GPU Acceleration for CNN Based Applications

A PROJECT REPORT

Submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

Electrical and Electronics Engineering

by
Ruturaj Nanoti (18BEE0134)
Amitvikram S. Pujar (18BEE0135)
Nishith Nayan (18BEE0166)
Under the guidance of Prof. Selvakumar K

SCHOOL OF ELECTRICAL ENGINEERING
Vellore Institute of Technology
VELLORE-632014, Tamil Nadu, India

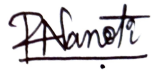
May, 2022

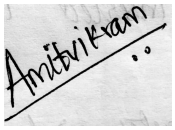
DECLARATION

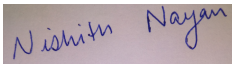
We hereby declare that the thesis entitled ***"GPU Acceleration for CNN based Applications"*** submitted by us, for the award of the degree of *Bachelor of Technology* to VIT, Vellore is a record of bonafide work done under the supervision of Prof. K. Selvakumar. We further declare that the work reported in this thesis has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place: Vellore
Date: 4th May, 2022

Signature of the Candidates:







CERTIFICATE

This is to certify that the thesis entitled ***"GPU Acceleration for CNN based Applications"*** submitted by Ruturaj A. Nanoti (18BEE0134), Amitvikram S. Pujar (18BEE0135) and Nishith Nayan(18BEE0166), School of Electrical Engineering, VIT, Vellore, for the award of the degree of Bachelor of Technology in Electrical and Electronics Engineering , is a record of bonafide work carried out by them under my supervision, as per the VIT code of academic and research ethics.

The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university. The thesis fulfills the requirements and regulations of the University and in my opinion meets the necessary standards for submission.

Place: Vellore
Date: 5/5/2022

Signature of the Guide

The thesis is satisfactory / unsatisfactory

Approved by

Head of the Department (EEE)

Dean

ACKNOWLEDGEMENTS

First and foremost we would like to thank our guide Prof. K. Selvakumar for his constant support and invaluable insight throughout the project. We've gained tremendous knowledge and experience and we're grateful to him. We are grateful to the author of the book titled **Deep Learning with Python**, Francois Chollet for his invaluable contribution in the form of book and the Keras API that enabled us to implement our Deep Learning model.
Last but not the least we extend our gratitude to the Chancellor Dr. G Viswanathan, Dean of SELECT Dr. Mathew M. Noel and the management at VIT,Vellore for having given us this opportunity to study at a world-class University.

Ruturaj A. Nanoti
Amitvikram S. Pujar
Nishith Nayan

Executive Summary

Artificial Intelligence is a field of study encompasses various branches of science like Mathematics, Statistics and Computer Science. The main objective of Artificial Intelligence or AI is to simulate human-level intelligence by adapting to newer environments and at the same time being more accurate than a human. The foundational ideas of AI have been in existence since 1950s. Yet, it is now that we are able to see such massive demand of AI in the world, like Autonomous Vehicles, Digital Content Recommendation Systems, Machine Vision, AI Powered Drones, Ed-Tech Industry, etc.

This huge gap of decades in the conceptualization of ideas and their fruition is primarily attributed to the gradual evolution of the compute systems starting from early mainframe computer to modern day High Performance Computers (HPCs) like IBM's DeepBlue, which got its prominence for defeating the famous chess grandmaster Garry Kasparov. Despite of having access to such highly capable compute platforms, they are restricted to research environments and hardly find any use in the everyday lives of the people. This is where companies like NVIDIA play a crucial role in bringing the technology of parallel computing underlying these HPCs to low-powered, cost-effective and portable platforms like the Jetson Nano. These platforms in unison with various subsystems enable the adoption of AI specifically Deep Learning which finds its use in the area of Computer Vision. This technology of combining Deep Learning and Computer Vision especially on embedded devices is termed as Machine Vision.

This project focuses on implementing inference of Deep Learning models for Image Classification which is a specific use case in Computer Vision. For this purpose we use the GPUs and the GPGPU (General Purpose GPU Programming) pipeline available in the CUDA ecosystem provided by NVIDIA. By leveraging the connectivity of the Jetson Nano with various camera subsystems like LiDAR, this project finds its use massively in the area of autonomous driving.

Contents

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
AP	Anchor Point
CA	Column Array
CNN	Convolution Neural Network
CPU	Central Processing Unit
CSR	Compressed Sparse Row
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random Access Memory
Gen.	Generation
GHz	Gigahertz
GPU	Graphics Processing Unit
ICP	Iterative Closest Point
KB	Kilo Byte
LIDAR	Light Detection and Ranging
LTI	Linear Time Invariant
MAD	Multiply-Add
PCL	Point Cloud Library
PTX	Parallel Thread Execution
ReLU	Rectified Linear Unit
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
SFU	Special Function Unit
SMs	Streaming Multiprocessors
SPs	Scalar Processors
TPC	Thread Processor Cluster

Chapter 1

Convolution: Moving beyond Multiplication

1.1 The Mathematics of Convolution

Convolution is a mathematical operation (particularly used in functional analysis) that operates on 2 functions (say function x and function y) and produces a third function as result $(x * y)$. The convolution operation expresses how much of function x overlaps with function y as it is shifted over it or in simpler words it expresses how shape of one function is modified by the other function.[5] The term itself refers both to the result as well as the process itself. Mathematically the convolution of function x and y is integration of product of the two functions of which one is shifted by certain amount and is expressed as:

$$z(t) = (x * y)(t) = \int_{-\infty}^{\infty} x(\tau)y(\tau - t)\, d\tau \tag{1.1}$$

For discrete convolution the resultant function can be expressed as:

$$z[n] = (x * y)[n]) = \sum_{-\infty}^{\infty} x[m]y[m - n] \tag{1.2}$$

discrete convolution for the purposes of machine learning, image processing is done over multiple axes consecutively, for example consider a two dimensional kernel K and a two dimensional input image I, then the operation is expressed as[5] :

$$s(i, j) = (L * K)(i, j) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} K[m, n]I[i - m, j - n] \tag{1.3}$$

The convolution operation is used extensively in the fields of mathematics, engineering and science. Some of the applications of the convolution operation are:

- Image Processing: In the field of Image processing the convolution operation is extensively used for the purpose filtering operations and used in important algorithms for the functions of edge detection, blurring, smoothing sharpening etc.
- Data processing: For analytical chemistry, to analyse spectroscopic data, Savitzky–Golay digital smoothing filter is used this improves ratio of signal to noise in data with minimal production of distortion in spectra.
- ML: In CNN's, the neural network uses cascaded convolution layers for the purpose of ML and AI.
- Signal Processing : In signal processing, the convolution operation of an input signal with an impulses response gives an output of LTI system.

1.2 Convolution in Image Processing

For the purposes of this project we have implemented convolution for Image processing and then accelerated the inference time by the help of GPU and different algorithms.
In image processing an input image is converted into a matrix where each of its is pixels is assigned some unique value. This way the image is converted to a matrix of distinct values. after this apply the discrete convolution operation on it.[21]
Let us consider an image I of size 5X5 which is subsequently converted to a matrix and a kernel K of size 3X3 as shown below.

$$I = \begin{bmatrix} I_{11} & I_{12} & I_{13} & I_{14} & I_{15} \\ I_{21} & I_{22} & I_{23} & I_{24} & I_{25} \\ I_{31} & I_{32} & I_{33} & I_{34} & I_{35} \\ I_{41} & I_{42} & I_{43} & I_{44} & I_{45} \\ I_{51} & I_{52} & I_{53} & I_{54} & I_{55} \end{bmatrix} \tag{1.4}$$

$$K = \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22}(AP) & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} \tag{1.5}$$

To perform convolution on this image we start from the top left corner cell of the image matrix and the anchor point of the kernel is placed over the cell. The overlapped cell of the kernels and image matrix is multiplied and all the products of overlapped cells are added together, for example let's consider cell I_{22} then the convolution of this cell will result as:

$$O_{22} = I_{22}K_{22} + I_{11}K_{11} + I_{12}K_{12} + I_{13}K_{13} + I_{21}]K_{21} + I_{23}K_{23} + I_{31}K_{31} + I_{32}K_{32} + I_{33}K_{33} \tag{1.6}$$

If a cell of kernel does not overlap with the cell of matrix then zero is multiplied in these cells (also called zero padding). This process is carried on for all the cells of the image matrix and the result is called convolved matrix O as shown below.[21]

$$O = \begin{bmatrix} O_{11} & O_{12} & O_{13} & O_{14} & O_{15} \\ O_{21} & O_{22} & O_{23} & O_{24} & O_{25} \\ O_{31} & O_{32} & O_{33} & O_{34} & O_{35} \\ O_{41} & O_{42} & O_{43} & O_{44} & O_{45} \\ O_{51} & O_{52} & O_{53} & O_{54} & O_{55} \end{bmatrix} \tag{1.7}$$

1.2.1 Separable Filters: Convolution with a split

Generally when we perform convolution on matrix I of size $M \times M$ using a kernel of size$N \times N$each pixel of image I performs N^2 operations thus total number of operation performed during convolution are $M^2 \times N^2$. For smaller size image normal convolution doesn't create much of a issue but as the the size of image increases number of operations performed during the convolution increases significantly , thereby increasing the computational time and requiring more computational power.To tackle this issue we use separable filters which requires much less computational power and time. A separable filter is a broken down filter of 2D kernel which is is represented as product of 1D row filter and 1D column filter.

1.2.1.1 Concept of Separability

As mentioned above the concept of separability states that a two dimensional kernel or filter is separable if it can be broken down into a single row and column matrix whose matrix multiplication results in the original kernel. For example consider a Sobel filter of size 3×3 (used for edge detection) as shown in the matrix below:

$$K = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{1.8}$$

this matrix can be decomposed into a column matrix of size (3×1) C:

$$\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \tag{1.9}$$

and a row matrix R of size (1×3): $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$ .The multiplication of this column and row matrix results in the original filter as shown below:

$$\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

1.2.1.2 Test for separability

Mathematically the concept of separability is based on commutativity property of 2D convolution. Consider a convolution of image I with kernel K as shown in equation :

$$s(i,j) = (I * K)(i,j) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} K[m,n] I[i-m, j-n] \tag{1.10}$$

Now as defined earlier the if Kernel or filter is separable then it can be broken down into the product of row and column matrix. Therefore the kernel K can be referred as:

$$K[i,j] = C[j] \times R[i] \tag{1.11}$$

Now the convolution expression becomes:

$$s(i,j) = (I * K)(i,j) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} C[j] \times R[i] \times I[i-m, j-n] \tag{1.12}$$

$$s(i,j) = \sum_{m=-\infty}^{\infty} R[i] \sum_{n=-\infty}^{\infty} C[j] \times I[i-m, j-n] \tag{1.13}$$

Therefore, we prove that in a separable kernel we perform convolution in two gradual steps to obtain same result and can be expressed as:

$$s[i,j] = R[i] * (C[j] * I[i,j]) \tag{1.14}$$

An easy way to determine if a given kernel or filter is separable or not is to determine the rank of the matrix.If the rank obtained is equal to 1 then the matrix is separable. To determine this if SVD (Singular Value Decomposition) of matrix is a unique singular value then the rank is 1.[21],[7]

1.2.1.3 Separable Filter Implementation

As discussed before in section 1.2.1 during a convolution operation of an square image matrix of size say $(M \times M)$ with a kernel of size $(N \times N)$ each pixel of the output convolved image is result of N^2 number of operations. Thus the total number of operation done to produce an output image is M^2N^2. This number becomes extremely large for large image sizes, for example consider an image of size (180×180) and a kernel of say size (5×5) is used then total number of operation becomes 810,000 operations this results excessive use of bandwidth of computation and increases the time of the image processing operation being performed. To address this issue we implement the approach of separable filters.
Lets consider the above filter is separable then the kernel can be decomposed into a row matrix of size $(1 \times N)$ and a column matrix of size $(N \times 1)$. For the same size image, the convolved each of output image pixel now is result of $(2N)$ operation. Thus the total number of operations performed to produce the output becomes $2M^2N$, which is considerably less than for non separable kernels.[7],[21]

During the convolution operation using separable filter the input image is first convolved with the column matrix or simply the column matrix is passed on every pixel of the input and an intermediate image is produced as a result. After this the intermediate image is convolved with the row matrix, and finally the convolved output image of desired image processing is produced. Fig1.1 shows how convolution is performed using separable filter (original kernel size 3×3).

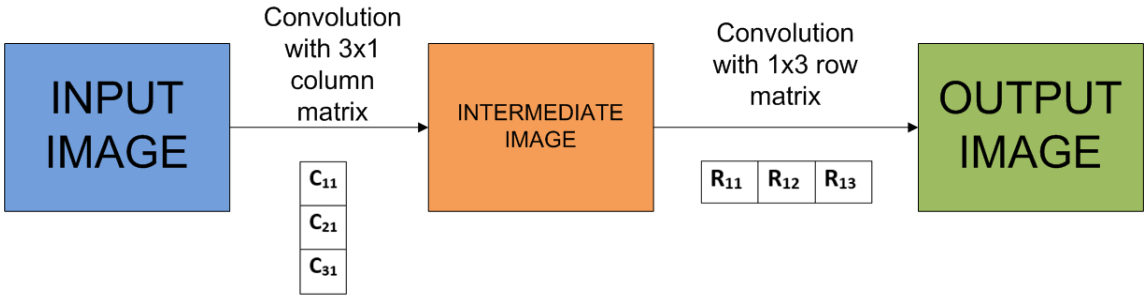


Figure 1.1: Convolution operation using separable filters

Implementation of separable filters provide benefits of more flexible operations and execution, also it reduces the arithmetical complexity and usage of bandwidth consumption for every data point.

Chapter 2

GPU Architecture and Memory Hierarchy

A GPU contains thousands of compute cores (Scalar Processors or SPs), whereas a CPU only contains a small number of processing cores (in multi-core CPUs). Although the current CPUs support a certain degree of parallel behaviour, where the **Classic Five Stage RISC-V Pipeline** : *Fetch* $\rightarrow$ *Decode* $\rightarrow$ *Execute* $\rightarrow$ *Memory (Used for Load/Store type of instructions)* $\rightarrow$ *Write-back (Updating Registers with recently obtained new values after calculation, depends on the instruction if this stage is required)* is used.[29],[4],[15]

In practical situations many hurdles need to be avoided to make this pipeline run successfully, due to the presence of various types of *Hazards* like *Structural Hazards*, *Data Hazards*, *Control Hazards (Branch Hazards)*, *etc.* This is where a GPU or any multi-threaded processor for that matter with a significant number of compute cores comes into picture, where each compute core can operate simultaneously along with the others. They can do the same operation on different data points simultaneously (which is gene rally the requirement in almost all of the algorithms), thus reducing the execution time to an extent that operating on n data points seems like operating on a single data point (since each core does the same operation in parallel).

Data Parallel Algorithms use a sequential implementation, where at each step a specific operation is performed on multiple data points simultaneously by using threads. For example, first we perform add operation on multiple data points simultaneously, then multiply, and so on. These algorithms handle multiple data points in each basic step. So instead of a scalar instruction, we have a vectorized instructions, which when executed operates on vectorized registers. So, that multiple element-wise computations can be done. To accommodate these vectorized instructions we have vectorized register with fixed dimensions that hold each component of the vector. This is used in vector processors. They need to have a lot of parallel based compute capability for precisely this and Here, compilers need to work on loop vectorization, dependency handling, determining which instructions can be executed parallely with data independency. The execution time will depend on the parallel hardware available.

In the GPU there is a hierarchy of processing elements, at the high level we have **Simultaneous Multi-Processing Units (SMs)**. Inside these *SMs* we have specific processors which are known as the **Scalar Processors (SPs)**. The popular names for these are *SM (Streaming Multi-Processors)*, and *SP (Streaming Processors)*. Each SM contains multiple instances of scalar processors or streaming processors.

The diagram shown below is of the *GeForce 8800 GPU* which has a 'Tesla' architecture. It has a total 128 SPs (Streaming or Scalar Processors). That is there are 16 units of SMs (Streaming Multiprocessors) and each SM has 8 units of SPs, giving us a total of 168 SPs $(16 \times 8 = 128)$. Furthermore, 2 SMs are arranged as independent processing units known as texture/processor clusters or Thread Processor Clusters (TPCs). Each TPC has 2 SMs, and each SM has,[28],[18],[19]

- 8 streaming / Scalar multiprocessors (SPs).
- 2 Special Function Units (SFUs).
- A multi-threaded instruction fetch and issue unit (MT Issue), which issues instructions in parallel to each of the Scalar Processors (SPs).
- An Instruction cache, a read only constant cache.
- A 16-Kbyte Read/Write shared memory.

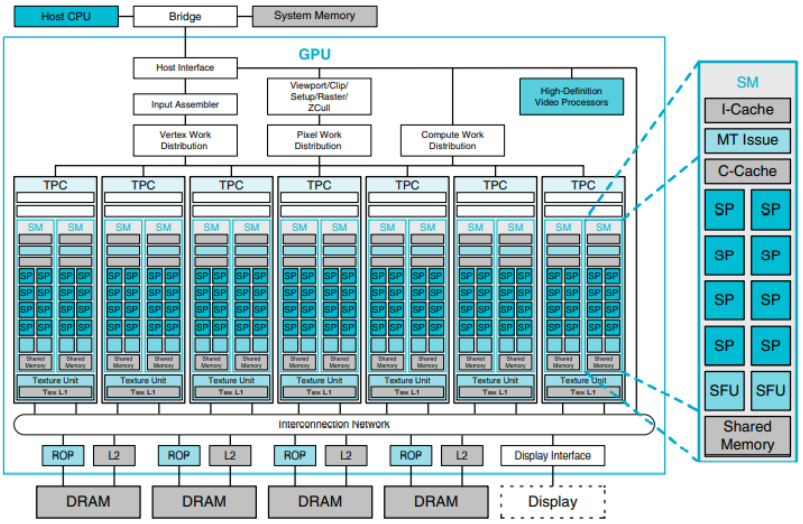


FIGURE B.2.5 Basic unified GPU architecture. Example GPU with 112 *streaming processor* (SP) cores organized in 14 *streaming multiprocessors* (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two *special function units* (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory.

Figure 2.1: GPU Architecture GeForce 8800

2.1 Streaming Multiprocessors

Each of the scalar multiprocessor core has a scalar MAD unit (Multiply-Add), giving the streaming multiprocessors eight Multiply-Add unit. The streaming multiprocessors have have two special function units that are utilized for performing the calculation and operation of transcendental functions/features and attribute interpolation.[18] Every SFU has four floating-point multipliers, as shown in Fig 2.2.

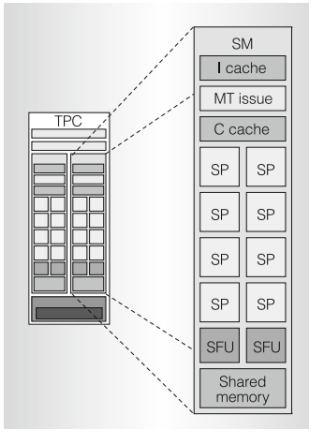


Figure 2.2: Streaming Multiprocessors
[18]

The complete architecture of the SM consists of eight MAD units and four float point multipliers(contained within two SFUs). This structure of SM allows and helps us in parallel execution of eight parallel scalar multiplication and eight floating point multiplication. For management and execution of multiple threads that run several programs or software simultaneously and efficiently GPUs processor use an architecture called SIMT. This architecture doesnot gives a single instruction to multiple data lanes but instead gives a single common instructions to many threads that execute it parallelly.[28],[18],[2]

2.2 Single Instruction Multi Threading

When we try to the execution of instructions for multiple vector, each instruction vectors are executed by threads computing consecutive elements of the vectors that are being operated on. A set of threads starts making progress with each given function. Each of the vector operation operates on multiple threads (SIMD).A grouping of SIMD tasks going on, on numerous various strings in parallel. The execution of threads and its branching actions are controlled by a SIMT function, because when branching happens different threads may have different behaviours. SIMT empowers developers to compose parallel coding at thread level for free threads as well as composition of data level parallel codes for coordinate threads. Generally, a constant code will be replicated across all the threads that are used for execution. SIMT is essentially a single thread of SIMD instructions. SMs contain unit for multi-threaded functions whose job is creation, management, scheduling and execution of different threads in a batch of thirty two threads in parallel which is known as thread . warps is considered to be the most fundamental unit for the purpose of parallel execution of a function in GPUs. In the GPUs every SM controls a batch of twenty four warps which contains 768 threads in total(specific to the GeForce architecture shown earlier). These numbers change with different GPU cards.Every warp threads of SMs are mapped to the SPs. In each operation cycle, the SM warp scheduler selects one of the 24 warps, then this is mapped to its SPs so that the SPs can process the warps. An issued warp executes over 4 processor cycles.The execution of instructions is done independently by the SPs core and the SFUs and is not shared. At a time, there may be many parallel warps being executed, depending on the size of the SM. The core of the SPs contains ALU units of integer type multiplires units of floating type .
The warp threads are assigned to scalar processors by the SMs. The function address and the register state are unique to each of the threads that run independent of each other. For execution of the threads in the warp they the take identical path. When there is a divergence in the threads of the warp due to data-reliant conditional branches, the warps carry the execution serially for each branch paths, disabling the threads that are not within the same pathway, leads to the convergence of threads to initial path of execution after all the pathway are completed. SMs generally handle the threads that are independent of each other that can diverge or converge via a branch sync. stack. Branch divergence only occurs within a deformation Every warps functions independently even if they have a similar code path or unrelated code pathway. This results in Tesla GPUs functioning at a much higher efficient level and highly flexible in functioning than any other previous generation of GPUs when we execute codes on it due to the thirty-two threads present that are much better than SIMD width of earlier generation GPUs.[18],[23],[10]
SIMT provides users a very simple and easy approach for programming of warps independent of each other which is extremely simpler than the complicated programming procedure of earlier versions of architecture of GPUs. As discussed before each warps contains thirty two threads of identical type: vertices,Geometry, Pixel or Calculation. The most basic element of pixel piece or fragment shader's processing unit consists of the quad of 2 pixels. The controller of the streaming multiprocessors bundles a quad of 8 pixels onto a warp of 32-strand treads.Likewise the SIMT bundles the primitives and vertices onto the wraps and bundles computation of thirty two threads on it. The design of SIMT is such that shares the unit that can carry and command the instructions of SMs in an efficient manner on its thirty two threads but for complete throughput efficiency it needs full warping of any active thread. The streaming multiprocessor as a unified graphics processor, programs and carries execution of multiple types of warping, for example: it performs vertex and pixel warps simultaneously. The warp scheduler of the SMs, runs at half speed of 1.5GHzCPU clock speed. For execution of every SIMT warp instruction it selects one out of the twenty four cycle in each cycle.[18]
The warp instruction executes a pair of sets of sixteen threads in four 4 processing cycles. The cores of SPs and the units of SFUs carry out the instructions independent of each other and scheduler keeps them both occupied completely by giving instruction to them in between of alternating cycles. Implementing warp scheduling without dynamic combination of various types of warp schedules and different schedule types is a design problem that is very challenging. The prioritization of existing warps is done by the command scheduler and the one with the highest priority is chosen to issue. The prioritization is taken into accountWarp Type, Order Type and "Equity" for all Warps running on the SM

2.3 Memory Hierarchy

The graphics processing unit (GPUs) have highly complicated and complex memory hierarchy for the purpose of exploiting their extremely massive and powerful power parallel computing operations. Generally, the GPU memory structure are characterised by the memory level hierarchy, they are: Global,Shared,Texture and Constant.

- Global Memory: It is largest available off chip memory of GPU. This functions as main and default on the GPUs. The access of global memory has limited bandwidth memory and high latencies . All the threads of a computing application share the data available on the global memory.[28]
- Shared Memory:This on-chip memory features low latency and high bandwidth. It is software managed and can only be accessed through active threads owned by SMs unit on a GPU.Shared memory transparently visible to all the cores. This is used when doing collaborative computation. It is used by threads in SMs to access data at a low-latency[28]
- Constant Memory:It is already defined area in the global memory section and is by default set to mode read only. The memory space is visible globally to all threads.Local Memory for per-thread, private, temporary memory (implemented in external DRAM, this is used when the registers allocated per-thread are not enough because the thread is engaged in some heavy computation). issues instructions in parallel to each of the Scalar Processors (SPs).[28]
- Texture Memory:Same as the constant read-only memory, texture memory is an off-chip memory space revamped for two-dimensional spatial space. It is useful for threads so that they can access memory addresses that are near together in 2D.[28]

In the end all SMs update value to the Global memory. It helps in collaborative computation across SMs. Even though the temporary memory for the local memory is also implemented in DRAM, it is only used on a per-thread basis, and only that thread can see that memory i.e. it is visble to that thread only. Each streaming multiprocessor has sixty four kilo byte of on-chip memory that can be configured or used as forty eight kilo byte of Shared Memory with sixteen kilo byte of L1 cache or 16KB of shared memory with forty eight kilo byte of L1 cache.

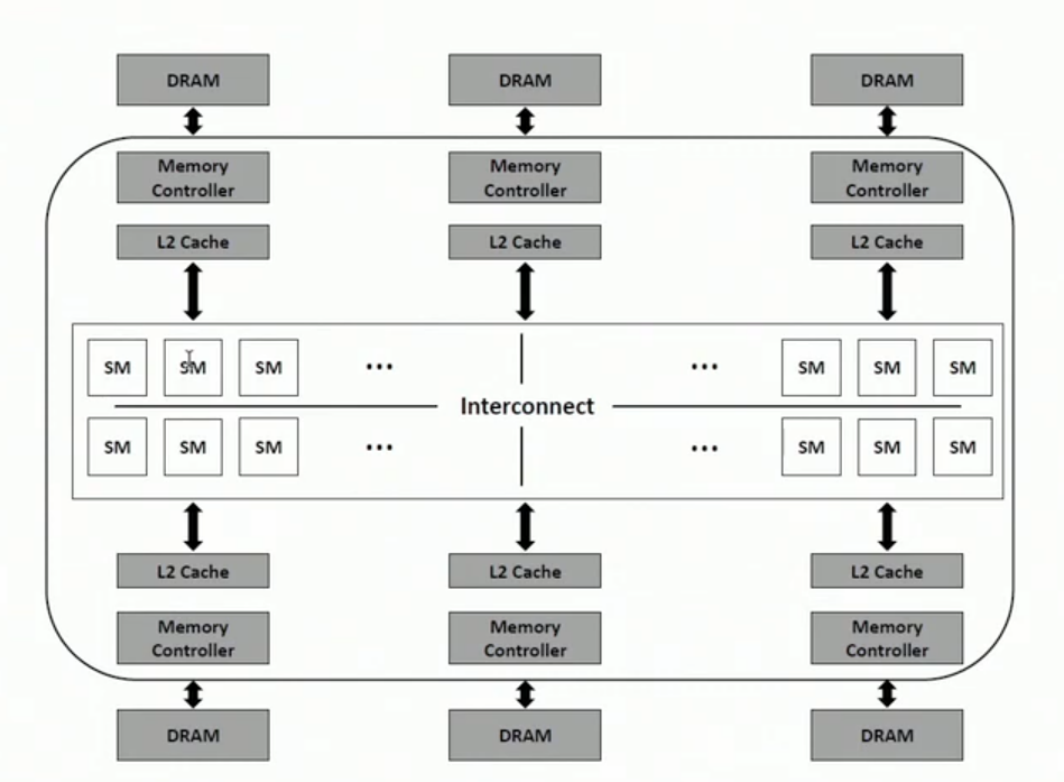


Figure 2.3: Memory Hierarchy of SMs
[18]

L1 (Data) cache plus Shared memory is private to SMs along with read-only texture and constant cache as shown in fig 2.3 .

L2 is unified for all SMs, 6 high-bandwidth DRAM channels. The DRAM is not organised as a big chunk of physical memory, but is divided into multiple banks of physical memory, and all these banks can be accessed in parallel, hence we have 6 high-bandwidth DRAM channels.

Compared to CPU, GPUs have larger register file because the GPUs have a very large number of compute cores (CUDA cores or SPs) that although are simple in nature but carry out operations simultaneously in a huge number which requires large data storage capacity. However, they have smaller L1/L2 cache with higher bandwidth.[18],[28] The access to the DRAM has to be done through a memory controller from the L2 cache.
Despite the fact that each NVIDIA GPUs share a comparable highlevel plan, every new gen. of GPUs present some new memory attributes or execute similar memory with different physcical associations. For example, Fermi GPUs introduced a true global memory cache hierarchy, whereas previous GPUs had no such caches.In GPUs by Kepler, L1 cache and shared memory are combined, while the texture cache is supported by its own memory chip. Pascal and Maxwell GPUs have dedicated memory for their shared memory. For the latest Volta GPUs, L1 cache, shared memory and texture.All caches are merged into a single unified store. All of these hardware changes directly affect memory optimization.[28],[18]

Chapter 3

Parallelizing Data Processing with GPU

3.1 Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture, also known as CUDA is an API and platform for programming parallel computing applications. CUDA provides the users with an interface in the form of CUDA toolkit to interact with the underlying GPU hardware. This toolkit comprises of various GPU-accelerated libraries, compiler, development tools and CUDA runtime.[26],[4],[29],[19]

3.1.1 Programming Model

The main concepts of CUDA programming model as exposed in C++ are outlined as follows:

3.1.1.1 Kernels

CUDA C++ extends the C++ language by allowing the programmer to define C++ functions, named **kernels**, that, when called, are executed M times in parallel by M different CUDA threads, in contrast to only once like regular C++ functions.
The __**global**__ declaration specifier is used to define a kernel.For a given kernel call, the number of CUDA threads that are responsible for its execution are defined using a new $<<<$ ... $>>>$. Every thread involved in the execution is provided a unique *thread ID* that can be accessed within the kernel through the in-built variable $threadIdx$.

3.1.1.2 Thread Arrangement

The $threadIdx$ variable is a 3-dimensional vector such that the threads are identified by 3-d indices. These threads when arranged in either of dimensions constitute a thread block that provides an intuitive way to invoke computation across elements in a vector, matrix or volume. Since each thread of a block is supposed to be on the same processor core, it has to share the limited resources availbale on that core. Currently, the GPUs support upto 1024 threads per thread-block.[25]

3.1.1.3 Memory Arrangement

CUDA threads access data from multiple memory spaces while executing. Every thread contains a private local memory. Every thread block contains shared memory accessible to all threads of a block and having same lifetime as that of the block they are in. Whilst, the global memory is accessible by all the threads. Additionally, the *constant* and *texture* memory spaces are read-only that can be accessed by all threads.

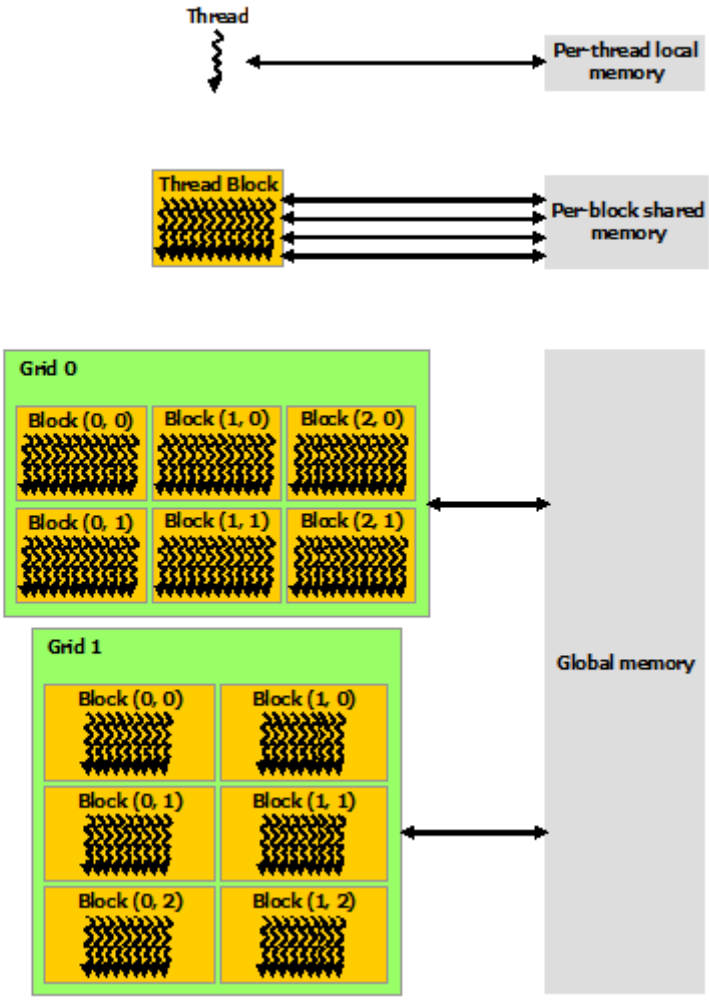


Figure 3.1: Memory Hierarchy

3.1.1.4 Heterogeneous Computing

The CUDA programming paradigm considers the CUDA threads to be executing on a separate hardware which acts as a co-processor to the host which is running C++ program. Additionally, CUDA programming model considers that the host and the device maintain their own memory spaces referred to as : *host memory* and *device memory* respectively.[22] A program manages the various memory spaces visible to the kernel via calls to the CUDA runtime. The host and device memory spaces are linked via the *managed memory*. This managed memory can be accessed from all the CPUs and GPUs in the system as an standalone integrated memory image with a common address space.

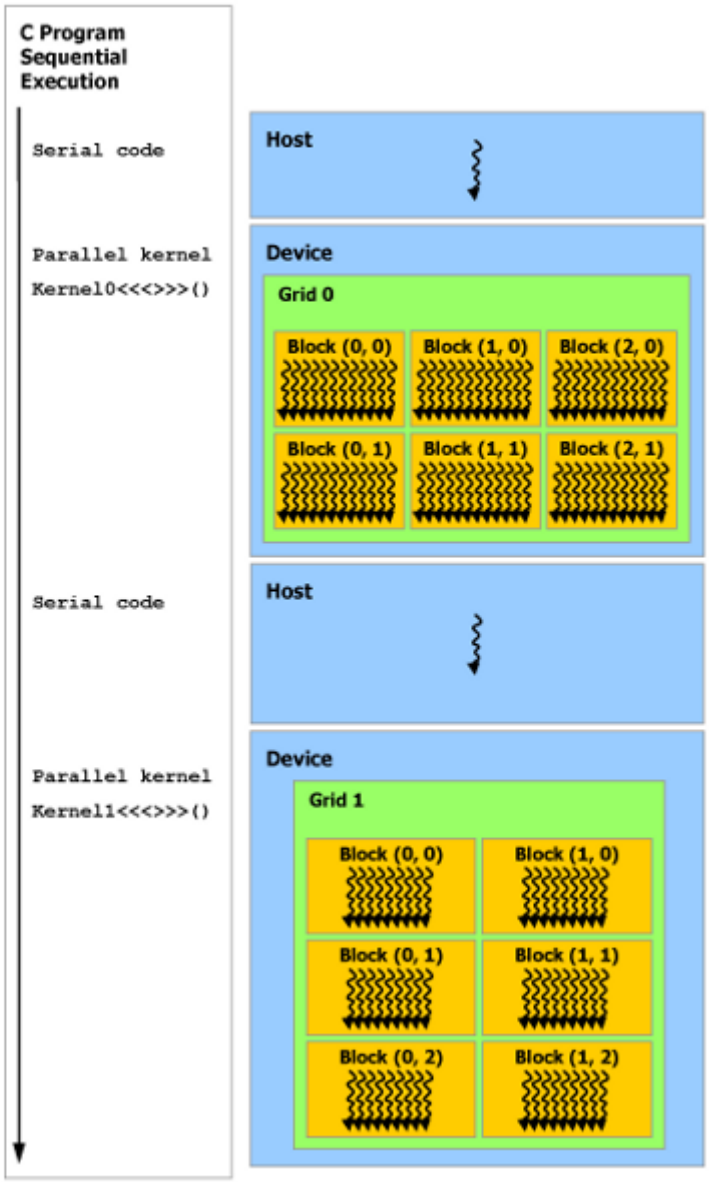


Figure 3.2: Heterogeneous Computing

3.1.2 Asynchronous SIMT compute model

CUDA models the thread to be the least level of abstraction for performing a computation or memory operation. Memory operations are accelerated by the asynchronous programming model that defines the nature of asynchronus operations with regards to CUDA threads. An asynchronous operation is initiated by a CUDA thread and is asynchronously executed as if via another thread.

3.1.3 Compute Capability

The compute capability of a GPU is represented by a version number, also referred to as ”SM version”. It identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.
The compute capability comprises a major revision number M and a minor revision number N and is denoted by M.N.
Devices having the same major revision number belong the same core architecture. The major revision number refer to the different architectures mentioned below:

- 1 for Tesla Architecture

- 2 for Fermi Architecture
- 3 for devices based on Kepler Architecture
- 5 for devices based on Maxwell Architecture
- 6 for devices based on Pascal Architecture
- 7 for devices based on Volta Architecture
- 8 for devices based on Ampere Architecture

The minor revision number corresponds to an incremental improvement to the core architecture, possibly including new features.
Turing is the architecture for devices of compute capability 7.5, and is an incremental update based on the Volta architecture

3.2 Build Process with NVCC

In this section, we will be looking at the process of compiling our CUDA Source Code, which will enable executing the Kernels, on the GPU. Before we jump into **nvcc**, let's understand what compiling actually is, or rather what a compiler like *nvcc* does with the code that we give it. This aspect of software development is as important as the actual application itself, a developer needs to know about the various parts of the build process, which helps in gaining a deeper understanding as to how actually the code that is written in an editor like **Visual Studio Code** or **Vim** is run on the machine, and how it's converted into a form that computer can understand.[30],[6]

3.2.1 Build Example

CUDA Supports language like C, C++, Fortran etc. but for the purposes of this project, we will be focusing on C++. Now to compile C++ code, we generally use the well-known **gcc** or the **g++** compiler. So, these compilers or most of the compilers for that matter follow a build process that contains 5 major steps, which are as follows:

- First, the source code is preprocessed to get a file with the extension **.i**.
- Then, the code is compiled into **Assembly** Files, which have a **.s** extension.
- Then, the assembler takes these assembly files and converts them into **Object** files which have a **.o** extension.
- Atfer that, we have the **Linker** which links all these object files into a single executable, which is known as a relocatable file.
- Finaly, the executable is placed at the right place in memory by a **Locator**, which helps us run the application.

Now that, we have an idea of what the compiler does at least at the surface level, we can jump into the part, where we actually interact with the compiler. A compiler can generally be invoked from the command line, where we pass it various flags and options that help us control various aspects of the build process. Like just compiling the source code into object file not linking it, or just assembling the source code, etc.

The only difference between these compilers that are talked about earlier and **nvcc** is that nvcc needs to generate **device** and **host** code, since we call our CUDA Kernels from the Host (CPU), and also support various range of architectures for the NVIDIA GPUs, like Maxwell, Tesla, Fermi, etc. with each GPU having varying Compute Capability. To do exactly this we will look at some fairly common and important flags and options that can be passed to nvcc, to enable efficient and easy compilation of our source code.

The extenstion for files containing CUDA Kernels or CUDA related code is **.cu**, like **.cpp** for C++. The way to build these .cu source files is very similar to how we build .cpp or .c files. The following shows a basic usage of nvcc.

The parameters provided as options specify different actions to be taken while compilation, like specifying the compilation phase, file and path specifications, compiler/linker behaviour specification, setting the gpu-code generation, etc.

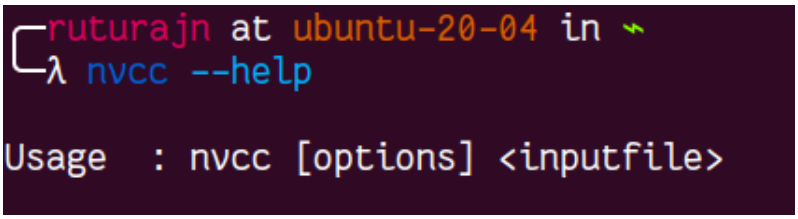


Figure 3.3: NVCC Basic Usage

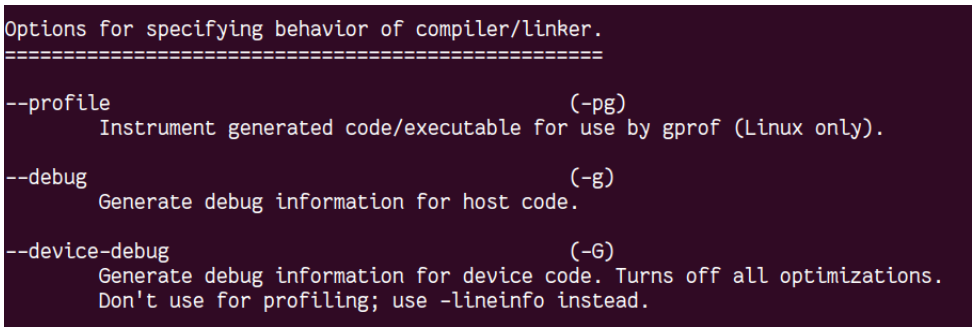


Figure 3.4: NVCC Debug Flags

The options '(-g)' and '(-G)' are important ones here. The former flag prints out the debug information while compiling for the host code, and the latter performs the same thing but with the device code, i.e. the part of the code that will run on the GPU (device).

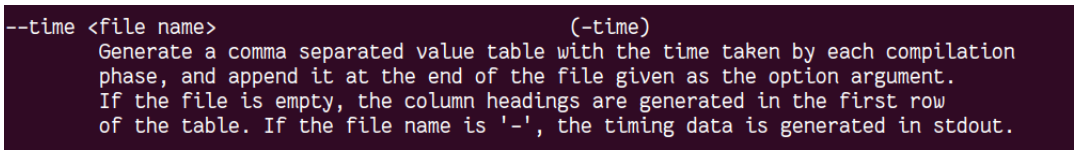


Figure 3.5: NVCC Time Flag

The flag mentioned in the figure above although not very important, it can be used to time the whole build, since it prints out the time for the compilation at the end of the build by giving information about the time taken by nvcc to process various stages of the compilation.

Now, we move on to the options that helps in controlling particular parts related to code generation based on GPU architecture. The flag shown in Fig. 3.6 *-arch* is used to compile the device code for a specific *virtual architecture* of the GPU as described above. It is often described in terms of the compute capability of the device. For example a *NVIDIA GeForce 940MX* has a compute capability of *5.0*, so for building the device code for the *virtual architecture* of this specific GPU we would provide the flag as *-arch=compute_50*, and as pointed out in Fig. 3.6 in the flag description this option alone is not enough to generate the **PTX** (**P**arallel **T**hread **E**xecution) code for the *real architecture* (the actual hardware architecture). Along with this option the *-code* flag also needs to be provided that takes care of exactly this, which is shown in Fig. 3.7.

The flag in Fig. 3.7 helps us define the actual hardware of our device (GPU), during the build. Following up with the example of the NVIDIA GeForce 940MX, we would use the *-code=sm_50* for generating PTX code for our particular graphics card. As you may have noticed the virtual architecture and real architecture take in the same number for the *-code* and the *-arch* flags, i.e. *-code=sm_50* and -arch=compute_50. This is also stated in the description for the flag in the Fig. 3.7 and is also highlighted.

Now, instead of specifying these two options separately, they can be clubbed together by giving a single flag. As can be seen in Fig. 3.8 the flag *-gencode* provides a generalization for the *-arch* and the *-code* flag. Its usage for our example would be *-gencode arch=compute_50,code=sm_50*, this helps us specify both the virtual and real architecture simultaneously.

```
Options for steering GPU code generation.
=========================================

--gpu-architecture <arch>                          (-arch)
        Specify the name of the class of NVIDIA 'virtual' GPU architecture for which
        the CUDA input files must be compiled.
        With the exception as described for the shorthand below, the architecture
        specified with this option must be a 'virtual' architecture (such as compute_50).
        Normally, this option alone does not trigger assembly of the generated PTX
        for a 'real' architecture (that is the role of nvcc option '--gpu-code',
        see below); rather, its purpose is to control preprocessing and compilation
        of the input to PTX.
        For convenience, in case of simple nvcc compilations, the following shorthand
        is supported.  If no value for option '--gpu-code' is specified, then the
        value of this option defaults to the value of '--gpu-architecture'.  In this
        situation, as only exception to the description above, the value specified
        for '--gpu-architecture' may be a 'real' architecture (such as a sm_50),
        in which case nvcc uses the specified 'real' architecture and its closest
        'virtual' architecture as effective architecture values.  For example, 'nvcc
        --gpu-architecture=sm_50' is equivalent to 'nvcc --gpu-architecture=compute_50
        --gpu-code=sm_50,compute_50'.
        -arch=all         build for all supported architectures (sm_*), and add PTX
        for the highest major architecture to the generated code.
        -arch=all-major   build for just supported major versions (sm_*0), plus the
        earliest supported, and add PTX for the highest major architecture to the
        generated code.
        Note: -arch=all, -arch=all-major cannot be used with the -code option, but
        can be used with -gencode options
        Note: the values compute_30, compute_32, compute_35, compute_37, compute_50,
        sm_30, sm_32, sm_35, sm_37 and sm_50 are deprecated and may be removed in
        a future release.
        Allowed values for this option:  'all','all-major','compute_35','compute_37',
        'compute_50','compute_52','compute_53','compute_60','compute_61','compute_62',
        'compute_70','compute_72','compute_75','compute_80','compute_86','compute_87',
        'lto_35','lto_37','lto_50','lto_52','lto_53','lto_60','lto_61','lto_62',
        'lto_70','lto_72','lto_75','lto_80','lto_86','lto_87','sm_35','sm_37','sm_50',
        'sm_52','sm_53','sm_60','sm_61','sm_62','sm_70','sm_72','sm_75','sm_80',
        'sm_86','sm_87'.
```

Figure 3.6: NVCC Architecture Specific Flags

```
--gpu-code <code>,...                              (-code)
        Specify the name of the NVIDIA GPU to assemble and optimize PTX for.
        nvcc embeds a compiled code image in the resulting executable for each specified
        <code> architecture, which is a true binary load image for each 'real' architecture
        (such as sm_50), and PTX code for the 'virtual' architecture (such as compute_50).
        During runtime, such embedded PTX code is dynamically compiled by the CUDA
        runtime system if no binary load image is found for the 'current' GPU.
        Architectures specified for options '--gpu-architecture' and '--gpu-code'
        may be 'virtual' as well as 'real', but the <code> architectures must be
        compatible with the <arch> architecture.  When the '--gpu-code' option is
        used, the value for the '--gpu-architecture' option must be a 'virtual' PTX
        architecture.
        For instance, '--gpu-architecture=compute_60' is not compatible with '--gpu-code=sm_52',
        because the earlier compilation stages will assume the availability of 'compute_60'
        features that are not present on 'sm_52'.
        Note: the values compute_30, compute_32, compute_35, compute_37, compute_50,
        sm_30, sm_32, sm_35, sm_37 and sm_50 are deprecated and may be removed in
        a future release.
        Allowed values for this option:  'compute_35','compute_37','compute_50',
        'compute_52','compute_53','compute_60','compute_61','compute_62','compute_70',
        'compute_72','compute_75','compute_80','compute_86','compute_87','lto_35',
        'lto_37','lto_50','lto_52','lto_53','lto_60','lto_61','lto_62','lto_70',
        'lto_72','lto_75','lto_80','lto_86','lto_87','sm_35','sm_37','sm_50','sm_52',
        'sm_53','sm_60','sm_61','sm_62','sm_70','sm_72','sm_75','sm_80','sm_86',
        'sm_87'.
```

Figure 3.7: NVCC Compute Related Flags

Finally a flag for the File and Path specification, is shown in Fig. 3.9. This is a simple flag it just helps in directly specifying the name of the executable or specify the path to where it should be stored. This is generally used at the

```
--generate-code <specification>,...            (-gencode)
       This option provides a generalization of the '--gpu-architecture=<arch> --gpu-code=<code>,
       ...' option combination for specifying nvcc behavior with respect to code
       generation.  Where use of the previous options generates code for different
       'real' architectures with the PTX for the same 'virtual' architecture, option
       '--generate-code' allows multiple PTX generations for different 'virtual'
       architectures.  In fact, '--gpu-architecture=<arch> --gpu-code=<code>,
       ...' is equivalent to '--generate-code arch=<arch>,code=<code>,...'.
       '--generate-code' options may be repeated for different virtual architectures.
       Allowed keywords for this option:  'arch','code'.
```

Figure 3.8: NVCC Gencode Flag

last step of the build process to generate the final executable.

```
File and path specifications.
=============================

--output-file <file>                           (-o)
       Specify name and location of the output file.  Only a single input file is
       allowed when this option is present in nvcc non-linking/archiving mode.
```

Figure 3.9: NVCC Output File Path Flag

Fig. 3.10 shows a flag that is used for providing any specific libraries that should be linked to the executable during the build process. For example a CUDA program utilizing libraries like *CUFFT*, *CUBLAS*, *CURAND*, etc. should use *-lcufft*, *-lcublas*, *-lcurand* flags.

```
--library <library>,...                        (-l)
       Specify libraries to be used in the linking stage without the library file
       extension.  The libraries are searched for on the library search paths that
       have been specified using option '--library-path'.
```

Figure 3.10: NVCC Linker FLag

All of these flags can be combined to make a full command that utilizes all of them, to build our source code, as shown in Fig. 3.11.

```
ruturajn at ubuntu-20-04 in ~
λ nvcc -G -g -gencode arch=compute_50,code=sm_50 -o kernel kernel.cu
```

Figure 3.11: NVCC Build Command

For example, when other external libraries like *OpenCV*, *PCL*, etc. they need to included and linked them at compile time, which can be done using the *-I* and the *-l* flags for including and linking respectively. A sample command to do just this is shown in Fig. 3.12.

```
ruturajn at ubuntu-20-04 in ~
λ nvcc -G -g -gencode arch=compute_50,code=sm_50 `pkg-config opencv --cflags --libs` -o kernel kernel.cu
```

Figure 3.12: NVCC Build with OpenCV

3.2.2 Compilation Workflow

So, to summarize the **nvcc** compilation workflow goes as follows:

3.2.2.1 Offline Compilation

Source files compiled with **nvcc** contain *host* code and *device* code. Therefore **nvcc** segregates the code into respective categories and then:

- Compiles the device code into assembly i.e. *PTX* and binary i.e. *cubin* object format.
- Replacing the <<< ... >>> syntax in host code by suitable CUDA runtime function calls to load and launch each compiled kernel from the *PTX* code or *cubin* object.

3.2.2.2 Just-In time compilation

The assembly i.e. PTX code loaded by an application at runtime gets compiled further to binary code by the device driver. This is called just-in-time(JIT) compilation. Although, the application load time increases, but it benefits from any new compiler improvements coming with each new device driver. This is the only way for applications to run on devices that did not exist at the time the application was compiled.

3.2.3 Automating the build process

As seen from the discussion above, the final command needed to build the CUDA source file is quite long, and typing it everytime can get tedious after some point. So, this process can be automated by **make**, from the **GNU Suite of Tools**. The following is a simple *Makefile*, which is contains all the flags that have been enumerated above, and more.

In the Makefile, some variables are defined which is similar to developing a program in a programming language. The *CC* variable is assigned the value of *nvcc*, which means that the *CC* variable defines our compiler. The *ARCH_FLAGS* variable contains various flags that define the real and virtual architecture, that the code should be compiled for. The other option provided suppresses the deprecated GPU warnings from nvcc. This is an optional flag. This Makefile is written for a code that uses OpenCV, so it needs to be linked and included at compile time. This is taken care of by the *OPENCV_FLAGS* variable that has the value of *'pkg-config opencv –cflags –libs'*, which includes the OpenCV libraries and contains their respective linker flags, that enable linking at compile time. Finally there are the *SRC* and *TARGET* variables which contain the name of the source file and the target output file that needs to be generated.

Then, we provide a *.PHONY* protection to our user defined targets and give a recipe to build those targets. For example, the recipe for the target *all* is at line 11. When make is called, all the values for the variables are substituted in the recipes and the command hence created is executed in the terminal.

```
1  CC = nvcc
2  NVCC_FLAGS = -G -g -gencode
3  ARCH_FLAGS = arch=compute_50,code=sm_50 -Wno-deprecated-gpu-targets
4  OPENCV_FLAGS = `pkg-config opencv --cflags --libs`
5  TARGET = Image_BLur
6  SRC = Image_Blur.cu
7
8
9  .PHONY: all clean
10
11 all: $(TARGET).out
12
13 $(TARGET).out:
14   $(CC) $(NVCC_FLAGS) $(ARCH_FLAGS) $(OPENCV_FLAGS) -o $(TARGET).out $(SRC)
15
16 clean:
17   rm -f *.out
```

Chapter 4

Algorithm Implementation

4.1 Accelerating Applications with CUDA

This chapter focuses on the implementation of the various CUDA Kernels that were done during the course of this project. The following is a list of various applications, that were developed using CUDA[22]:

- 2D Matrix Addition
- Image Blurring
- Image Blurring Optimized by utilizing Shared Memory
- Image Blurring with Separable Filters
- Translating and Rotating Point Clouds
- Simple Directed Graph

4.1.1 2D Matrix Addition

This is a basic implementation that uses CUDA Kernels to compute the sum of two square matrices. A CUDA Kernel is commonly launched with two parameters one is the number of **thread blocks** and the other is the number of **threads per block**. So, the actual parallelization happening is each thread processing a single element of the 2D Matrix, i.e. each thread in a thread block is responsible for computing the sum of two elements at a particular index in both the matrices. To exemplify, let's assume two 1D arrays of size 100, so a total of 100 threads are launched, and thread 0 is going to compute the sum of the 0^{th} elements of both the arrays. Similarly, each thread computes the sum based on its respective index, and the whole addition is computed parallely. Hence, the amount of time it would take to add two elements is the amount of time it takes to add 100 elements, since all of the addition operations are being done parallely.[2],[17],[20]

The example discussed above, shows the approach taken for a 1D array, or a row or column vector, our implementation focused on adding 2-Dimensional Matrices, so the threads need to be launched in 2-Dimensions, i.e. we have a 2D thread block, which is arranged in its own 2D grid as shown in Fig. 4.1. This helps us index the threads in 2D, and work easily with the given matrices, making sure that the operation performed is done efficiently and accurately. The global ID of a particular thread in a thread block is calculated as follows,

$$row = threadIdx.y + (blockIdx.y \times blockDim.y) \tag{4.1}$$

$$col = threadIdx.x + (blockIdx.x \times blockDim.x) \tag{4.2}$$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{21} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{21} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{21} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} \tag{4.3}$$

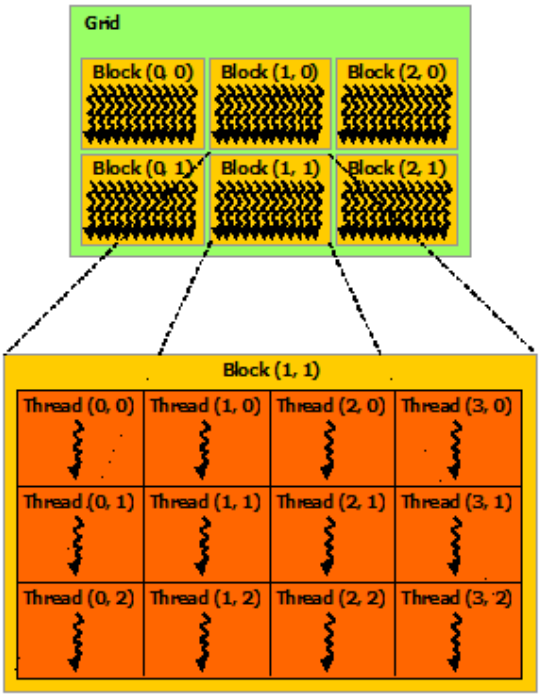


Figure 4.1: Threads and Thread Blocks

Since, in each thread block the thread ID starts from 0, we do not have a track of which element in the actual matrix is being addressed by that thread, so to overcome that we define this global thread ID which gives us the index that a particular thread in a thread block is accessing. This pair of (row, col) is then used to add corresponding elements from both the matrices, and as can be seen from the equations above we have our threads and thread blocks extending in both the x and y direction. Algorithm 1 shows the per thread execution flow, which is the same across all the threads that are launched for the Kernel.

Algorithm 1 Pseudocode for Matrix Addition Kernel

$m \leftarrow$ number of rows
$n \leftarrow$ number of columns
$col \leftarrow threadIdx.x + (blockIdx.x \times blockDim.x)$
$row \leftarrow threadIdx.y + (blockIdx.y \times blockDim.y)$
if $row \leq$ m and $col \leq$ n **then**
    $index \leftarrow (row \times n + col)$
    $C[index] = A[index] + B[index]$
end if

This is not the only configuration of threads and blocks that can be used to launch a kernel that adds 2D Matrices. We could launch 1D threads, and thread blocks where each thread handles or computes addition for a whole corresponding rows of the matrices, or the same configuration can be used where each thread computes the addition for a whole column. The possibilities are immense, the job of a programmer developing these kernels is make sure that they are as optimized as possible.

4.1.2 Image Blurring

Image Blurring is a highly common pre-processing technique, that is used in many Image Processing Algorithms and Convolutional Neural Networks. So, it would be greatly advantageous if this operation could be parallelized, by everaging the GPU compute power.[17],[20]

Image Blurring comes into play in various applications that we see around us everyday. Some of them are,

- Blurring inappropriate or explicit content in the news on TV
- Hiding Data that is private to people, in some cases, even blurring out faces.
- Portrait Effects in Cameras
- Special Effects in movies and digital content, that helps in beautifying scenes.

One of the most important aspect of blurring, is that after this operation is performed on the input image, the noise is reduced, which is especially helpful while training CNNs. There are many types of Blur that can be performed on the image like *Mean/Box, Gaussian,* etc. In this section we will be focusing on the **Gaussian** Blur, which is generated by convolving the input image with a Gaussian Filter. The elements of this filter can be computed by the formula,

$$G(x, y) = \frac{1}{2\pi\sigma} e^{-(x^2+y^2)/2\sigma^2} \tag{4.4}$$

Where x and y are the indices of the filter. As seen in Fig. 4.2 the filter is placed over the image and made to traverse the whole image with a particular stride. The violet box represents the input image, the yellow box represents the filter and the yellow bounding box around represents the padding that needs to be done while convolution, so that the values of the pixels at the boundary are also taken into account. Convolution can be done with and without padding, the latter approach does not take into account the contribution from the pixels at the boundary of the image. This also results in a reduce in the size of the output image, as compared to the input. For the purposes of this project we have taken padding into account and computed convolution according to that.Types of padding :

- **Constant**: Every pixel that gets padded to the original image is assigned a constant value.
- **Replicate**: The pixel values at the edge of the active frame are repeated to make rows and columns of padding pixels.
- **Symmetric**: Padded pixels are added such that they mirror the image edge.
- **Reflection**: The padding pixels are added such that they reflect around the pixel at the edge of the image.This is used for machine learning applications as it removes the edge contrast and maintains texture.

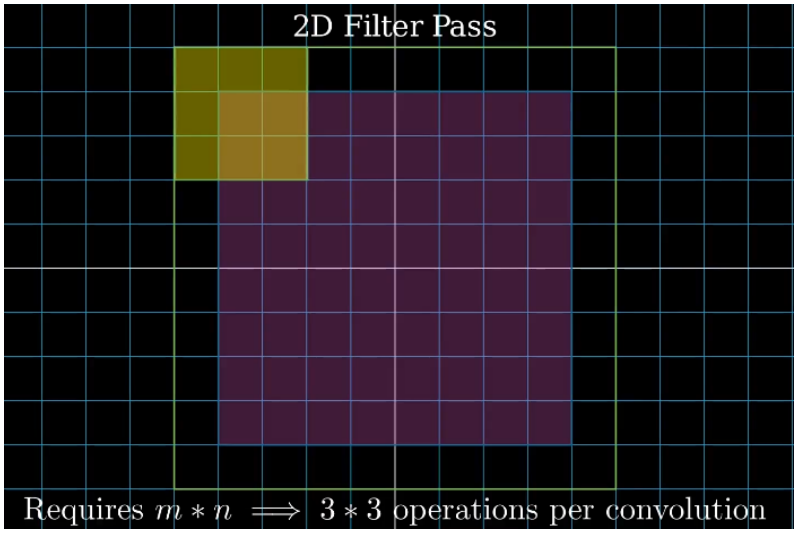


Figure 4.2: Image Blurring Operation Initial Position

From Fig. 4.3 it can easily be understood that the filter moves over the image while computing convolution between itself and the area or patch of the image that is below it. This result is stored as a single pixel value in the output image.

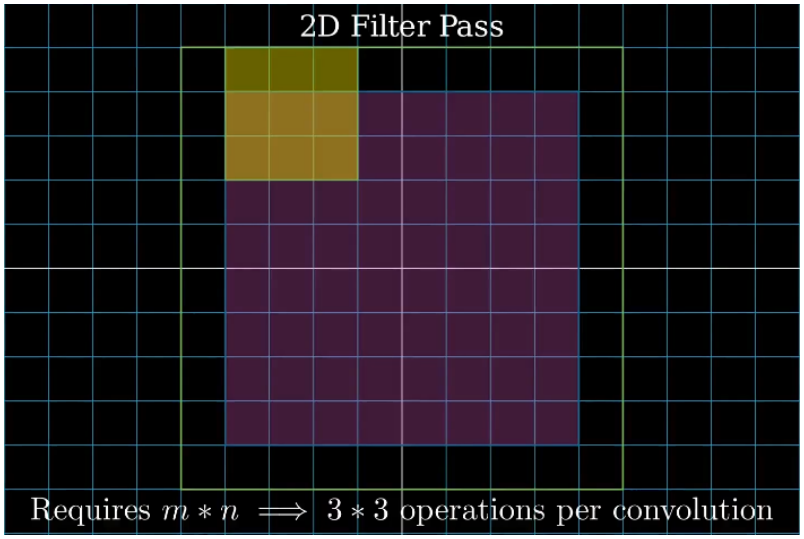


Figure 4.3: Image Blurring Operation

As is clear by now there are many *independent* computations that need to be performed while Blurring an Image, so we can just delegate these calculations to the GPU, which will parallely compute the convolution for each pixel, and that is exactly what is done. A CUDA Kernel is launched with number of threads equal to the number of pixels in the input image. Now, we also need to take into account the separate R,G,and B channels that an image has, since the blurring operation is being performed on a colour image. So, to be explicit, the kernel is launched with number of threads equal to the number of pixels in each channel, where each channel has the same number of pixels. This means, that each thread needs to perform convolution to get the output value for three pixels, which will go into the R,G and B channel.

So, when the filter traverses over the image it places itself over all the three channels, and then we traverse the filter to perform point-wise multiplication between corresponding values of the filter and image.[1],[11] Fig. 4.6 shows the arrangement of the channels in a colour image. This can be seen in Algorithm 2, where each thread thread traverses the area that is under the filter in the image, and computes the convolution for that patch. The result is then written to the three separate channels. Algorithm 2 highlights the execution flow for a single thread, which is the same across all the threads that are launched.[27],[14]

Algorithm 2 Pseudocode for Convolution Kernel

$m \leftarrow$ number of rows
$n \leftarrow$ number of columns
$col \leftarrow threadIdx.x + (blockIdx.x \times blockDim.x)$
$row \leftarrow threadIdx.y + (blockIdx.y \times blockDim.y)$
if $row \leq$ m and $col \leq$ n **then**
    Defining the starting points of the filter
    $start\_row = row - (filter\_width)/2$
    $start\_col = col - (filter\_width)/2$
    for i=0 to filter _width **do**
        for j=0 to filter _width **do**
            $row\_val = start\_row + i$
            $col\_val = start\_col + j$
            Defining the Padding, i.e. clamping the values to the border
            if $row\_val \leq 0$ **then**
                $row\_val = 0$
            else if $row\_val > m$ **then**
                $row\_val = m - 1$
            end if
            Doing the Same thing for the columns
            if $col\_val \leq 0$ **then**
                $col\_val = 0$
            else if $col\_val > n$ **then**
                $col\_val = n - 1$
            end if
            $index \leftarrow (row\_val \times n) + col\_val$
            $img\_val\_r \leftarrow img[index \times 3]$
            $img\_val\_g \leftarrow img[(index \times 3) + 1]$
            $img\_val\_b \leftarrow img[(index \times 3) + 2]$
            $filter\_val \leftarrow filter[(i \times filter\_width) + j]$
            Performing Convolution
            $prod\_r \leftarrow prod\_r + (img\_val\_r \times filter\_val)$
            $prod\_g \leftarrow prod\_g + (img\_val\_g \times filter\_val)$
            $prod\_b \leftarrow prod\_b + (img\_val\_b \times filter\_val)$
        end for
    end for
    $res[((row \times n) + col) \times 3] \leftarrow prod\_r$
    $res[((row \times n) + col) \times 3 + 1] \leftarrow prod\_g$
    $res[((row \times n) + col) \times 3 + 2] \leftarrow prod\_b$
end if

Results

Table 4.1 shows the performance characteristics for Image Blurring done on Images of various sizes, on the GPU and the CPU (Intel i5-8250, 8 cores, 3.4 GHz). Fig. 4.4, and 4.5 show the blurred images that were obtained. The GPU(NVIDIA GeForce 940 MX) gives a 90.6528% decrease in execution time as compared to the CPU for an image size of *3840x2560*. For an image size of *1920x1080* there is a 90.6765% decrease in execution time on the GPU. As far as the Jetson Nano us concerned, we have a 71.2144% decrease in execution time, when an image of size *3840x2560* is blurred. When an image of size *1920x1080* is blurred on the Jetson Nano, we have a 71.2558% decrease in the execution time. All the Images were blurred with a *9x9* Gaussian Filter.

Figure 4.4: Image Blurred with Global Memory Utilization 1

Figure 4.5: Image Blurred with Global Memory Utilization 2

Table 4.1: Global Memory Comparison

Size	CPU code (ms)	NVIDIA GeForce 940 MX (ms)	Jetson Nano 2GB (ms)
3840 x 2560	16000	1495.55	4605.691
1980 x 1080	3400	317	977.304
1280 x 720	1520	143	445.989
640 x 480	513	47.7	158.264
313 x 557	300	28	103.8946
183 x 275	95	8.781	45.669

4.1.3 Optimizing Image Blurring by Using Shared Memory

Fig. 4.2 and Fig. 4.3 show that the values used from the image to compute the convolution of adjacent pixels overlap quite a bit, and global memory accesses account for a major chunk of execution time of the kernel. Since, the values required for convolution overlap a lot, we are making separate memory access request every time we perform the convolution for a single pixel, and many of these access are repeated, i.e. the values that are required for the convolution adjacent pixels are requested twice, which costs a lot, in terms of computation time. [8]

To tackle this issue we can either reduce the number of global memory accesses, or store the data elsewhere which is near to the SPs. Shared Memory helps us acheive the latter, since its on-chip, and the time required to access it

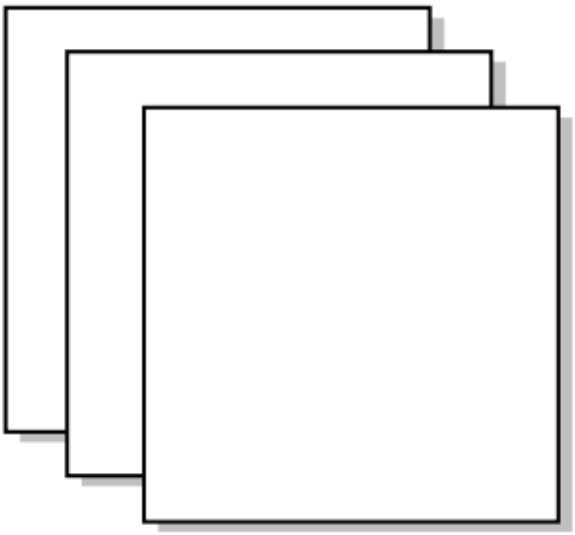


Figure 4.6: Image Channels

lower than global memory, which helps reduce the overall execution time of the kernel.

In this approach if we need to copy the data from the global memory into the shared memory before computation is performed using it. All the threads when launched bring in a specific number of pixels into the shared memory parallely, so the whole image is loaded into shared memory simultaneously. It is important to consider the number of pixels each thread brings in for two reasons which are:

- Convolution Depends on neighbouring pixel values
- Shared memory is private to a thread block, i.e. only threads inside a block can access a particular blocks shared memory.

So, there might be some cases where the complete convolution operation cannot be performed due to the lack of availability of the neighbouring pixel values in a particular block's shared memory. To solve this, we will need to load apron pixels into shared memory along with the actual pixel values of the image. Apron pixels are the similar to padding, that was shown in 4.2. This will enable convolution to be performed without worrying about the filter falling off the image block in the shared memory.

There is a little problem that arises when one tries to implement blurring with the technique described above, with the number of threads equal to the total pixels in the image and the apron pixels. When such a configuration is used, after loading the data into shared memory all the threads that brought in the apron pixels will remain idle and not perform any computation, since they do not actually correspond to any index in the image, they are just used ti load in the required padding. This results in **thread divergence**, which is undesirable in a kernel, and needs to be addressed. So, we launch the kernel with the number of threads equal to the number of pixels in the image, where each thread brings in more than a single pixel into shared memory, which eliminates the problem of idle threads, and also reduces the use of GPU resources, which is always preferable.

In the approach used here, we load in 4 pixels per thread for a filter size of 9×9, which handles the issue of thread divergence, and reduces memory access time since we are utilizing shared memory. Once the loading is done, we have the whole image along with the padding in shared memory which is split in various blocks. Then, we use a similar method to compute convolution as described in the previous section, the only difference is instead if accessing global memory everytime the data is accessed from the shared memory which has a lifetime of of the kernel, i.e. the data remains as long as the kernel is executing on the device.

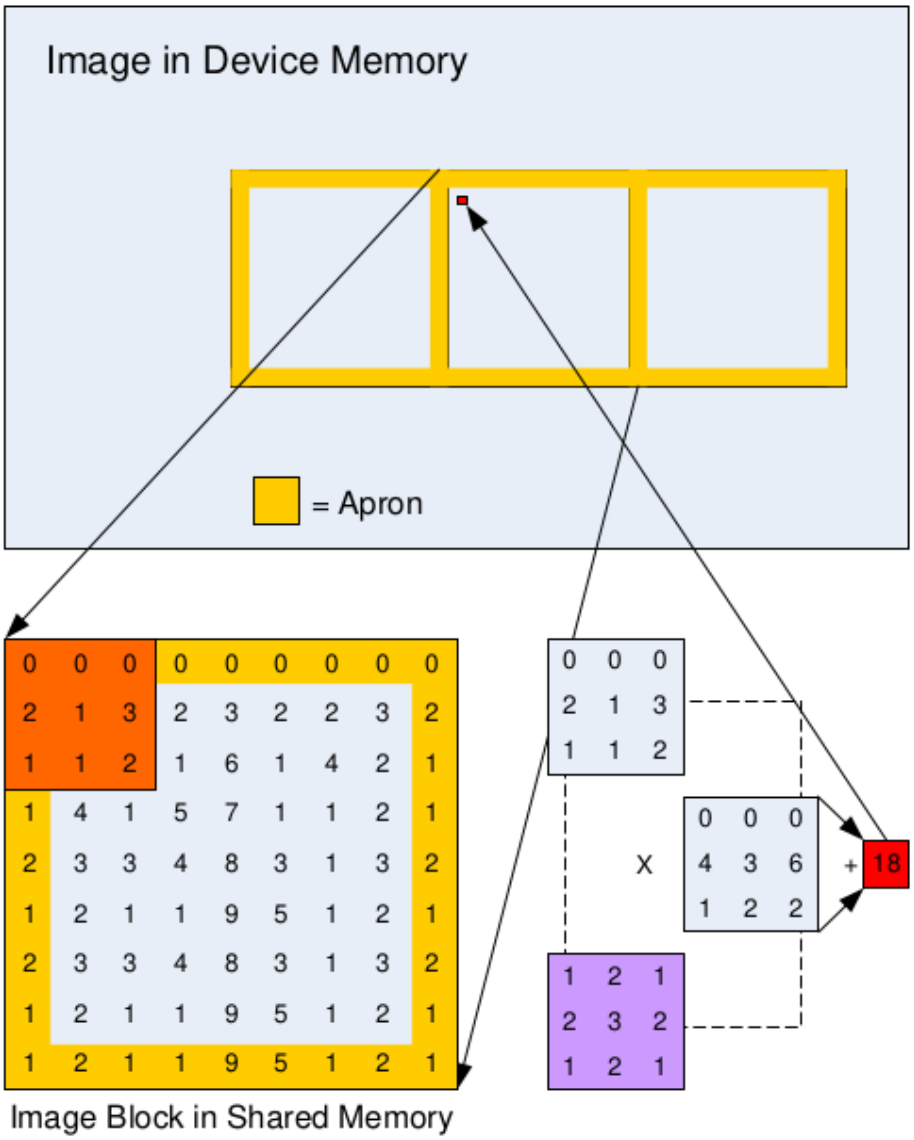


Figure 4.7: Loading Image into Shared Memory

This is explained via Algorithm 3. Where, a block of four adjacent pixels in a 2D patch are transferred from Global (DRAM) to Shared Memory. This data can then be accessed for computation, to decrease memory access time, and increase the performance of the kernel. At the end we use Algorithm 2 to compute the convolution. A very important thing that needs to be kept in mind while utilizing shared memory is that it is very small as compared to the DRAM (Global Memory), so wise use of it by the programmer is necessary to avoid memory access related errors. Algorithm 3 is followed by each thread among all the threads launched.

All the Image Blurring Operations that are discussed use OpenCV for reading images from disk. It is also used to convert the image data into a 1D array, and from the 2D image. Once, we have the actual data it needs to be transferred to the device from the host, which is done using inbuilt CUDA memory access directives. The Algorithm 3 shows the execution flow for a single thread, this logic is run by all the threads that are launched in a parallel manner.

Algorithm 3 Kernel Pseudocode for Convolution with Shared Memory

$m \leftarrow$ number of rows
$n \leftarrow$ number of columns
$col \leftarrow threadIdx.x + (blockIdx.x \times blockDim.x)$
$row \leftarrow threadIdx.y + (blockIdx.y \times blockDim.y)$
$BLOCK\_PADDED \leftarrow filter\_width + THREADSx$
$temp\_img[(BLOCK\_PADDED - 1) \times (BLOCK\_PADDED - 1) \times 3]$
Defining the starting points of the filter
$start\_row \leftarrow row - (filter\_width)/2$
$start\_col \leftarrow col - (filter\_width)/2$
$img\_idx = start\_row$
$img\_idy = start\_col$
if ($start\_row \leq 0$ or $start\_col \leq 0$ or $start\_row \geq (m-1)$ or $start\_col \geq (n-1)$) **then**
    Copy the Padding to the Shared Memory
else
    Copy the Actual Pixel values from Global Memory into Shared Memory
end if
$img\_idx \leftarrow start\_row + 1$
$img\_idy \leftarrow start\_col$
if (($start\_row + 1$) ≤ 0 or $start\_col \leq 0$ or $(start\_row + 1) \geq (m-1)$ or $start\_col \geq (n-1)$) **then**
    Copy the Padding to the Shared Memory
else
    Copy the Actual Pixel values from Global Memory into Shared Memory
end if
$img\_idx \leftarrow start\_row$
$img\_idy \leftarrow start\_col + 1$
if ($start\_row \leq 0$ or $(start\_col + 1) \leq 0$ or $start\_row \geq (m-1)$ or $(start\_col + 1) \geq (n-1)$) **then**
    Copy the Padding to the Shared Memory
else
    Copy the Actual Pixel values from Global Memory into Shared Memory
end if
$img\_idx \leftarrow start\_row + 1$
$img\_idy \leftarrow start\_col + 1$
if (($start\_row + 1$) ≤ 0 or $(start\_col + 1) \leq 0$ or $(start\_row + 1) \geq (m-1)$ or $(start\_col + 1) \geq (n-1)$)
then
    Copy the Padding to the Shared Memory
else
    Copy the Actual Pixel values from Global Memory into Shared Memory
end if
Now implement Algorithm 2 here for Convolution

Results

Table 4.2 shows the performance of the Image Blurring performed on the CPU (Intel i5-8250, 8 cores, 3.4 GHz), versus GPU, and Fig. 4.8 and Fig. 4.9 demonstrates the Blurring done on images. As can be seen from Table 4.2, the NVIDIA GeForce 940 MX reduces the execution time for a *3840x2560* image by approximately 91.5233%. Similar reduction in execution time can be noticed while comparing the *1280x720* image, where the reduction in the execution time on the GPU as compared to the CPU is 91.773%. As far as the Jetson Nano is concerned, it gives us a 76.1292% decrease in the execution time for the *3840x2560* image, and a 75.8426% decrease for an image of size *1280x720*. This trend applies for all the images that were tested, which makes it very clear that utilizing the Shared Memory available in the GPU, can greatly improve performance, and this boost in performance gets amplified as the image sizes increase. All of these images were convolved with a Gaussian Filter of size *9x9*.

Figure 4.8: Image Blurred with Shared Memory Utilization 1

Figure 4.9: Image Blurred with Shared Memory Utilization 2

Table 4.2: Shared Memory Comparison

Size	**CPU code (ms)**	**NVIDIA GeForce 940 MX (ms)**	**Jetson Nano 2GB (ms)**
3840 x 2560	16000	1356.27	3819.33
1980 x 1080	3400	285	820.445
1280 x 720	1520	125	367.193
640 x 480	513	42.22	137.925
313 x 557	300	25.5	86.866
183 x 275	95	7.77	41.582

4.1.4 Image Blurring with Separable Filters

All the convolution approaches presented upto now involved using the conventional type of 2D filters. Now, we look at *separable* filters, where a 2D filter can be broken down into a **row** and a **column** filter, such that when we multiply them the result is the original 2D fitler. The idea here is to perform convolution twice, first by convolving the input with the column filter, and convolving the output of that operation with the row kernel, which will give us a result that corresponds to convolving the input image with a 2D filter.

Hence, two kernels are defined, one that performs the convolution with the row filter and the other to perform the convolution with the column filter. A very similar approach used in the previous section is applied here, the kernel is launched with the number of threads equal to the number of pixels in the input image and the convolution in performed in a cascaded manner, i.e. the output from the convolution of the image with the row filter is fed as the input for the convolution to be performed with the column filter.

Algorithm 4 Convolution with Separable Filters Row Kernel

$m \leftarrow$ number of rows
$n \leftarrow$ number of columns
$col \leftarrow threadIdx.x + (blockIdx.x \times blockDim.x)$
$row \leftarrow threadIdx.y + (blockIdx.y \times blockDim.y)$
if $row \leq$ m and $col \leq$ n **then**
  for i=0 to filter _width **do**
    $col\_val \leftarrow col + i - (filter\_width)/2$
    Defining the Padding, i.e. clamping the values to the border
    if $col\_val \leq 0$ **then**
      $col\_val = 0$
    else if $col\_val \geq n$ **then**
      $col\_val = n - 1$
    end if
    $temp\_id\_conv \leftarrow ((row \times n + col) \times 3)$
    $filter\_val \leftarrow filter\_row[i]$
    $img\_val\_r \leftarrow img[temp\_id\_conv \times 3]$
    $img\_val\_g \leftarrow img[(temp\_id\_conv \times 3) + 1]$
    $img\_val\_b \leftarrow img[(temp\_id\_conv \times 3) + 2]$
    Performing Convolution
    $prod\_r \leftarrow prod\_r + (img\_val\_r \times filter\_val)$
    $prod\_g \leftarrow prod\_g + (img\_val\_g \times filter\_val)$
    $prod\_b \leftarrow prod\_b + (img\_val\_b \times filter\_val)$
  end for
  $res[((row \times n) + col) \times 3] \leftarrow prod\_r$
  $res[((row \times n) + col) \times 3 + 1] \leftarrow prod\_g$
  $res[((row \times n) + col) \times 3 + 2] \leftarrow prod\_b$
end if

As can be seen from Algorithm 4 and Algorithm 5, both the kernels work in a very similar fashion, and the logical flow given in those algorithms is followed by every thread that is launched. The only part where they differ is where the padding is done. For the row kernel we pad the left and right side of the image, and for the column kernel we pad the top and bottom of the image. Another important aspect that needs to be noted for using separable filters for convolution is that it drastically reduces the number of required operations from n^2 to $2 \times n$, since we need to perform n operations per row filter and n operations per column filter, which gives us $2 \times n$.

Fig. 4.10 and Fig. 4.11 show the padding that is done for each row and column filters, which also explains visually the reduced number of operations that need to performed for this technique. To put this into mathematical terms, conside the example given below,

$$\frac{1}{4}\begin{bmatrix}1\\2\\1\end{bmatrix} \times \frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix} = \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{bmatrix} \tag{4.5}$$

Algorithm 5 Convolution with Separable Filters Column Kernel

$m \leftarrow$ number of rows
$n \leftarrow$ number of columns
$col \leftarrow threadIdx.x + (blockIdx.x \times blockDim.x)$
$row \leftarrow threadIdx.y + (blockIdx.y \times blockDim.y)$
if $row \leq$ m and $col \leq$ n **then**
  for i=0 to filter _width **do**
    $row\_val \leftarrow row + i - (filter\_width)/2$
    Defining the Padding, i.e. clamping the values to the border
    if $row\_val \leq 0$ **then**
      $row\_val = 0$
    else if $row\_val \geq n$ **then**
      $row\_val = n - 1$
    end if
    $temp\_id\_conv \leftarrow ((row \times n + col) \times 3)$
    $filter\_val \leftarrow filter\_row[i]$
    $img\_val\_r \leftarrow img[temp\_id\_conv \times 3]$
    $img\_val\_g \leftarrow img[(temp\_id\_conv \times 3) + 1]$
    $img\_val\_b \leftarrow img[(temp\_id\_conv \times 3) + 2]$
    Performing Convolution
    $prod\_r \leftarrow prod\_r + (img\_val\_r \times filter\_val)$
    $prod\_g \leftarrow prod\_g + (img\_val\_g \times filter\_val)$
    $prod\_b \leftarrow prod\_b + (img\_val\_b \times filter\_val)$
  end for
  $res[((row \times n) + col) \times 3] \leftarrow prod\_r$
  $res[((row \times n) + col) \times 3 + 1] \leftarrow prod\_g$
  $res[((row \times n) + col) \times 3 + 2] \leftarrow prod\_b$
end if

$$\begin{bmatrix}1 & 2 & 1\end{bmatrix} * \begin{bmatrix}2 & 3 & 3\\ 3 & 5 & 5\\ 4 & 4 & 6\end{bmatrix} = \begin{bmatrix}11\\ 18\\ 11\end{bmatrix} \tag{4.6}$$

$$\begin{bmatrix}1\\ 2\\ 1\end{bmatrix} * \begin{bmatrix}11\\ 18\\ 11\end{bmatrix} = 65 \tag{4.7}$$

$$\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix} * \begin{bmatrix}2 & 3 & 3\\ 3 & 5 & 5\\ 4 & 4 & 6\end{bmatrix} = 65 \tag{4.8}$$

Equations 4.6, 4.7 and 4.8 which are **point-wise** multiplications show that when we multiply the two filters it gives us the 2D matrix. It also demonstrates the reason for getting the same result from convolution with row and column filters and the 2D filter as a whole.

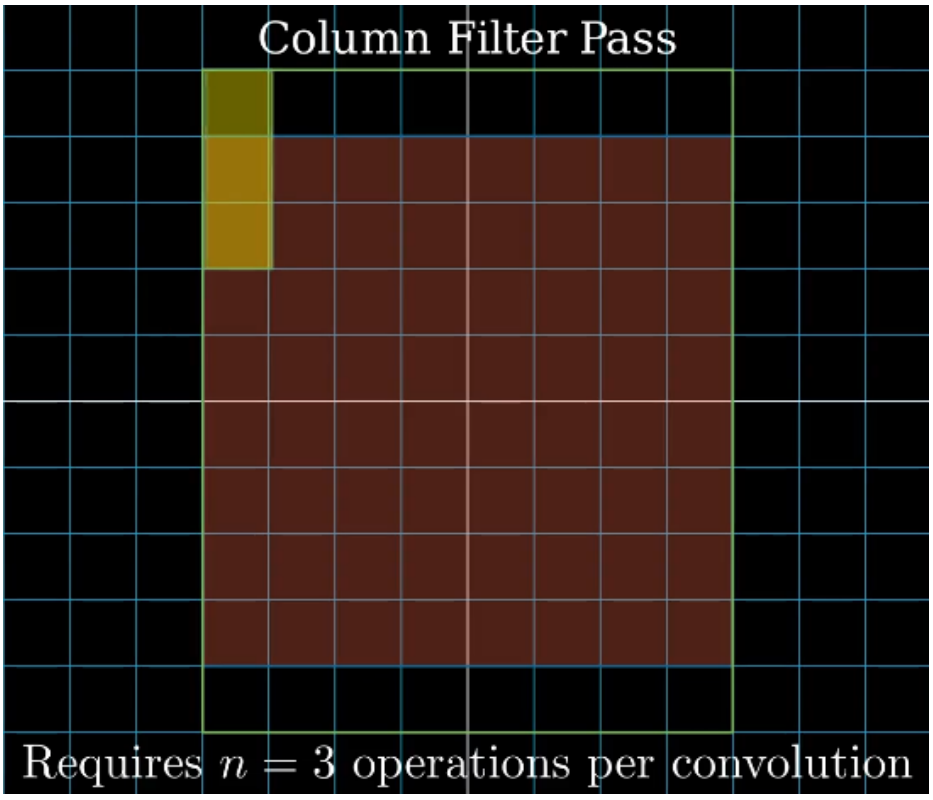


Figure 4.10: Convolution with Column Filter

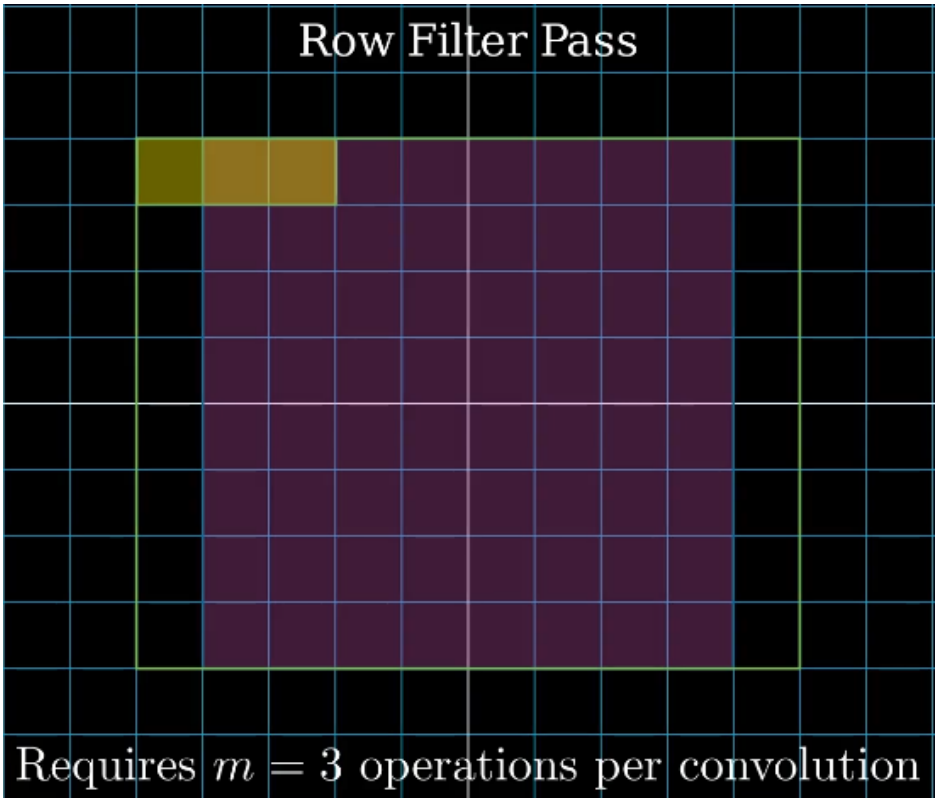


Figure 4.11: Convolution with Row Filter

Results

Figure 4.12: Image Blurred with Separable Gaussian Filter 1

Figure 4.13: Image Blurred with Separable Gaussian Filter 2

Table 4.3: Blurring with Separable F ilters Comparison

Size	CPU code (ms)	NVIDIA GeForce 940 MX (ms)	Jetson Nano 2GB (ms)
3840 x 2560	16000	281.546	725.581
1980 x 1080	3400	58.596	189.110
1280 x 720	1520	27.064	146.532
640 x 480	513	9.325	73.122
313 x 557	300	5.415	50.335
183 x 275	95	1.610	31.543

Table 4.3 shows the execution timing for Convolution with Seprarable Filters. Convolution with this technique on the GPU (NVIDIA GeForce 940 MX) gives a 79.2412% decrease in execution time as compared to the time it takes to perform conventional convolution on the same GPU, which is recorded in Table 4.2. Similarly on the Jetson Nano, we see a 81.0024% decrease in the execution time for an image size of *3840x2560*. This trend is followed by the decrease in execution time for all the various image sizes that were tested. This drastic fall in the execution time results from the vast decrement in the number of computations that need to be done in the case of Separable Filters. This affect is amplified as we appraoch higher image sizes. The Images were convolved with a Gaussian *9x1* column and *1x9* row filter.

4.1.5 Transforming Point Clouds

Point cloud data represents 3-dimensional object or space. They represent the geometric co-ordinates of a point of a sampled surface. This kind of data is generated via 3-dimensional laser scans and Light Detection and Ranging (LiDAR) technology. A point in the point cloud indicates a single laser scan measurement. The collection of these scans are arranged together to create a complete capture of a scene. This process is referred to as registration.[31],[3]
Point Clouds are immensely large, and handling such a huge amount of data in a sequential manner, will require a huge amount of time. A GPU is the perfect device to process such a large chunks of data, which can work on individual points parallely.[16],[12]

For the purposes of this project we have translated and rotated a point cloud, to show the way in which a GPU can be utilized to process this type of data. This type of transformation is very useful in important algorithms like **ICP** (**I**terative **C**losest **P**oint).

Algorithm 6 Pseudo Code for Point Cloud Transformation Kernel

$tid \leftarrow threadIdx.x + (blockIdx.x \times blockDimx.x)$
if $tid \leq num\_points$ **then**
  for $i = 0$ to 3 **do**
    $prod\_out[i] \leftarrow prod\_out[i] + rot\_natrix[i \times 3] \times points\_in[tid].x$
    $prod\_out[i] \leftarrow prod\_out[i] + rot\_natrix[(i \times 3) + 1] \times points\_in[tid].y$
    $prod\_out[i] \leftarrow prod\_out[i] + rot\_natrix[(i \times 3) * 2] \times points\_in[tid].z$
  end for
  $points\_out[tid].x \leftarrow prod\_out[i] + trans\_matrix[0]$
  $points\_out[tid].y \leftarrow prod\_out[i] + trans\_matrix[1]$
  $points\_out[tid].z \leftarrow prod\_out[i] + trans\_matrix[2]$
end if

Algorithm 6 demonstrates the logical flow that is followed by each and everyone of the threads that are launched. Each point in the point cloud goes through the transformation, that is it's (x, y, z) are treated as a column vector which is operated upon by a rotation matrix and a translation matrix. Since, the same computation needs to be done independently, on each point in the point cloud, there exists a high parallelization potential, and we leverage that property of point clouds. The implementation showed in Algorithm 6 employs a rotation matrix that rotates the point cloud about the $z - axis$, which is given by,

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.9}$$

The translation matrix is just a column vector with it's elements being the values by which the point needs to be translated in each direction. It is given by,

$$T = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{4.10}$$

The final operation that is being done on an individual point can be represented as,

$$\begin{bmatrix} x_{Transformed} \\ y_{Transformed} \\ z_{Transformed} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_{Initial} \\ y_{Initial} \\ z_{Initial} \end{bmatrix} + \begin{bmatrix} x_{Translation} \\ y_{Translation} \\ z_{Translation} \end{bmatrix} \tag{4.11}$$

Point cloud data is read with the help of the **PCL**(**P**oint **C**loud **L**ibrary). This library is specifically made to work on various types of point cloud formats, and the vast operations that can be done on them. When, the data is read, it is stored a 1D array with each point being of the type *PointXYZ* , so we need to multiply 3 from the starting position each time to access the x, y and z coordinates of the point.

Results

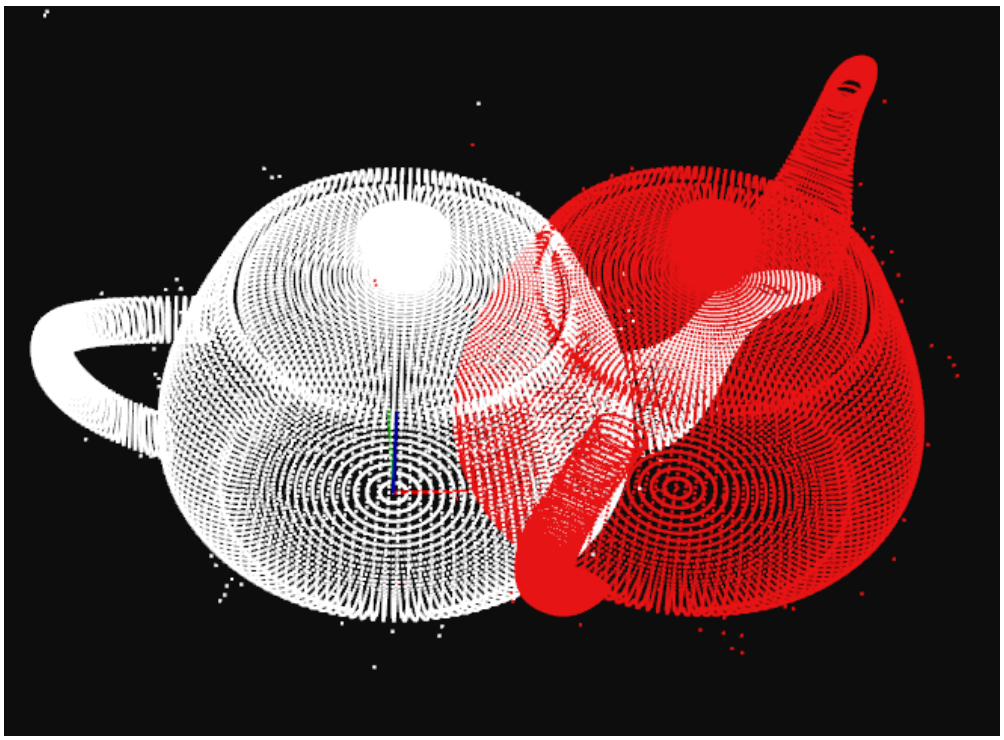


Figure 4.14: Teapot Point Cloud Transformation

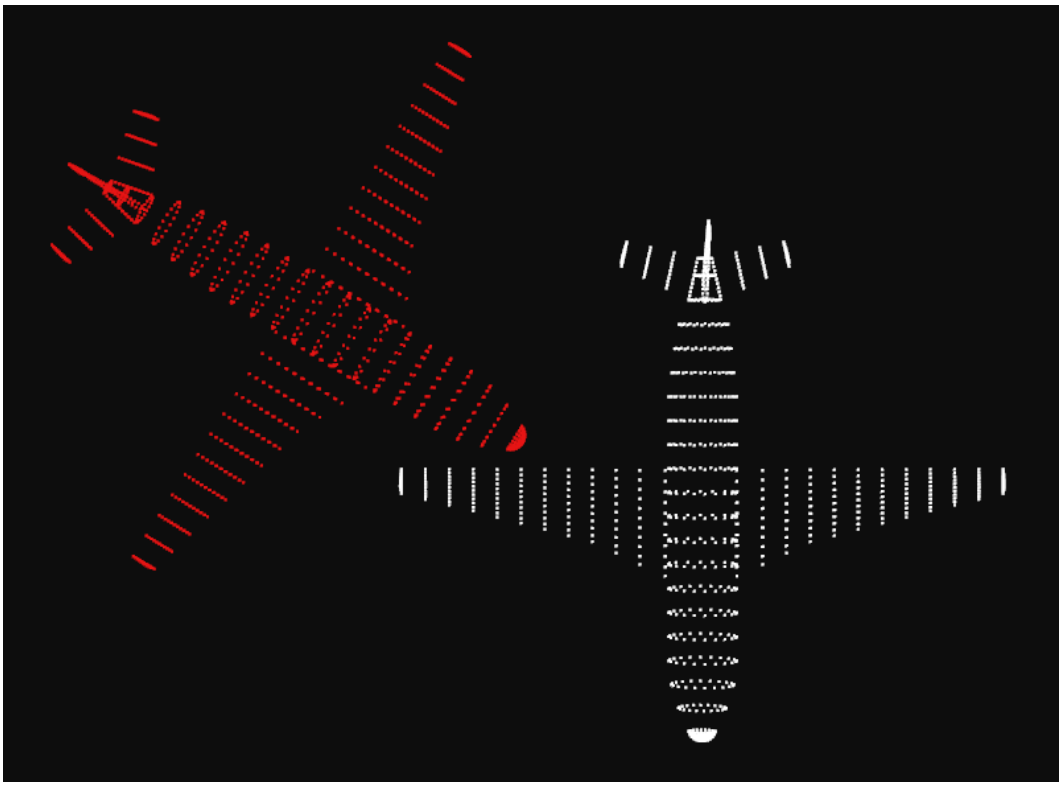


Figure 4.15: Airplane Point Cloud Transformation

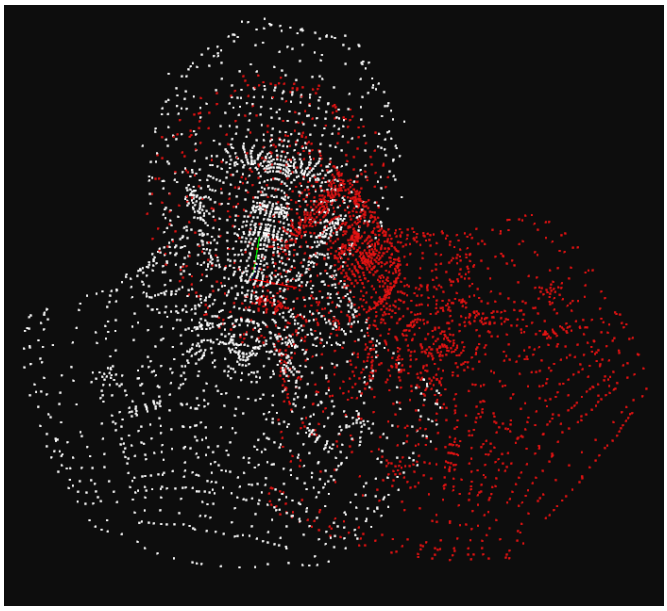


Figure 4.16: Beethoven Point Cloud Transformation

Table 4.4: Point Cloud Transformations Comparison

Number of Points	CPU code (ms)	NVIDIA GeForce 940 MX (ms)	Jetson Nano 2GB (ms)
41771	1581.5	1.3675	5.35
1335	43.6	0.0492	0.4703
1117	1520	0.0433	0.3667

Table 4.4 demonstrates the performance of the CPU(Intel i5-8250, 8 cores, 3.4 GHz) and GPU for transforming Point Clouds. The original Point Clouds are denoted using white coloured points and the transformed point cloud is red in colour. As can be seen the GPU(NVIDIA GeForce 940 MX) gives a 99.9135% decrease as compared to the CPU in processing time for a point cloud containing $41,771$ points. For the same case the Jetson Nano, gives a 99.6616% decrease in the execution time. This enormous difference in execution time shows that Point Cloud processing can be highly parallelized, which can give us an enormous computation gain as compared to sequential execution. Furthermore it can help in porting Point Cloud based algorithms to embedded platforms like the NVIDIA Jetson Nano.

4.1.6 Directed Unweighted Graphs

A graph is a common data structure that consists of a finite set of nodes (or vertices) and a set of edges connecting them. In real world, graphs are used to solve problems that involve representation of the problem space as a network. For instance networks include telephone networks, circuit networks, social networks etc. In directed unweighted graphs, nodes are connected by directional edges. Practically, these graphs emphasize on the existence of relationship among the nodes than the weight of their existence.[24],[30]
For the implementation done for this project, we have an unweighted directed graph, where each node has an initial local value. Here, each node sends a particular value to its neighbours, i.e. every node has to update the values associated with the neighbours of that particular node. There are three operations that can be performed on the neighbours of a particular node in this implementation which are, *Min*, *Max* and *Add*. When the *Max* operation is being performed the neighbours of the sender node compare their respective local values with the value being sent to them, if the value that is sent is greater than the local value of a particular neighbour or receiver node, then the receiver node changes its local value to the sent value. In the *Min* operation a similar logical flow exists, the only difference is that instead of the sent value being greater than the local value, the sent value should be lesser that the receiver node's local value to trigger an update. Whereas, in the *Add* operation the receiver nodes add the sent value to their local value.[26],[11]

Algorithm 7 Pseudocode for Directed Graph Add Operation

$node\_id \leftarrow blockIdx.x$
$num\_elements \leftarrow OA[node\_id + 1] - OA[node\_id]$
$current\_up \leftarrow currentupdate[node\_id]$
for $i = 0$ to $num\_elements$ **do**
    $index \leftarrow CA[i + OA[node\_id]]$
    Add $currentupdate[node\_id]$ to $locals[index]$
end for

Algorithm 7 demonstrates the logical flow for the *Add* Operation performed between nodes. Generally, directed graphs are represented as a sparse matrix. To exemplify, let's consider a simple directed graph which can be seen in Fig. 4.17. Now the sparse representation for this directed graph can be written as,

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.12}$$

Where each row represents a particular node's neighbours, i.e. since there is a 1 at index $(0, 1)$ it signifies that *node 2* is a neighbour of *node 1*. Similarly since there is a 1 at the index $(2, 3)$ it implies that *node 4* is a neighbour of *node 3*. All the places marked with a 0 signify that a connection between those corresponding nodes does not exist. This is the format in which we code directed graphs for a computer to work on. As can be seen from 4.12 this is a sparse matrix, i.e. there are a very few non-zero values. So, in this case instead of going about the calculations as we normally would with a matrix, which will result in a waste of computing resources and incur high cost in terms of execution time, we need to encode this matrix so that only the non-zero values are taken into account.[13] There are many ways in which this can be done, in this implementation the **CSR**(**C**ompressed **S**pasre **R**ow) format is used. In this representation the sparse matrix is encoded into 3 1D arrays, which are the **value** array, the **column index** array and the **row pointer** array. The names of these arrays help us understand their functions. The *value* array stores the non-zero values of the sparse matrix. The *column index* array stores the column index of the non-zero values stored in the *value* array. the *row pointer* array consists of the number of elements in a particular row, i.e $row\_pointer\_array[i + 1] - row\_pointer\_array[i]$ gives us the number of elements, which are the number of neighbours of a node if we think about it in terms of a directed graph. This array is also known as the the *Offset Array*, and the column array can be abbreviated as *CA*. These terms are used in Algorithm 7, 8 and 9. The execution flow in these algorithms are followed by each thread in a thread block. The kernels are launched with 1 thread per block and the number of blocks are equal to number of nodes. The *currentupdate* and *locals* arrays contain the values to be sent and the local values of each node respectively.

Algorithm 8 Pseudocode for Directed Graph Min Operation

$node\_id \leftarrow blockIdx.x$
$num\_elements \leftarrow OA[node\_id + 1] - OA[node\_id]$
$current\_up \leftarrow currentupdate[node\_id]$
for $i = 0$ to $num\_elements$ **do**
    $index \leftarrow CA[i + OA[node\_id]]$
    Find the minimum value between $currentupdate[node\_id]$ and $locals[index]$
    Update the value of $locals[index]$ according to the statement above
end for

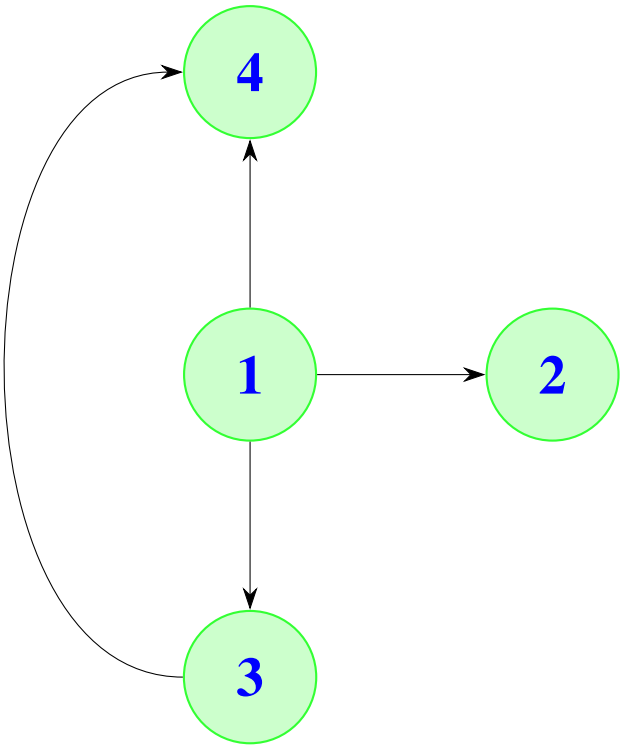


Figure 4.17: Directed Unweighted Graph

Algorithm 9 Pseudocode for Directed Graph Max Operation

$node\_id \leftarrow blockIdx.x$
$num\_elements \leftarrow OA[node\_id + 1] - OA[node\_id]$
$current\_up \leftarrow currentupdate[node\_id]$
for $i = 0$ to $num\_elements$ **do**
    $index \leftarrow CA[i + OA[node\_id]]$
    Find the maximum value between $currentupdate[node\_id]$ and $locals[index]$
    Update the value of $locals[index]$ according to the statement above
end for

Chapter 5

GPU Acceleration for CNN

5.1 CNN Architecture

In a convolution neural network when an image is fed to the model the convolution operation is applied to extract feature maps which are also called kernels. These filters extract important features from the image. After feature extraction the max-pooling operation is performed. The pooling layer is responsible for size reduction of convolved layer. This helps in reducing the computational power required in processing data. It is used to compress the feature map. After pooling is done flattening is performed where all the 2D pixels are converted into 1D and fed into the input of fully connected dense neural network.

5.2 Need for Custom CNN Inference

Although many platforms for CNN inference, like Tensorflow, PyTorch, Theano, etc. exist, the need for writing custom CUDA kernels, for this process arises when further optimization in the inference task is required. While using conventional tools like the ones stated before there is hardly any interference from the user in the Inference process, nor is there any room for optimizing the job. This is where CUDA comes into picture, where there is a huge amount that can be done as far as optimization is concerned. Not only does it provide insight into the inner workings of a neural network such as the CNN, but also allows us to deploy these models on resource constrained embedded devices like the *NVIDIA Jetson Nano*.

5.3 GPU Implementation

This section is focused on the implementation of Custom CUDA Kernels for CNN Inference. The main goal of this project is to develop a kernel for each of the major layers in a Convolutional Neural Network, which are the *Convolutional* Layer, the *Max-Pooling* Layer and the *Dense* Layer. As discussed in the previous chapters we have seen the way in which convolution operation has a huge potential for parallelization, and we want to leverage this fact and apply it to an application. This is where CNN comes into the picture. The Convolutional Layers in a CNN can be accelerated on the GPU, to improve performance during inference. This technique can also be applied to perform inference or deploy CNNs on embedded platforms like the *NVIDIA Jetson Nano*, which will help in low-cost, low-power computing and at the same time be portable.

Algorithm 10 demonstrates the execution flow for a single thread. This is followed by all the threads that are launched. The number of threads launched for the convolutional layer is equal to the number of pixels in the image. The method used here is very similar to Algorithm 2. Since, the Convolutional Layer boils down to performing convolution with the input and the filter. The main difference lies in the fact that the filters used are also multi-channeled like the input. So, instead of performing simple convolution *depth-wise* convolution needs to be performed, i.e. for each channel in the input there is a corresponding channel in the filter. To exemplify, let's consider that the 1st convolutional layer, has 32 channels, which means that when an input image consisting of 3 channels (R, G and B) is passed to the first convolution layer, each channel from the image has a separate filter that needs to be convolved with it. The 1st filter among the 32 filters contains three channels, so when the 1st filter is applied to the input image, the respective channel of the filter and the input image are convolved. This pattern of convolution is called depth-wise convolution. This computation is done for each of the multi-channel filter. Before

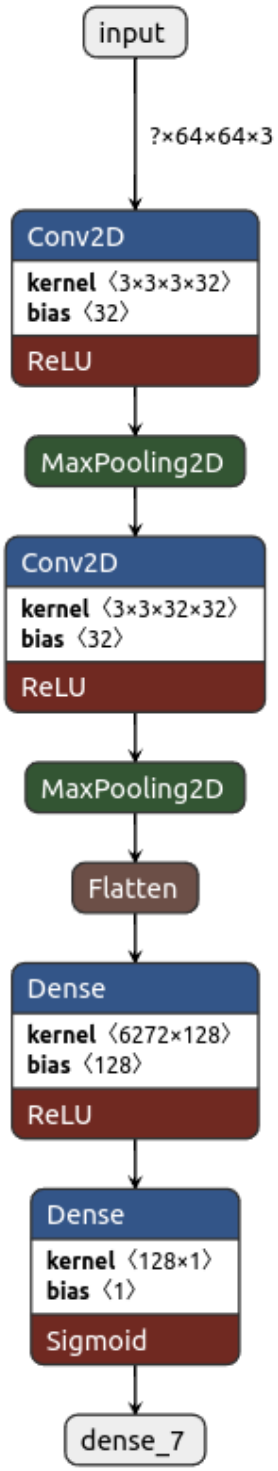


Figure 5.1: CNN Model Architecture

passing the input image to the CNN, it is normalized.

After the Convolution Layer, the output needs to be passed through a **ReLU** (**Re**ctified **L**inear **U**nit) function. This function gets rid of all the negative values from the output. The result obtained from this step, is passed to the *Max-Pooling* Layer, which reduces the output size in half. For this layer there is no filter as such, but we can imagine a frame traversing over the input and just keeping the value that is largest in that frame, all the other values are discarded, hence the reduction in output.

Algorithm 10 Pseudocode for Convolutional Layer

```
m ← num_rows
n ← num_cols
row ← threadIdx.y + (blockIdx.y × blockDim.y)
col ← threadIdx.x + (blockIdx.x × blockDim.x)
if row < m and col < n then
    num_rows_img ← m + (2 × (filter_width/2))
    num_cols_img ← n + (2 × (filter_width/2))
    prod = 0
    for i = 0 to num_filters do
        prod ← layer_bias[i]
        ch_index ← (i × num_channels × filter_width × filter_width)
        for j = 0 to num_channels do
            for k = 0 to filter_width do
                row_val ← row + k
                for l = 0 to filter_width do
                    col_val ← col + l
                    if row_val ≥ 0 and row_val < num_rows_img then
                        if col_val ≥ 0 and col_val < num_cols_img then
                            img_index ← ((row_val × num_cols_img) + col_val) × num_channels + j
                            filter_index ← (k × filter_width) + l
                            filter_index_1 ← (j × filter_width × filter_width) + ch_index
                            f_filter_ind ← filter_index + filter_index_1
                            prod ← prod + (in_img[img_index] × layer_weight[f_filter_index])
                        end if
                    end if
                end for
            end for
        end for
        img_index_out ← ((row × n) + col) × num_filters + i
        Perform ReLU Operation
        if prod ≤ 0 then
            layer_conv_out[img_index_out] ← 0
        else
            layer_conv_out[img_index_out] ← prod
        end if
    end for
end if
```

Algorithm 11 highlights the executional flow at a single thread level. All the threads that are launched perform these operations parallely. The reason for the reduction of the output to size to such an extent, is that the frame that was discussed above moves across the input with a stride greater than one.

These are some of the most important layers in any CNN, now that the implementation of the kernels for these two layers is done, we can call them for however many times, as required based on the depth of the CNN. After all the Convolutional and Max-Pooling Layers are done, we need to flatten the output from the last layer into a 1D layer, which is the *Dense* Layer.

The CNN Model used for the purposes of this project contains two dense layers, the 1st one is after the flattening is done, and then we have the 2nd Dense Layer that reduces the output to a single neuron, upon which the *Sigmoid*

Activation function is applied, to obtain the final prediction. Algorithm 12 showcases the pseudo-code for the 1st Dense Layer that is placed after flattening the output. This execution flow pertains to a single thread, and all the threads launched follow this execution pattern parallely.[9]

The Sigmoid Activation Function is given by,

$$S(x) = \frac{1}{1 + e^{-x}} \tag{5.1}$$

Algorithm 11 Pseudocode for Max-Pooling Layer

$m \leftarrow num\_rows\_after\_max\_pool$
$n \leftarrow num\_cols\_after\_max\_pool$
$row \leftarrow threadIdx.y + (blockIdx.y \times blockDim.y)$
$col \leftarrow threadIdx.x + (blockIdx.x \times blockDim.x)$
if $row < m$ and $col < n$ **then**
  $start\_row \leftarrow (row \times stride)$
  $start\_col \leftarrow (col \times stride)$
  for $i = 0$ to $num\_channels$ **do**
    $max\_val = 0$
    for $k = 0$ to max_pool_size **do**
      for $l = 0$ to max_pool_size **do**
        $row\_val \leftarrow start\_row + k$
        $col\_val \leftarrow start\_col + l$
        if $row\_val \geq 0$ and $row\_val < num\_rows\_after\_conv$ **then**
          if $col\_val \geq 0$ and $col\_val < num\_cols\_after\_conv$ **then**
            $img\_index \leftarrow ((row\_val \times num\_cols\_after\_conv) + col\_val) \times num\_channels + i$
            if $max\_val \leq layer\_prev\_conv\_out[img\_index]$ **then**
              $max\_val \leftarrow layer\_prev\_conv\_out[img\_index]$
            end if
          end if
        end if
      end for
    end for
    Write the Output
    $img\_index\_out \leftarrow ((row \times n) + col) \times num\_filters + i$
    $layer\_pool\_out[img\_index\_out] \leftarrow max\_val$
  end for
end if

As far as the ReLU function is concerned, it can be expressed in mathematical terms as follows,

$$f(x) = \begin{cases} x, & x > 0 \\ 0, & otherwise \end{cases} \tag{5.2}$$

After this the final layer or step that remains is to implement the kernel for the final dense layer, whose logical flow can be seen in Algorithm 13 where each of the thread that is launched follows this pattern, so that the computations can be performed parallely.

Algorithm 12 Pseudocode for the 1st Dense Layer

$tid \leftarrow threadIdx.x + (blockIdx.x \times blockDim.x)$
if $tid < num\_dense\_elements$ **then**
  $prod \leftarrow layer\_bias[tid]$
  for $i = 0$ to $num\_flattened\_elemets$ **do**
    $index \leftarrow (tid \times num\_flattened\_elemets) + i$
    $prod \leftarrow prod + layer\_conv\_out[i] \times layer\_wt[index]$
  end for
  Perform ReLU Operation
  if $prod > 0$ **then**
    $pred[tid] \leftarrow prod$
  else
    $pred[tid] \leftarrow 0$
  end if
end if

Algorithm 13 is a very simple and small kernel that just performs point-wise multiplication between the output from the 1st Dense Layer and the weights. The 2nd Dense Layer gives us an array of values that needs to be summed up, which is done on the host. After, the sum is computed the Sigmoid Activation function is applied to it, which helps us scale the output.

Algorithm 13 Pseudocode for the 2nd Dense Layer

$tid \leftarrow threadIdx.x + (blockIdx.x \times blockDim.x)$
if $tid < num\_dense\_elements$ **then**
  $final\_pred[tid] \leftarrow layer\_prev\_out[tid] \times layer\_weight[tid]$
end if

Results

5.3.0.1 Prediction for Dogs

Figure 5.2: Dog Prediction 1

Figure 5.3: Dog Prediction 2

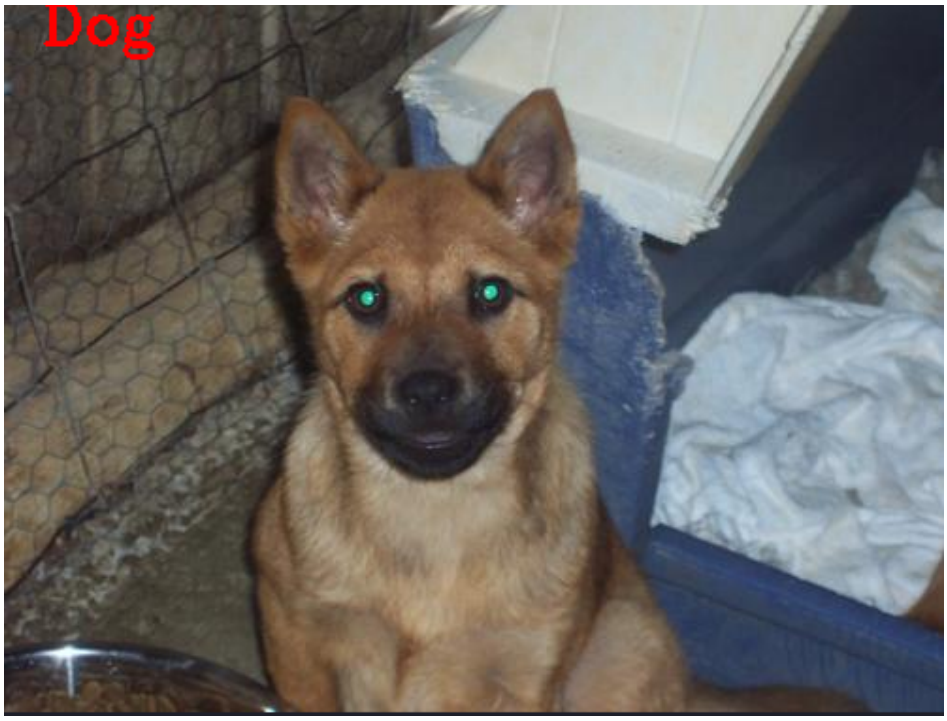


Figure 5.4: Dog Prediction 3

5.3.0.2 Prediction for Cats

Figure 5.5: Cat Prediction 1

Figure 5.6: Cat Prediction 2

Figure 5.7: Cat Prediction 3

Table 5.1: Image Classifier CNN Inference Comparison

Size	CPU code (ms)	NVIDIA GeForce 940 MX (ms)	Jetson Nano 2GB (ms)
356 x 232	85,995.12	38.8324	92.066
393 x 500	86,019.49	38.305	93.800
499 x 500	85,998.87	40.982	93.973
1110 x 619	86,326.54	45.228	94.895
1920 x 1080	88,989.95	45.643	89.088

Table 5.1 shows us the Inference time for the Cats-vs-Dogs Model, on the CPU(Intel i5-8250, 8 cores, 3.4 GHz), GPU (NVIDIA GeForce 940 MX) and NVIDIA Jetson Nano (2GB). As can be seen there is a 99.8548% decrease in execution time as compared to the CPU. This is because of the vast amount of calculations that are sequential implemented on the CPU, are highly parallelized on the GPU. This difference in performance will be increasingly prominent as the Size of the Convolutional Neural Network increases, i.e. as the number of layers in the CNN increase the performance gap between the GPU and the CPU will also increase. As far as Jetson Nano is concerned we get a 99.8848% decrease in execution time as compared to the CPU.

Chapter 6

Conclusion

So far, the tasks carried out involve a huge number of computations per unit time. This takes a lot of time for the processing that happens on the CPU. Whereas the same data processing on the GPU happens in a fraction of time of what it takes on the CPU. This huge speed difference is attributed to the sequential processing that happens on the CPU in contrast to the massively parallel data crunching performed by the GPU. However, this vast speed-up is visible only on large amounts of data.

Further optimizations like using the different memory paradigms viz. shared memory, texture memory etc. can vastly improve the present performance to run the inference in near real-time speed. This real-time implementation provides a very promising future for future works in the field of autonomous driving, due to time-critical computations that need to be performed. An autonomous vehicle is constantly gathering a huge amount of data from the LIDAR, the cameras, it's various sensors, etc. To process all of this data in such a manner that the car can take action in real-time it is essential that we leverage the parallel compute capability of the GPU.

Most of the current autonomous cars employ NVIDIA's powerful Tegra based GPUs, like the NVIDIA Jetson Xavier, which are termed as edge devices, that are used as a platform for AI inference. The importance of these edge devices is growing day by day, due to the need for offline real-time computations that are required to automate many of the tasks. Furthermore, due to their low-power, low-cost and portable nature they can be embedded into various products.

The goal of this project was to perform CNN inference on edge devices, by writing custom CUDA kernels, that can be optimized to perform the vast computations that are needed by each and every layer of a Deep Neural Network. Although NVIDIA provides frameworks like Tensor-RT and cuDNN, which are used to accelerate ML model inference, the user has to rely on the inbuilt functions provided by these libraries. On the other hand, when we write our owm CUDA kernels, the possibilities for tweaking and optimizing the inference process are limitless. This sort of control over the code, can help us build kernels tailored towards specific platforms.

As far as inference on edge devices is concerned, we can improve it's performance multi-fold by utilizing weight quantization process. The weights can be transformed to *floating-point 16-bit precision* from *32-bit floating point*. This can help in drastically reducing the execution time and provide a good performance boost, especially on resource-constrained devices. To take this idea further, we can then quantize the weights to *int8*. Since, int based operations are much faster and easier to perform this will give another significant boost in performance.

Combining all of the techniques mentioned above, like utilizing shared memory, texture memory, quantization of weights, can be considered as a future work for this project. Once we achieve real-time DNN (Deep Neural Network) inference, on edge devices especially, the applications for such a technology is boundless. It can be employed in unmanned aerial vehicles that are small in size, which can have huge implications in the defence industry, since these tiny drones can perform surveillance while being completely oblivious to the people around it. It can also be employed in search and rescue operations conducted after major natural calamities like Earthquakes, landslides, etc. All of this along with the power of AI can really transform the world that we currently live in.

Bibliography

[1] Mouna Afif, Yahia Said, and Mohamed Atri. Efficient 2d convolution filters implementations on graphics processing unit using nvidia cuda. *IJ Image, Graphics and Signal Processing*, (8):1–8, 2018.

[2] Cristhian A. Aguilera, Cristhian Aguilera, Cristóbal A. Navarro, and Angel D. Sappa. Fast cnn stereo depth estimation through embedded gpu devices. *Sensors*, 20(11):3249, Jun 2020.

[3] Bhaskar Anand, Anuj G Patil, Mrinal Senapati, Vivek Barsaiyan, and P Rajalakshmi. Comparative run time analysis of lidar point cloud processing with gpu and cpu. In *2020 IEEE International Conference on Computing, Power and Communication Technologies (GUCON)*, pages 650–654. IEEE, 2020.

[4] Benjamin Planche Eliot Andres. *Hands-On Computer Vision with TensorFlow 2*. Machine Learning series. Packt Publishing Ltd, 2019.

[5] FRANÇOIS CHOLLET. *Deep Learning with Python*. Machine Learning series. Manning Publications Co, 2017.

[6] Subhasis Das and Song Han. Neuraltalk on embedded system and gpu-accelerated rnn. 2015.

[7] Robert Laganiere David Millán Escrivá. *OpenCV 4 Computer Vision Application Programming Cookbook*. Image Processing. Packt Publishing Ltd, 2019.

[8] Adam Dziekonski, Piotr Sypek, Adam Lamecki, and Michal Mrozowski. Accuracy, memory, and speed strategies in gpu-based finite-element matrix-generation. *IEEE Antennas and Wireless Propagation Letters*, 11:1346–1349, 2012.

[9] Benjamin Eckart, Kihwan Kim, Alejandro Troccoli, Alonzo Kelly, and Jan Kautz. Accelerated generative models for 3d point cloud data. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5497–5505, 2016.

[10] Ahmed Elliethy and Gaurav Sharma. Accelerated parametric chamfer alignment using a parallel, pipelined gpu realization. *Journal of Real-Time Image Processing*, 16, 10 2019.

[11] Al Ghadani, Ahmed Khamis Abdullah, Waleeja Mateen, and Rameshkumar G Ramaswamy. Tensor-based cuda optimization for ann inferencing using parallel acceleration on embedded gpu. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pages 291–302. Springer, 2020.

[12] Fabian Groh, Patrick Wieschollek, and Hendrik Lensch. Flex-convolution (million-scale point-cloud learning beyond grid-worlds). *arXiv preprint arXiv:1803.07289*, 2018.

[13] Jongmin Jo, Sucheol Jeong, and Pilsung Kang. Benchmarking gpu-accelerated edge devices. In *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 117–120. IEEE, 2020.

[14] Marc Jordà, Pedro Valero-Lara, and Antonio J Peña. cuconv: Cuda implementation of convolution for cnn inference. *Cluster Computing*, pages 1–15, 2022.

[15] Jason Sanders Edwarrd Kandrot. *CUDA by example : an introduction to general-purpose GPU programming*. GPU acceleration. Addison-Wesely, 2011.

[16] Beena Kumari, Avijit Ashe, and Jaya Sreevalsan-Nair. Remote interactive visualization of parallel implementation of structural feature extraction of three-dimensional lidar point cloud. In *International Conference on Big Data Analytics*, pages 129–132. Springer, 2014.

[17] Seyyed Salar Latifi Oskouei, Hossein Golestani, Matin Hashemi, and Soheil Ghiasi. Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android. In *Proceedings of the 24th ACM international conference on Multimedia*, pages 1201–1205, 2016.

[18] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008.

[19] David B. Kirk Wen mei W. Hwu. *Programming Massively Parallel Processors A Hands-on Approach*. GPU acceleration. Katey Birtcher, 2010.

[20] Victor Podlozhnyuk. Image convolution with cuda. *NVIDIA Corporation white paper, June*, 2097(3), 2007.

[21] Richard E. Wood Rafael C. Gonzalez. *Digital Image Processing*. Image Processing. Pearson Education Limited, 2017.

[22] Shivani Raghav, Martino Ruggiero, Andrea Marongiu, Christian Pinto, David Atienza, and Luca Benini. Gpu acceleration for simulating massively parallel many-core platforms. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1336–1349, 2015.

[23] Remya Ramakrishnan, Aditya K V Dev, A S Darshik, Renuka Chinchwadkar, and Madhura Purnaprajna. Demystifying compression techniques in cnns: Cpu, gpu and fpga cross-platform analysis. In *2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID)*, pages 240–245, 2021.

[24] Rico Richter, Jan Eric Kyprianidis, and Jürgen Döllner. Out-of-core gpu-based change detection in massive 3 d point clouds. *Transactions in GIS*, 17(5):724–741, 2013.

[25] Muhammad T. Satria, Bormin Huang, Tung-Ju Hsieh, Yang-Lang Chang, and Wen-Yew Liang. Gpu acceleration of tsunami propagation model. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 5(3):1014–1023, 2012.

[26] Jaegeun Han Bharatkumar Sharma. *Learn CUDA Programming*. CUDA. Packt Publishing Ltd, 2019.

[27] Mingcong Song, Yang Hu, Yunlong Xu, Chao Li, Huixiang Chen, Jingling Yuan, and Tao Li. Bridging the semantic gaps of gpu acceleration for scale-out cnn-based big data processing: Think big, see small. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 315–326, 2016.

[28] Larisa Stoltzfus, Murali Emani, Pei-Hung Lin, and Chunhua Liao. Data placement optimization in gpu memory hierarchy using predictive modeling. In *Proceedings of the Workshop on Memory Centric High Performance Computing*, pages 45–49, 2018.

[29] Bhaumik Vaidya. *Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA*. CUDA. Packt Publishing Ltd, 2018.

[30] Jing Xin, Guo Xie, Bo Yan, Mao Shan, Peng Li, and Kaiyuan Gao. Multimobile robot cooperative localization using ultrawideband sensor and gpu acceleration. *IEEE Transactions on Automation Science and Engineering*, pages 1–12, 2021.

[31] Xun Zeng and Wei He. Gpgpu-based parallel processing of massive lidar point cloud. In *MIPPR 2009: Medical Imaging, Parallel Processing of Images, and Optimization Techniques*, volume 7497, pages 300–305. SPIE, 2009.

Curriculum Vitae

Name:	Ruturaj A. Nanoti
Father's Name:	Ashutosh A. Nanoti
Date of Birth:	16th September, 2000
Nationality:	Indian
Sex:	Male
Company Placed:	-
Permanent Address:	1108, Saarrthi Success Square, Karve Road, Kothrud, Pune, Maharashtra- 411038
Phone Number:	8451071616
Mobile Number:	8451071616
E-mail ID:	nanotiruturaj@gmail.com

CGPA : 9.32

Examinations Taken:

1. GRE
2. TOEFL

Placement Details: NA

Curriculum Vitae

Name:	Amitvikram S. Pujar
Father's Name:	Sanjeev V. Pujar
Date of Birth:	27/11/2000
Nationality:	Indian
Sex:	Male
Company Placed:	Cognizant Technological Services
Permanent Address:	Building no. M-2,6th floor, Spaghetti CHS, Sector-15, Kharghar Navi Mumbai-410210
Phone Number:	+91-8652251127
Mobile Number:	+91-8652251127
E-mail ID:	amitvikram.pujar@gmail.com

CGPA : 9.12

Examinations Taken:

1. GATE

Placement Details: Cognizant Technological Services

Curriculum Vitae

Name:	Nishith Nayan
Father's Name:	Mrityunjay Kumar
Date of Birth:	30th June, 1999
Nationality:	Indian
Sex:	Male
Company Placed:	-
Permanent Address:	801/H4/A,Amba Nagar, Pandra, Ratu Road Ranchi, Jharkhand,834005
Phone Number:	7667841300
Mobile Number:	7667841300
E-mail ID:	nishithnayan30@gmail.com

CGPA : 8.02

Examinations Taken:

NA

Placement Details: Unplaced

Capstone Project

Project Title	GPU Acceleration for CNN Based Applications
Team Members	Ruturaj A. Nanoti (18BEE0134), Amitvikram S. Pujar (18BEE0135), Nishith Nayan (18BEE0166)
Faculty Guide	Prof. K Selvakumar
Semester/Year	VIII/ IV year
Project Abstract	This project focuses on accelerating Deep Learning inference by utilizing the massively parallel computation that is characteristic of the Graphics Processing Unit (GPU). With increasing demand of Artificial Intelligence (AI) especially in the field of Computer Vision, it is now the need of the hour to port this application to edge devices. These devices then can be integrated with different subsystems to create something novel like autonomous vehicles. The work carried out under this project focuses on implementing the fundamental building blocks of the neural networks that are instrumental in achieving the goal. Also, some data processing is done on the LiDAR based point-cloud data, that are used in autonomous vehicles. The aim here is to have custom defined elements that can be optimized to a great extent, thereby achieving greater accuracy in prediction.
Project Title	GPU Acceleration for CNN Based Applications
List **codes** and **standards** that significantly affect your project.	IEEE 754-2008 : Standard for Binary Floating-Point Arithmetic IEEE2413 : Standard for an Architectural Framework for the Internet of Things (IoT)
List at least two significant **realistic design constraints** that are applied to your project.	1. Images captured from the camera systems are very huge and therefore are very heavy on computation. Thus, the images are re-scaled and then normalized for efficient computation. 2. While training the Deep Learning model, few values are dropped by setting the suitable drop-out rate. This is done to ensure that the model does not overfit to the training data.
Briefly explain two **significant trade-offs** considered in your design, including options considered and the solution chosen	1. Resource constrained real-time inference & accurate prediction for writing efficient custom CUDA kernels. 2. Using the on-chip shared memory which is significantly lower in size than DRAM or global memory for faster execution times.
Describe the **computing aspects, if any**, of your project. Specifically identifying **hardware-software** trade-offs, interfaces, and/or interactions	1.The CPU used for performance analysis is Intel i5-8250 with 8 cores clocked at 3.4 GHz along with 8 GB RAM. The GPU used is NVIDIA GeForce 940MX with 2 GB of dedicated memory having a CUDA compute capability of 5.0. The embedded Jetson Nano platform has 128 cores with 2GB of total memory size with a CUDA compute capability of 5.3. 2. CUDA is the API provided by NVIDIA to interface with the underlying GPU hardware. The image processing library used for image transformations is OpenCV. Whereas, PCL library is the framework used for processing Point Cloud data.