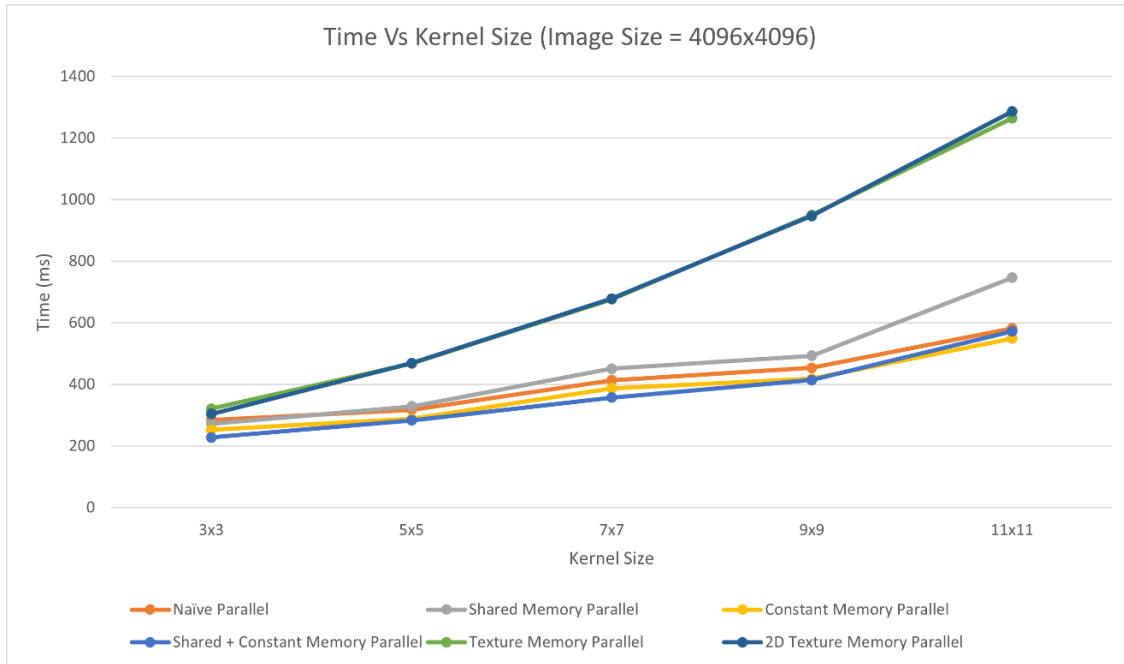


Figure 38 - Parallel implementation comparisons (4096x4096) with adjusted block width for shared memory implementation

As expected, the performance of the shared memory implementations has significantly improved. We can observe that the performance of shared and constant memory implementation has improved in terms of throughput and provides results equivalent or better (at smaller kernel sizes) to a naïve & constant memory parallel implementation.

However, this implementation adds additional restrictions on kernel size. The maximum size of thread block is 1024 thread which implies a block width of 32. This restriction must be kept in mind when performing convolution using shared memory. In appendix C, we observe that shared memory implementations of block size 24x24 and kernel size 11x11 remains empty. This is because the adjusted block size becomes 34x34 (1156 threads) which is greater than the maximum allocatable threads per block.

To overcome the limitations on block size, we develop an algorithm that aims to overcome the issues faces by considering overlapping blocks.



Figure 39 - New shared memory working

We implement the algorithm such that for every thread, we construct a 2D rectangle with a size equal to size of the block. We then assign shared memory locations at the corner of every rectangle, not inclusive of the point itself. This allows each thread to allocated at most four shared memory location. This method frees us from the limitations cause by using overlapping regions between blocks and additionally helps improve overall performance.

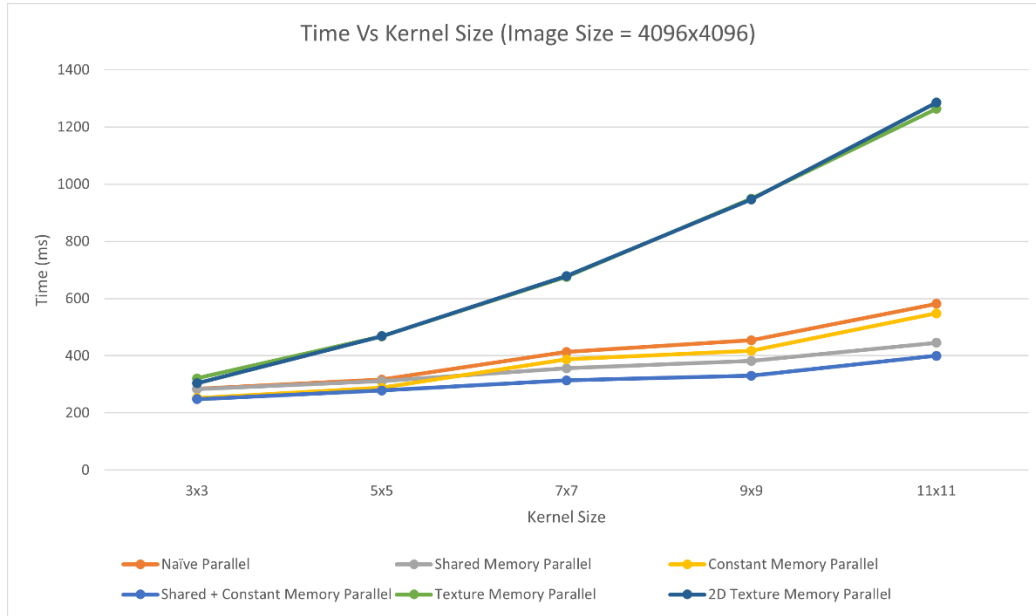


Figure 40 - Parallel implementation comparisons (4096x4096) adjusted shared memory implementation.

This results to shared memory implementations outperforming naïve and constant memory implementations. Although when compared to shared memory implementations, there exists an overhead of assigning more than one pixel per thread, this overhead is a good trade-off for the improvement in throughput of the image convolution algorithm. Furthermore, the algorithm allows to use block size of 32x32. Updated table with for shared memory implementation without overlapping blocks is given in appendix E and F.

While the shared memory implementation referred before has an effective size of 16x16, the block size still generated would be size 26x26 for a kernel of size 11. Thus, a greater number of threads will be used when compared to a standard 16x16 block.



$$\text{Improvement for kernel of size } 16 \times 16 = \frac{26 * 26}{16 * 16} = 2.64$$

This implies, shared memory implementation using overlapping memory with effective size of  $16 \times 16$  has 2.64 more threads within the block than the shared memory with no overlap implementation. We obtain a speed up of over 133 for images of size  $4096 \times 4096$  and kernel size  $11 \times 11$  when using shared with constant memory implementation over a serial implementation.

*What is the significance of the texture memory implementation?*

The texture memory has the significance of providing constant memory access. The texture memory refers to global memory with an additional cache architecture. Therefore, the additional execution time observed is because of cache misses. We observe regardless of block size or image size, the total time to perform convolution with texture memory is almost directly proportional to the size of the kernel. This is because, texture memory is optimized for threads close to another within a block to access spatially close memory locations. However, our algorithm for convolution involves having a single thread access multiple memory location that are within a spatial locality. Furthermore, due to multiple threads accessing same memory locations, the throughput of the system decreases. We would get better performance if like in [4] if our algorithm used each block to calculate the value of a single pixel.

*How is constant memory implementation providing slightly better performance?*

Constant memory is a read only memory. This memory is allocated at the beginning of program executions and its value is only changed during the lifetime of the program. Moreover, constant memory makes use of constant memory caches to improve memory access time.

### 4.1.2 Effect of image size on convolution algorithms

Table 6 - Execution time comparison based on image size.

<b>Image Size</b>	<b><i>Serial</i></b>	<b><i>Naïve Parallel</i></b>	<b><i>Shared Memory Parallel</i></b>	<b><i>Constant Memory Parallel</i></b>	<b><i>Shared + Constant Memory Parallel</i></b>	<b><i>Texture Memory Parallel</i></b>	<b><i>2D Texture Memory Parallel</i></b>
256x256	90.292	11.656	18.757	6.548	14.7	12.36	12.357
512x512	363.994	24.448	28.459	23.014	24.674	27.209	27.182
1024x1024	1460.294	38.769	46.953	36.781	38.828	46.728	46.296
2048x2048	14338.897	140.347	326.226	130.3	254.415	323.84	324.192
4096x4096	57356.285	581.864	1299.01	548.667	1015.172	1263.854	1285.623

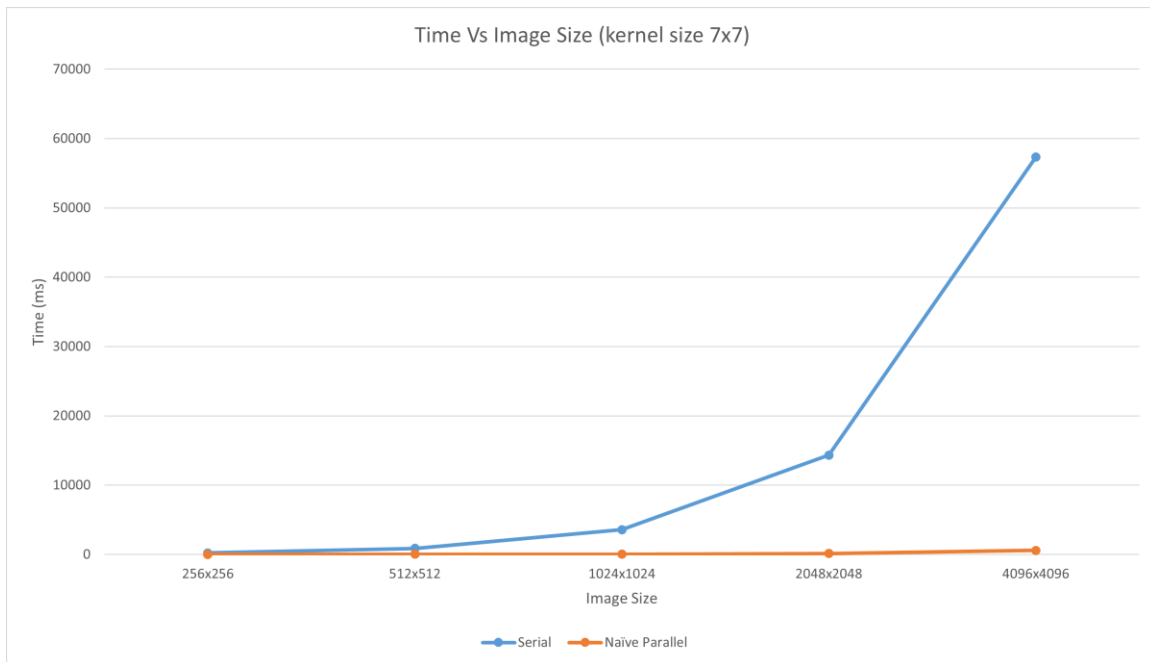


Figure 41 - Image size comparison for serial & parallel

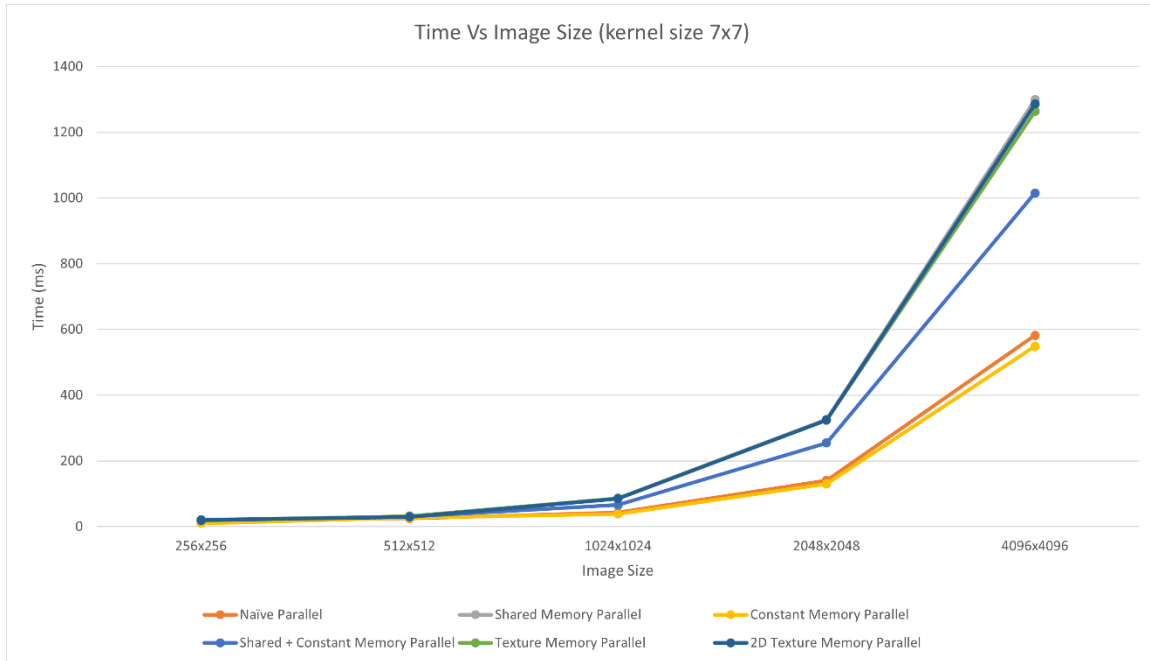


Figure 42 - Image size comparison for different parallel methods

#### 4.1.2.1 Observations

- Serial implementation execution time increases almost exponentially, while parallel implementations appear to be constant from figure 41.
- Parallel implementations resemble that of the serial implementations when the time scale is reduced.

### 4.1.3 Effect of block size on convolution algorithms

Table 7- Execution time comparison based on block size.

<b>Block Size</b>	<b><i>Naïve Parallel</i></b>	<b><i>Shared Memory Parallel</i></b>	<b><i>Constant Memory Parallel</i></b>	<b><i>Shared + Constant Memory Parallel</i></b>	<b><i>Texture Memory Parallel</i></b>	<b><i>2D Texture Memory Parallel</i></b>
<b>16x16</b>	42.608	86.116	38.732	65.934	85.434	85.063
<b>24x24</b>	39.72	51.478	38.216	42.868	86.405	87.228
<b>32x32</b>	37.19	43.77	34.697	37.396	82.831	82.851

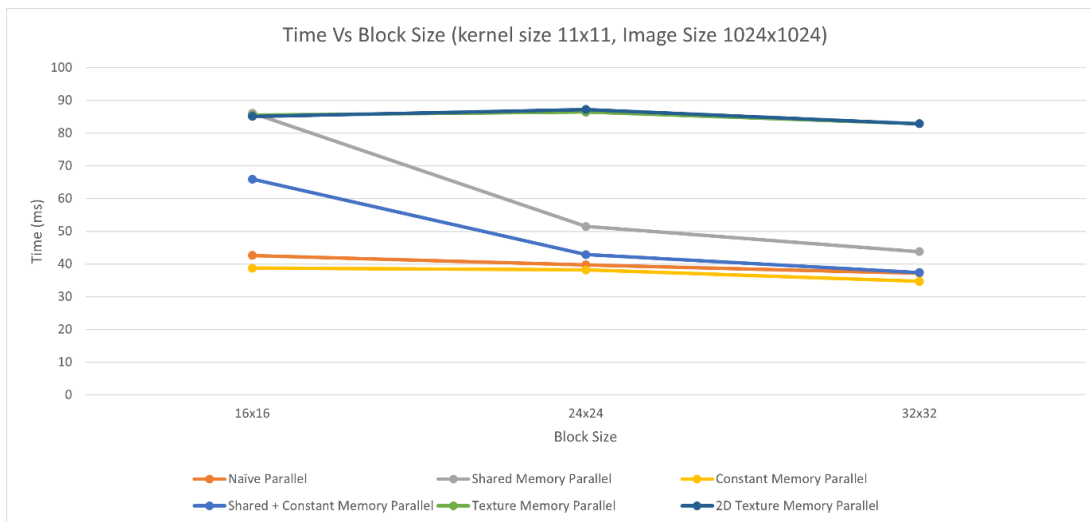


Figure 43 - Block size comparison for different parallel methods

Block size has a variant effect. This is because, as the number of threads per blocks decreases, the number of blocks increases. Whereas, to reduce the number of blocks, the number of threads per block must increase.

We can also interpret that shared memory implementations are greatly affected by block size. This is because the number of blocks into which the image is sliced into varies according to block width.

## 4.2 OpenCV Results

Table 8 - Serial Vs Parallel OpenCV convolution comparison

Image Size	Kernel Size	<i>Serial (ms)</i>	<i>CUDA (ms)</i>	<i>Speedup</i>
<b>256x256</b>	3x3	0.85	51.26	0.02
	5x5	1.99	38.49	0.05
	7x7	3.75	38.22	0.10
	9x9	13.81	38.38	0.36
	11x11	13.84	38.24	0.36
<b>512x512</b>	3x3	2.98	53.03	0.06
	5x5	7.82	51.79	0.15
	7x7	14.8	46.05	0.32
	9x9	32.86	45.24	0.73
	11x11	32.92	45	0.73
<b>1024x1024</b>	3x3	12.03	59.22	0.20
	5x5	31.23	57.44	0.54
	7x7	59.305	56.81	1.04
	9x9	96	57.31	1.68
	11x11	96.25	58.35	1.65
<b>2048x2048</b>	3x3	47.37	118.51	0.40
	5x5	125.08	108.7	1.15
	7x7	238.275	108.554	2.19
	9x9	323.742	108.8	2.98
	11x11	382.21	109.18	3.50
<b>4096x4096</b>	3x3	189.7	307.27	0.62
	5x5	500.11	306.69	1.63
	7x7	955.27	307.77	3.10
	9x9	1203.08	306.96	3.92
	11x11	1208.77	308.47	3.92

Time taken for serial implementation increases with increase in kernel width, whereas time taken for OpenCV's CUDA implementation remains almost constant regardless of kernel size. At lower image sizes, serial implementation provides significantly better performance. However, as image size increases, CUDA implementation provides consistent and better performance.

When compared to a pure CUDA C implementation, we observe that OpenCV's serial implementation is significantly more optimized than a serial implementation in CUDA C. OpenCV's CUDA implementation provides constant performance regardless of kernel size. At lower image sizes ( $<1024$ ), CUDA C provides better performance but the converse is true for images of larger sizes.



Figure 44 - OpenCV CUDA convolution

(Image Size 2048x2048 & Kernel Size 32x32)

The OpenCV CUDA convolution results however include stitching lines depending on image size. These can be avoided but the speedup would reduce from 3.5 to at most 1.35 for an image of size 2048x2048 and kernel of size 11x11. This is done by changing the user defined block size while creating the convolutional pointer.

As we can see evidently from the code snippets, OpenCV's CUDA implementation provides a much easier interface to take advantage of a machine's GPU architecture. However, The CUDA C implementation provides correct results with a speedup of 4.15 as opposed to 1.35 of OpenCV when compared to OpenCV's serial implementation.

# Chapter 5

## Conclusions and Future Work

### 5.1 Conclusions

In conclusion, convolution is an essential component of an image recognition algorithm. Any progress made to improve the efficiency of the convolution process will greatly impact the processing time of the algorithm. These advances can be made by exploiting the resources provided on the Jetson Nano GPU with help of the CUDA architecture by using a combination of methods. With the help of CUDA convolution methods, we have accomplished a speedup of up to 133 for an image of size 2048x2048 and a kernel of size 11x11 when compared to a serial implementation. We also observe that our CUDA C outperforms OpenCV's CUDA implementation with good accuracy and good speedup. Similar methods can be used across different sections of a CNN to improve its throughput and thereby improve image recognition algorithms.

### 5.2 Recommendation in Future Work

- Conduct performance analysis for other CUDA interfaces.
- Find optimizations for other parts of CNN such as max pooling.

# References

- [1] S. Choi and K. Lee, "A CUDA-based implementation of convolutional neural network," *2017 4th International Conference on Computer Applications and Information Processing Technology (CAIPT)*, Kuta Bali, 2017, pp. 1-4.
- [2] B. Shi, S. Chen, F. Huang, C. Wang and K. Bi, "The Parallel Processing Based on CUDA for Convolution Filter FDK Reconstruction of CT," *2010 3rd International Symposium on Parallel Architectures, Algorithms and Programming*, Dalian, 2010, pp. 149-153.
- [3] L. M. Russo, E. C. Pedrino, E. Kato and V. O. Roda, "Image convolution processing: A GPU versus FPGA comparison," *2012 VIII Southern Conference on Programmable Logic*, Bento Goncalves, 2012, pp. 1-6.
- [4] L. Feng, D. Zheng and J. Yu, "CUDA Optimization Method for Activation Function in Convolution Operation," *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCLOUD/SocialCom/SustainCom)*, Xiamen, China, 2019, pp. 519-525.
- [5] E. Cervera, "GPU-Accelerated Vision for Robots: Improving System Throughput Using OpenCV and CUDA," in *IEEE Robotics & Automation Magazine*, vol. 27, no. 2, pp. 151-158, June 2020.
- [6] D. Vintache, B. Humbert and D. Brasse, "Iterative reconstruction for transmission tomography on GPU using Nvidia CUDA," in *Tsinghua Science and Technology*, vol. 15, no. 1, pp. 11-16, Feb. 2010.
- [7] CUDA C++ programming guide. (2006). Retrieved March 20, 2021, from <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [8] NVIDIA cuDNN. (2021, January 28). Retrieved March 21, 2021, from <https://developer.nvidia.com/cudnn>
- [9] "NVIDIA TensorRT." NVIDIA Developer, 11 Feb. 2021, <https://developer.nvidia.com/tensorrt>.
- [10] Computer vision: What it is and why it matters. (n.d.). Retrieved March 20, 2021, from [https://www.sas.com/en\\_us/insights/analytics/computer-vision.html](https://www.sas.com/en_us/insights/analytics/computer-vision.html).
- [11] Dataman, D. (2020, December 07). What is image recognition? Retrieved March 15, 2021, from <https://medium.com/dataman-in-ai/module-6-image-recognition-for-insurance-claim-handling-part-i-a338d16c9de0>
- [12] Jetson nano developer kit. (2021, January 28). Retrieved March 16, 2021, from <https://developer.nvidia.com/embedded/jetson-nano-developer-kit>.
- [13] Getting started with jetson nano developer kit. (2021, March 11). Retrieved March 15, 2021, from <https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit>.



- [14] Connect to wifi network through ubuntu terminal. Retrieved March 16, 2021, from <https://askubuntu.com/questions/294257/connect-to-wifi-network-through-ubuntu-terminal>
- [15] NVIDIA shield TV 4k hdr. (n.d.). Retrieved March 17, 2021, from <https://www.nvidia.com/en-us/shield/>
- [16] NVIDIA. (2020, August 25). CUDA refresher: The CUDA programming model. Retrieved March 15, 2021, from <https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/>
- [17] Initialization and information. (n.d.). Retrieved March 21, 2021, from [https://docs.opencv.org/3.4/d8/d40/group\\_\\_cudacore\\_\\_init.html](https://docs.opencv.org/3.4/d8/d40/group__cudacore__init.html)
- [18] Embedded Boards & Systems (n.d.). Retrieved March 15, 2021, from <https://www.avnet.com/shop/us/c/embedded-boards-systems/>

# Appendix

## Appendix A

Table 9 - Table with execution time for different convolution implementation styles

Block Dim	Image Size	Kernel Size	Serial	Naïve Parallel	Shared Memory Parallel	Constant Memory Parallel	Shared + Constant Memory Parallel	Texture Memory Parallel	2D Texture Memory Parallel
16x16	256x256	3x3	19.441	5.84	7.181	3.312	4.505	4.652	4.183
		5x5	49.861	7.34	9.653	4.513	8.064	7.699	7.71
		7x7	90.292	11.656	18.757	6.548	14.7	12.36	12.357
		9x9	143.523	13.371	21.877	7.172	17.971	18.419	18.45
		11x11	208.978	19.07	19.296	10.128	20.972	18.558	19.003
	512x512	3x3	75.408	19.588	12.78	7.8	9.991	11.521	10.495
		5x5	197.797	16.741	20.109	14.477	16.147	24.238	24.787
		7x7	363.994	24.448	28.459	23.014	24.674	27.209	27.182
		9x9	588.575	22.005	25.262	26.221	25.703	29.397	32.191
		11x11	864.658	24.921	28.228	27.175	30.296	31.767	29.996
	1024x1024	3x3	299.043	35.054	25.579	21.29	22.179	26.244	26.819
		5x5	799.245	35.986	37.061	32.964	35.688	39.628	41.157
		7x7	1460.294	38.769	46.953	36.781	38.828	46.728	46.296
		9x9	2396.45	41.632	43.356	34.545	35.815	62.863	63.199
		11x11	3571.452	42.608	86.116	38.732	65.934	85.434	85.063
	2048x2048	3x3	1220.997	62.687	69.035	58.777	59.734	73.485	74.896
		5x5	3356.846	65.346	82.956	66.708	71.283	111.289	111.302
		7x7	6241.395	92.023	134.257	90.494	106.825	167.49	167.318
		9x9	9879.396	105.607	154.041	97.149	125.696	238.044	237.523
		11x11	14338.897	140.347	326.226	130.3	254.415	323.84	324.192
	4096x4096	3x3	5588.372	284.51	299.446	252.413	257.238	320.966	303.842
		5x5	13935.337	317.156	354.913	287.956	310.331	468.274	468.397
		7x7	25023.135	413.184	555.188	387.354	457.864	676.534	678.619
		9x9	39502.422	454.032	625.457	417.678	530.631	949.636	947.247
		11x11	57356.285	581.864	1299.01	548.667	1015.172	1263.854	1285.623
24x24	256x256	3x3	19.25	5.857	6.763	3.355	3.301	4.663	4.002
		5x5	50.001	7.606	9.119	4.241	4.413	8.041	8.082
		7x7	90.404	11.836	14.987	6.38	6.44	12.991	13.082
		9x9	143.294	14.358	17.024	7.044	7.54	19.57	19.499
		11x11	208.328	20.503	17.312	10.102	12.102	20.226	19.41
	512x512	3x3	75.184	19.063	10.843	7.218	7.655	11.952	8.845
		5x5	197.103	27.096	18.337	13.75	14.175	21.901	22.013
		7x7	359.901	24.866	21.25	22.522	21.872	28.219	24.666
		9x9	583.594	22.713	24.407	25.065	19.955	27.431	26.056
		11x11	849.478	23.228	26.295	22.987	26.775	26.496	27.247
	1024x1024	3x3	299.334	32.187	22.103	17.986	20.879	26.629	26.465
		5x5	797.717	34.081	34.523	29.905	34.488	39.491	39.033
		7x7	1456.811	36.067	36.808	34.71	37.282	46.958	47.72
		9x9	2395.488	34.79	35.342	34.019	34.275	64.541	65.068
		11x11	3574.074	39.72	51.478	38.216	42.868	86.405	87.228
	2048x2048	3x3	1215.714	76.38	68.785	62.169	59.667	77.254	77.505
		5x5	3347.842	74.367	81.44	70.656	70.737	115.882	115.877
		7x7	6224	99.994	110.182	93.631	89.168	172.896	175.947
		9x9	9855.117	112.727	126.462	99.224	104.171	245.411	247.42
		11x11	14318.065	149.729	204.634	135.144	158.983	333.184	336.151
	4096x4096	3x3	5604.938	283.295	334.31	295.605	291.544	331.776	332.333
		5x5	13874.678	322.506	390.105	339.047	341.3	486.904	483.718
		7x7	25054.182	426.451	499.259	440.226	421.063	702.671	702.852
		9x9	39479.602	475.103	558.264	463.039	476.909	973.57	975.996
		11x11	57337.051	637.999	853.983	598.697	680.945	1312.911	1321.027
32x32	256x256	3x3	19.278	5.699	6.603	3.846	5.13	7.152	3.751
		5x5	49.813	7.202	9.252	7.012	7.427	13.323	7.566
		7x7	90.326	11.351	13.485	11.053	10.153	22.83	12.315
		9x9	143.339	12.893	16.174	12.282	12.768	18.592	18.177
		11x11	208.614	19.13	15.817	18.374	18.561	18.351	17.925
	512x512	3x3	75.156	18.51	10.594	7.335	8.576	12.59	8.777
		5x5	197.046	25.503	17.634	13.304	13.858	20.654	21.821
		7x7	359.888	24.261	20.927	21.8	20.221	25.662	25.858
		9x9	585.076	26.455	22.888	24.724	25.38	29.373	28.615
		11x11	849.793	24.926	25.694	25.541	25.871	28.549	27.073
	1024x1024	3x3	299.407	30.382	21.542	17.783	18.163	26.277	26.457
		5x5	799.383	32.051	31.281	30.258	29.853	38.409	37.728
		7x7	1456.731	37.045	36.269	35.103	33.694	45.603	45.64
		9x9	2398.076	35.247	34.35	31.84	36.308	61.787	61.899
		11x11	3577.877	37.19	43.77	34.697	37.396	82.831	82.851
	2048x2048	3x3	1215.928	68.256	67.265	62.2	60.703	74.503	75.774
		5x5	3344.906	73.058	78.308	70.536	69.424	111.012	111.117
		7x7	6217.988	97.134	105.562	93.166	85.617	166.8	167.44
		9x9	9853.688	106.171	118.876	99.1	98.07	236.722	235.226
		11x11	14313.627	142.615	171.994	134.384	129.324	320.517	319.989
	4096x4096	3x3	5610.314	283.159	319.587	300.769	292.308	333.626	319.221
		5x5	13883.563	344.193	367.635	337.71	331.043	475.14	463.14
		7x7	24993.002	448.242	478.183	435.008	397.34	670.511	666.386
		9x9	39547.781	480.604	525.686	460.759	443.522	935.217	931.51
		11x11	57306.035	616.089	724.562	594.588	557.783	1264.485	1262.321

# Appendix B

Table 10 - Table with speedup of different convolution implementation styles

Block Dim	Image Size	Kernel Size	Serial	Naive Parallel	Shared Memory Parallel	Constant Memory Parallel	Shared + Constant Memory Parallel	Texture Memory Parallel	2D Texture Memory Parallel
16x16	256x256	3x3	1	3.33	2.71	5.87	4.32	4.18	4.65
		5x5	1	6.79	5.17	11.05	6.18	6.48	6.47
		7x7	1	7.75	4.81	13.79	6.14	7.31	7.31
		9x9	1	10.73	6.56	20.01	7.99	7.79	7.78
		11x11	1	10.96	10.83	20.63	9.96	11.26	11.00
	512x512	3x3	1	3.85	5.90	9.67	7.55	6.55	7.19
		5x5	1	11.82	9.84	13.66	12.25	8.16	7.98
		7x7	1	14.89	12.79	15.82	14.75	13.38	13.39
		9x9	1	26.75	23.30	22.45	22.90	20.02	18.28
		11x11	1	34.70	30.63	31.82	28.54	27.22	28.83
	1024x1024	3x3	1	8.53	11.69	14.05	13.48	11.39	11.15
		5x5	1	22.21	21.57	24.25	22.40	20.17	19.42
		7x7	1	37.67	31.10	39.70	37.61	31.25	31.54
		9x9	1	57.56	55.27	69.37	66.91	38.12	37.92
		11x11	1	83.82	41.47	92.21	54.17	41.80	41.99
	2048x2048	3x3	1	19.48	17.69	20.77	20.44	16.62	16.30
		5x5	1	51.37	40.47	50.32	47.09	30.16	30.16
		7x7	1	67.82	46.49	68.97	58.43	37.26	37.30
		9x9	1	93.55	64.13	101.69	78.60	41.50	41.59
		11x11	1	102.17	43.95	110.05	56.36	44.28	44.23
	4096x4096	3x3	1	19.64	18.66	22.14	21.72	17.41	18.39
		5x5	1	43.94	39.26	48.39	44.90	29.76	29.75
		7x7	1	60.56	45.07	64.60	54.65	36.99	36.87
		9x9	1	87.00	63.16	94.58	74.44	41.60	41.70
		11x11	1	98.57	44.15	104.54	56.50	45.38	44.61
24x24	256x256	3x3	1	3.29	2.85	5.74	5.83	4.13	4.81
		5x5	1	6.57	5.48	11.79	11.33	6.22	6.19
		7x7	1	7.64	6.03	14.17	14.04	6.96	6.91
		9x9	1	9.98	8.42	20.34	19.00	7.32	7.35
		11x11	1	10.16	12.03	20.62	17.21	10.30	10.73
	512x512	3x3	1	3.94	6.93	10.42	9.82	6.29	8.50
		5x5	1	7.27	10.75	14.33	13.90	9.00	8.95
		7x7	1	14.47	16.94	15.98	16.45	12.75	14.59
		9x9	1	25.69	23.91	23.28	29.25	21.27	22.40
		11x11	1	36.57	32.31	36.95	31.73	32.06	31.18
	1024x1024	3x3	1	9.30	13.54	16.64	14.34	11.24	11.31
		5x5	1	23.41	23.11	26.68	23.13	20.20	20.44
		7x7	1	40.39	39.58	41.97	39.08	31.02	30.53
		9x9	1	68.86	67.78	70.42	69.89	37.12	36.82
		11x11	1	89.98	69.43	93.52	83.37	41.36	40.97
	2048x2048	3x3	1	15.92	17.67	19.55	20.37	15.74	15.69
		5x5	1	45.02	41.11	47.38	47.33	28.89	28.89
		7x7	1	62.24	56.49	66.47	69.80	36.00	35.37
		9x9	1	87.42	77.93	99.32	94.61	40.16	39.83
		11x11	1	95.63	69.97	105.95	90.06	42.97	42.59
	4096x4096	3x3	1	19.78	16.77	18.96	19.23	16.89	16.87
		5x5	1	43.02	35.57	40.92	40.65	28.50	28.68
		7x7	1	58.75	50.18	56.91	59.50	35.66	35.65
		9x9	1	83.10	70.72	85.26	82.78	40.55	40.45
		11x11	1	89.87	67.14	95.77	84.20	43.67	43.40
32x32	256x256	3x3	1	3.38	2.92	5.01	3.76	2.70	5.14
		5x5	1	6.92	5.38	7.10	6.71	3.74	6.58
		7x7	1	7.96	6.70	8.17	8.90	3.96	7.33
		9x9	1	11.12	8.86	11.67	11.23	7.71	7.89
		11x11	1	10.91	13.19	11.35	11.24	11.37	11.64
	512x512	3x3	1	4.06	7.09	10.25	8.76	5.97	8.56
		5x5	1	7.73	11.17	14.81	14.22	9.54	9.03
		7x7	1	14.83	17.20	16.51	17.80	14.02	13.92
		9x9	1	22.12	25.56	23.66	23.05	19.92	20.45
		11x11	1	34.09	33.07	33.27	32.85	29.77	31.39
	1024x1024	3x3	1	9.85	13.90	16.84	16.48	11.39	11.32
		5x5	1	24.94	25.55	26.42	26.78	20.81	21.19
		7x7	1	39.32	40.16	41.50	43.23	31.94	31.92
		9x9	1	68.04	69.81	75.32	66.05	38.81	38.74
		11x11	1	96.21	81.74	103.12	95.68	43.19	43.18
	2048x2048	3x3	1	17.81	18.08	19.55	20.03	16.32	16.05
		5x5	1	45.78	42.71	47.42	48.18	30.13	30.10
		7x7	1	64.01	58.90	66.74	72.63	37.28	37.14
		9x9	1	92.81	82.89	99.43	100.48	41.63	41.89
		11x11	1	100.37	83.22	106.51	110.68	44.66	44.73
	4096x4096	3x3	1	19.81	17.55	18.65	19.19	16.82	17.58
		5x5	1	40.34	37.76	41.11	41.94	29.22	29.98
		7x7	1	55.76	52.27	57.45	62.90	37.27	37.51
		9x9	1	82.29	75.23	85.83	89.17	42.29	42.46
		11x11	1	93.02	79.09	96.38	102.74	45.32	45.40

# Appendix C

Table 11 - Table with execution time for shared memory implementations with updated block size value

Block Dim	Image Size	Kernel Size	Serial	Naive Parallel	Shared Memory Parallel	Constant Memory Parallel	Shared + Constant Memory Parallel	Texture Memory Parallel	2D Texture Memory Parallel
16x16	256x256	3x3	19.441	5.84	6.548	3.312	3.858	4.652	4.183
		5x5	49.861	7.34	9.191	4.513	7.636	7.699	7.71
		7x7	90.292	11.656	14.69	6.548	11.001	12.36	12.357
		9x9	143.523	13.371	16.908	7.172	13.605	18.419	18.45
		11x11	208.978	19.07	17.592	10.128	20.965	18.558	19.003
	512x512	3x3	75.408	19.588	22.979	7.8	7.363	11.521	10.495
		5x5	197.797	16.741	18.929	14.477	14.789	24.238	24.787
		7x7	363.994	24.448	24.316	23.014	21.778	27.209	27.182
		9x9	588.575	22.005	24.398	26.221	20.149	29.397	32.191
		11x11	864.658	24.921	27.147	27.175	22.64	31.767	29.996
	1024x1024	3x3	299.043	35.054	29.083	21.29	17.755	26.244	26.819
		5x5	799.245	35.986	35.353	32.964	32.304	39.628	41.157
		7x7	1460.294	38.769	36.398	36.781	32.628	46.728	46.296
		9x9	2396.45	41.632	34.587	34.545	31.475	62.863	63.199
		11x11	3571.452	42.608	48.67	38.732	37.41	85.434	85.063
	2048x2048	3x3	1220.997	62.687	69.641	58.777	59.637	73.485	74.896
		5x5	3356.846	65.346	82.969	66.708	71.234	111.289	111.302
		7x7	6241.395	92.023	114.42	90.494	90.086	167.49	167.318
		9x9	9879.396	105.607	125.606	97.149	103.984	238.044	237.523
		11x11	14338.897	140.347	193.964	130.3	144.92	323.84	324.192
	4096x4096	3x3	5588.372	284.51	272.124	252.413	227.738	320.966	303.842
		5x5	13935.337	317.156	328.233	287.956	282.912	468.274	468.397
		7x7	25023.135	413.184	451.279	387.354	357.394	676.534	678.619
		9x9	39502.422	454.032	492.492	417.678	413.713	949.636	947.247
		11x11	57356.285	581.864	746.505	548.667	572.056	1263.854	1285.623
24x24	256x256	3x3	19.25	5.857	7.297	3.355	5.216	4.663	4.002
		5x5	50.001	7.606	9.611	4.241	7.294	8.041	8.082
		7x7	90.404	11.836	14.799	6.38	10.893	12.991	13.082
		9x9	143.294	14.358	16.105	7.044	12.652	19.57	19.499
		11x11	208.328	20.503		10.102		20.226	19.41
	512x512	3x3	75.184	19.063	26.188	7.218	17.406	11.952	8.845
		5x5	197.103	27.096	21.112	13.75	23.72	21.901	22.013
		7x7	359.901	24.866	22.096	22.522	21.894	28.219	24.666
		9x9	583.594	22.713	23.467	25.065	25.8	27.431	26.056
		11x11	849.478	23.228		22.987		26.496	27.247
	1024x1024	3x3	299.334	32.187	35.671	17.986	28.578	26.629	26.465
		5x5	797.717	34.081	34.784	29.905	31.4	39.491	39.033
		7x7	1456.811	36.067	35.464	34.71	34.328	46.958	47.72
		9x9	2395.488	34.79	34.13	34.019	31.846	64.541	65.068
		11x11	3574.074	39.72		38.216		86.405	87.228
	2048x2048	3x3	1215.714	76.38	72.798	62.169	68.388	77.254	77.505
		5x5	3347.842	74.367	84.195	70.656	70.529	115.882	115.877
		7x7	6224	99.994	111.453	93.631	88.737	172.896	175.947
		9x9	9855.117	112.727	117.822	99.224	97.739	245.411	247.42
		11x11	14318.065	149.729		135.144		333.184	336.151
	4096x4096	3x3	5604.938	283.295	303.216	295.605	252.759	331.776	332.333
		5x5	13874.678	322.506	343.535	339.047	290.315	486.904	483.718
		7x7	25054.182	426.451	452.955	440.226	369.469	702.671	702.852
		9x9	39479.602	475.103	478.603	463.039	408.613	973.57	975.996
		11x11	57337.051	637.999		598.697		1312.911	1321.027

# Appendix D

Table 12 - Table with Speedup for shared memory implementations with updated block size value

Block Dim	Image Size	Kernel Size	Serial	Naïve Parallel	Shared Memory Parallel	Constant Memory Parallel	Shared + Constant Memory Parallel	Texture Memory Parallel	2D Texture Memory Parallel
16x16	256x256	3x3	1.00	3.33	2.97	5.87	5.04	4.18	4.65
		5x5	1.00	6.79	5.42	11.05	6.53	6.48	6.47
		7x7	1.00	7.75	6.15	13.79	8.21	7.31	7.31
		9x9	1.00	10.73	8.49	20.01	10.55	7.79	7.78
		11x11	1.00	10.96	11.88	20.63	9.97	11.26	11.00
	512x512	3x3	1.00	3.85	3.28	9.67	10.24	6.55	7.19
		5x5	1.00	11.82	10.45	13.66	13.37	8.16	7.98
		7x7	1.00	14.89	14.97	15.82	16.71	13.38	13.39
		9x9	1.00	26.75	24.12	22.45	29.21	20.02	18.28
		11x11	1.00	34.70	31.85	31.82	38.19	27.22	28.83
	1024x1024	3x3	1.00	8.53	10.28	14.05	16.84	11.39	11.15
		5x5	1.00	22.21	22.61	24.25	24.74	20.17	19.42
		7x7	1.00	37.67	40.12	39.70	44.76	31.25	31.54
		9x9	1.00	57.56	69.29	69.37	76.14	38.12	37.92
		11x11	1.00	83.82	73.38	92.21	95.47	41.80	41.99
	2048x2048	3x3	1.00	19.48	17.53	20.77	20.47	16.62	16.30
		5x5	1.00	51.37	40.46	50.32	47.12	30.16	30.16
		7x7	1.00	67.82	54.55	68.97	69.28	37.26	37.30
		9x9	1.00	93.55	78.65	101.69	95.01	41.50	41.59
		11x11	1.00	102.17	73.93	110.05	98.94	44.28	44.23
	4096x4096	3x3	1.00	19.64	20.54	22.14	24.54	17.41	18.39
		5x5	1.00	43.94	42.46	48.39	49.26	29.76	29.75
		7x7	1.00	60.56	55.45	64.60	70.02	36.99	36.87
		9x9	1.00	87.00	80.21	94.58	95.48	41.60	41.70
		11x11	1.00	98.57	76.83	104.54	100.26	45.38	44.61
24x24	256x256	3x3	1.00	3.29	2.64	5.74	3.69	4.13	4.81
		5x5	1.00	6.57	5.20	11.79	6.86	6.22	6.19
		7x7	1.00	7.64	6.11	14.17	8.30	6.96	6.91
		9x9	1.00	9.98	8.90	20.34	11.33	7.32	7.35
		11x11	1.00	10.16		20.62		10.30	10.73
	512x512	3x3	1.00	3.94	2.87	10.42	4.32	6.29	8.50
		5x5	1.00	7.27	9.34	14.33	8.31	9.00	8.95
		7x7	1.00	14.47	16.29	15.98	16.44	12.75	14.59
		9x9	1.00	25.69	24.87	23.28	22.62	21.27	22.40
		11x11	1.00	36.57		36.95		32.06	31.18
	1024x1024	3x3	1.00	9.30	8.39	16.64	10.47	11.24	11.31
		5x5	1.00	23.41	22.93	26.68	25.41	20.20	20.44
		7x7	1.00	40.39	41.08	41.97	42.44	31.02	30.53
		9x9	1.00	68.86	70.19	70.42	75.22	37.12	36.82
		11x11	1.00	89.98		93.52		41.36	40.97
	2048x2048	3x3	1.00	15.92	16.70	19.55	17.78	15.74	15.69
		5x5	1.00	45.02	39.76	47.38	47.47	28.89	28.89
		7x7	1.00	62.24	55.84	66.47	70.14	36.00	35.37
		9x9	1.00	87.42	83.64	99.32	100.83	40.16	39.83
		11x11	1.00	95.63	#DIV/0!	105.95	#DIV/0!	42.97	42.59
	4096x4096	3x3	1.00	19.78	18.48	18.96	22.18	16.89	16.87
		5x5	1.00	43.02	40.39	40.92	47.79	28.50	28.68
		7x7	1.00	58.75	55.31	56.91	67.81	35.66	35.65
		9x9	1.00	83.10	82.49	85.26	96.62	40.55	40.45
		11x11	1.00	89.87		95.77		43.67	43.40

# Appendix E

Table 13 - Table with execution time for share memory implementations with no overlapping between blocks

Block Dim	Image Size	Kernel Size	Serial	Naïve Parallel	Shared Memory Parallel	Constant Memory Parallel	Shared + Constant Memory Parallel	Texture Memory Parallel	2D Texture Memory Parallel
16x16	256x256	3x3	19.441	5.84	5.438	3.312	4.932	4.652	4.183
		5x5	49.861	7.34	7.183	4.513	6.422	7.699	7.71
		7x7	90.292	11.656	8.768	6.548	8.673	12.36	12.357
		9x9	143.523	13.371	10.108	7.172	9.538	18.419	18.45
		11x11	208.978	19.07	13.125	10.128	12.533	18.558	19.003
	512x512	3x3	75.408	19.588	18.356	7.8	9.16	11.521	10.495
		5x5	197.797	16.741	23.804	14.477	11.868	24.238	24.787
		7x7	363.994	24.448	18.591	23.014	15.366	27.209	27.182
		9x9	588.575	22.005	21.144	26.221	18.433	29.397	32.191
		11x11	864.658	24.921	26.621	27.175	23.265	31.767	29.996
	1024x1024	3x3	299.043	35.054	36.643	21.29	21.208	26.244	26.819
		5x5	799.245	35.986	36.369	32.964	32.244	39.628	41.157
		7x7	1460.294	38.769	35.679	36.781	35.459	46.728	46.296
		9x9	2396.45	41.632	36.41	34.545	35.495	62.863	63.199
		11x11	3571.452	42.608	36.289	38.732	35.355	85.434	85.063
	2048x2048	3x3	1220.997	62.687	62.658	58.777	55.827	73.485	74.896
		5x5	3356.846	65.346	68.737	66.708	63.523	111.289	111.302
		7x7	6241.395	92.023	80.855	90.494	70.481	167.49	167.318
		9x9	9879.396	105.607	87.284	97.149	78.589	238.044	237.523
		11x11	14338.897	140.347	104.236	130.3	92.143	323.84	324.192
	4096x4096	3x3	5588.372	284.51	283.468	252.413	248.443	320.966	303.842
		5x5	13935.337	317.156	310.968	287.956	278.139	468.274	468.397
		7x7	25023.135	413.184	356.321	387.354	314.056	678.619	678.619
		9x9	39502.422	454.032	381.759	417.678	330.522	949.636	947.247
		11x11	57356.285	581.864	445.37	548.667	400.262	1263.854	1285.623
24x24	256x256	3x3	19.25	5.857	5.439	3.355	4.721	4.663	4.002
		5x5	50.001	7.606	6.848	4.241	6.12	8.041	8.082
		7x7	90.404	11.836	8.817	6.38	7.945	12.991	13.082
		9x9	143.294	14.358	10.117	7.044	9.504	19.57	19.499
		11x11	208.328	20.503	12.887	10.102	12.097	20.226	19.41
	512x512	3x3	75.184	19.063	18.271	7.218	9.563	11.952	8.845
		5x5	197.103	27.096	24.25	13.75	12.265	21.901	22.013
		7x7	359.901	24.866	21.243	22.522	16.021	28.219	24.666
		9x9	583.594	22.713	21.892	25.065	19.52	27.431	26.056
		11x11	849.478	23.228	28.39	22.987	24.74	26.496	27.247
	1024x1024	3x3	299.334	32.187	32.575	17.986	21.493	26.629	26.465
		5x5	797.717	34.081	34.416	29.905	33.972	39.491	39.033
		7x7	1456.811	36.067	35.749	34.71	34.334	46.958	47.72
		9x9	2395.488	34.79	36.924	34.019	34.131	64.541	65.068
		11x11	3574.074	39.72	36.763	38.216	36.317	86.405	87.228
	2048x2048	3x3	1215.714	76.38	77.984	62.169	63.602	77.254	77.505
		5x5	3347.842	74.367	78.516	70.656	72.277	115.882	115.877
		7x7	6224	99.994	88.212	93.631	77.595	172.896	175.947
		9x9	9855.117	112.727	94.058	99.224	86.169	245.411	247.42
		11x11	14318.065	149.729	110.438	135.144	100.282	333.184	336.151
	4096x4096	3x3	5604.938	283.295	312.093	295.605	282.771	331.776	332.333
		5x5	13874.678	322.506	344.854	339.047	312.837	486.904	483.718
		7x7	25054.182	426.451	391.519	440.226	351.009	702.671	702.852
		9x9	39479.602	475.103	415.94	463.039	383.764	973.57	975.996
		11x11	57337.051	637.999	477.838	598.697	428.98	1312.911	1321.027
32x32	256x256	3x3	19.278	5.699	5.656	3.846	4.873	7.152	3.751
		5x5	49.813	7.202	6.966	7.012	6.333	13.323	7.566
		7x7	90.326	11.351	8.971	11.053	8.12	22.83	12.315
		9x9	143.339	12.893	10.33	12.282	10.206	18.592	18.177
		11x11	208.614	19.13	13.026	18.374	12.524	18.351	17.925
	512x512	3x3	75.156	18.51	19.448	7.335	9.499	12.59	8.777
		5x5	197.046	25.503	24.744	13.304	12.174	20.654	21.821
		7x7	359.888	24.261	20.225	21.8	16.072	25.662	25.858
		9x9	585.076	26.455	21.568	24.724	19.049	29.373	28.615
		11x11	849.793	24.926	27.1	25.541	23.67	28.549	27.073
	1024x1024	3x3	299.407	30.382	31.093	17.783	21.816	26.277	26.457
		5x5	799.383	32.051	32.679	30.258	33.234	38.409	37.728
		7x7	1456.731	37.045	35.338	35.103	36.115	45.603	45.64
		9x9	2398.076	35.247	35.955	31.84	36.332	61.787	61.899
		11x11	3577.877	37.19	36.874	34.697	35.069	82.831	82.851
	2048x2048	3x3	1215.928	68.256	67.265	56.52	60.703	74.503	75.774
		5x5	3344.906	73.058	78.308	63.1	69.424	111.012	111.117
		7x7	6217.988	97.134	105.562	71.436	85.617	166.8	167.44
		9x9	9853.688	106.171	118.876	79.061	98.07	236.722	235.226
		11x11	14313.627	142.615	171.994	92.198	129.324	320.517	319.989
	4096x4096	3x3	5610.314	283.159	312.11	300.769	287.13	333.626	319.221
		5x5	13883.563	344.193	339.741	337.71	315.953	475.14	463.14
		7x7	24993.002	448.242	381.842	435.008	350.256	670.511	666.386
		9x9	39547.781	480.604	411.655	460.759	383.705	935.217	931.51
		11x11	57306.035	616.089	475.199	594.588	430.43	1264.485	1262.321

# Appendix F

Table 14 - Table with Speedup for share memory implementations with no overlapping between blocks

Block Dim	Image Size	Kernel Size	Serial	Naïve Parallel	Shared Memory Parallel	Constant Memory Parallel	Shared + Constant Memory Parallel	Texture Memory Parallel	2D Texture Memory Parallel
16x16	256x256	3x3	1.00	3.33	3.58	5.87	3.94	4.18	4.65
		5x5	1.00	6.79	6.94	11.05	7.76	6.48	6.47
		7x7	1.00	7.75	10.30	13.79	10.41	7.31	7.31
		9x9	1.00	10.73	14.20	20.01	15.05	7.79	7.78
		11x11	1.00	10.96	15.92	20.63	16.67	11.26	11.00
	512x512	3x3	1.00	3.85	4.11	9.67	8.23	6.55	7.19
		5x5	1.00	11.82	8.31	13.66	16.67	8.16	7.98
		7x7	1.00	14.89	19.58	15.82	23.69	13.38	13.39
		9x9	1.00	26.75	27.84	22.45	31.93	20.02	18.28
		11x11	1.00	34.70	32.48	31.82	37.17	27.22	28.83
	1024x1024	3x3	1.00	8.53	8.16	14.05	14.10	11.39	11.15
		5x5	1.00	22.21	21.98	24.25	24.79	20.17	19.42
		7x7	1.00	37.67	40.93	39.70	41.18	31.25	31.54
		9x9	1.00	57.56	65.82	69.37	67.52	38.12	37.92
		11x11	1.00	83.82	98.42	92.21	101.02	41.80	41.99
	2048x2048	3x3	1.00	19.48	19.49	20.77	21.87	16.62	16.30
		5x5	1.00	51.37	48.84	50.32	52.84	30.16	30.16
		7x7	1.00	67.82	77.19	68.97	88.55	37.26	37.30
		9x9	1.00	93.55	113.19	101.69	125.71	41.50	41.59
		11x11	1.00	102.17	137.56	110.05	155.62	44.28	44.23
	4096x4096	3x3	1.00	19.64	19.71	22.14	22.49	17.41	18.39
		5x5	1.00	43.94	44.81	48.39	50.10	29.76	29.75
		7x7	1.00	60.56	70.23	64.60	79.68	36.99	36.87
		9x9	1.00	87.00	103.47	94.58	119.52	41.60	41.70
		11x11	1.00	98.57	128.78	104.54	143.30	45.38	44.61
24x24	256x256	3x3	1.00	3.29	3.54	5.74	4.08	4.13	4.81
		5x5	1.00	6.57	7.30	11.79	8.17	6.22	6.19
		7x7	1.00	7.64	10.25	14.17	11.38	6.96	6.91
		9x9	1.00	9.98	14.16	20.34	15.08	7.32	7.35
		11x11	1.00	10.16	16.17	20.62	17.22	10.30	10.73
	512x512	3x3	1.00	3.94	4.11	10.42	7.86	6.29	8.50
		5x5	1.00	7.27	8.13	14.33	16.07	9.00	8.95
		7x7	1.00	14.47	16.94	15.98	22.46	12.75	14.59
		9x9	1.00	25.69	26.66	23.28	29.90	21.27	22.40
		11x11	1.00	36.57	29.92	36.95	34.34	32.06	31.18
	1024x1024	3x3	1.00	9.30	9.19	16.64	13.93	11.24	11.31
		5x5	1.00	23.41	23.18	26.68	23.48	20.20	20.44
		7x7	1.00	40.39	40.75	41.97	42.43	31.02	30.53
		9x9	1.00	68.86	64.88	70.42	70.19	37.12	36.82
		11x11	1.00	89.98	97.22	93.52	98.41	41.36	40.97
	2048x2048	3x3	1.00	15.92	15.59	19.55	19.11	15.74	15.69
		5x5	1.00	45.02	42.64	47.38	46.32	28.89	28.89
		7x7	1.00	62.24	70.56	66.47	80.21	36.00	35.37
		9x9	1.00	87.42	104.78	99.32	114.37	40.16	39.83
		11x11	1.00	95.63	129.65	105.95	142.78	42.97	42.59
	4096x4096	3x3	1.00	19.78	17.96	18.96	19.82	16.89	16.87
		5x5	1.00	43.02	40.23	40.92	44.35	28.50	28.68
		7x7	1.00	58.75	63.99	56.91	71.38	35.66	35.65
		9x9	1.00	83.10	94.92	85.26	102.87	40.55	40.45
		11x11	1.00	89.87	119.99	95.77	133.66	43.67	43.40
32x32	256x256	3x3	1.00	3.38	3.41	5.01	3.96	2.70	5.14
		5x5	1.00	6.92	7.15	7.10	7.87	3.74	6.58
		7x7	1.00	7.96	10.07	8.17	11.12	3.96	7.33
		9x9	1.00	11.12	13.88	11.67	14.04	7.71	7.89
		11x11	1.00	10.91	16.02	11.35	16.66	11.37	11.64
	512x512	3x3	1.00	4.06	3.86	10.25	7.91	5.97	8.56
		5x5	1.00	7.73	7.96	14.81	16.19	9.54	9.03
		7x7	1.00	14.83	17.79	16.51	22.39	14.02	13.92
		9x9	1.00	22.12	27.13	23.66	30.71	19.92	20.45
		11x11	1.00	34.09	31.36	33.27	35.90	29.77	31.39
	1024x1024	3x3	1.00	9.85	9.63	16.84	13.72	11.39	11.32
		5x5	1.00	24.94	24.46	26.42	24.05	20.81	21.19
		7x7	1.00	39.32	41.22	41.50	40.34	31.94	31.92
		9x9	1.00	68.04	66.70	75.32	66.00	38.81	38.74
		11x11	1.00	96.21	97.03	103.12	102.02	43.19	43.18
	2048x2048	3x3	1.00	17.81	18.08	21.51	20.03	16.32	16.05
		5x5	1.00	45.78	42.71	53.01	48.18	30.13	30.10
		7x7	1.00	64.01	58.90	87.04	72.63	37.28	37.14
		9x9	1.00	92.81	82.89	124.63	100.48	41.63	41.89
		11x11	1.00	100.37	83.22	155.25	110.68	44.66	44.73
	4096x4096	3x3	1.00	19.81	17.98	18.65	19.54	16.82	17.58
		5x5	1.00	40.34	40.87	41.11	43.94	29.22	29.98
		7x7	1.00	55.76	65.45	57.45	71.36	37.27	37.51
		9x9	1.00	82.29	96.07	85.83	103.07	42.29	42.46
		11x11	1.00	93.02	120.59	96.38	133.14	45.32	45.40