ENABLING ARCHITECTURE RESEARCH ON GPU SIMULATOR FOR DEEP LEARNING APPLICATIONS

by

Abhishek D Nikam

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2018

Approved by:

_____
Dr.Hamed Tabkhi

_____
Dr. Erik Saule

_____
Dr. James Conrad

ABSTRACT

ABHISHEK D NIKAM. Enabling Architecture Research on GPU Simulator for Deep Learning Applications. (Under the direction of DR.HAMED TABKHI)

Deep learning uses stacks of multiple processing layers to learn representations of data with different levels of abstraction. It enables machines to have the understanding of outer environment just like the human body, opening a path for diverse range of applications like autonomous control of a self driving car or monitoring a certain set of devices. Convolutional Neural Networks (CNN), arguably the most popular deep learning architecture, consists of multiple convolutional and pooling layers stacked on each other. The convolutional layer is used for capturing the features of an image while the pooling layers help reduce the number of parameters involved. With deep learning consisting of multiple processing layers to learn the representation of data the computation involved in it is intense.

Graphical Processing Units work on a Single Thread Multiple Instruction execution model and the uniform structure of each layer in convolutional neural network fits well with the computations GPUs performs efficiently. GPU simulators are a useful tool in making architectural modifications to GPU hardware.

In this research, we explore the challenges involved in making Convolutional Neural Networks compatible with latest versions of GPU simulators. We develop optimized GPU kernels (GPU code) and integrate it with Darknet deep learning framework which help us explore the actual hardware bottlenecks affecting GPU performance for deep learning applications. We also do Architecture modeling of embedded GPUs and performance comparison with actual GPUs to verify the accuracy of our architecture modeling specially for deep learning applications.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Tabkhi for his immense help and guidance throughout my entire research and college career. His patience and expertise in the field served as an inspiration for me to complete the research.

I would also like to thank Dr. Saule for his contribution in improving my general coding style and inspiring me to think in a parallel way of solving problems.

I also take an opportunity to thank Dr. Conrad for his constant encouragement and insightful comments.

Finally, I thank Dr. Tabkhi and the Department of Electrical Engineering at UNC Charlotte for providing me with a state of art research lab and high-end GPUs.

DEDICATION

This thesis is dedicated to the open source community which provided me with the software tools that helped me complete my research and also my parents without whom this could not have been possible. .

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATION

API Application Program Interface

ASIC Application-Specific Integrated Circuit

BLAS Basic Linear Algebra Subprograms

CNN Convolutional Neural Net

CUDA Compute Unified Device Architecture

cuBLAS Cuda Basic Linear Algebra Subprograms

GEMM General Matrix Multiply

GPU Graphic Processing Unit

HPC High Performance Computing

OpenCL Open Computing Language

PTX Parallel Thread Exceution

Relu Rectified Linear Unit

SIMT Single Instruction Multiple Thread

SM Streaming Multiprocessor

TPU Tensor Processing Unit

CHAPTER 1: INTRODUCTION

With deep learning enabling human-like understanding of the physical environment in machines, it has been used in a wide range of applications like voice recognition, search-related product recommendation, drug discovery and language translation. Tech giants like Google, YouTube, Quora, Amazon and Facebook are investing heavily in deep learning. Google's search engine, voice recognition system and self-driving cars all rely heavily on deep learning while YouTube used deep learning to choose an attractive thumbnail from a video.

With deep learning becoming a quintessential part of human life, there has been a lot of focus on improving the performance of deep learning applications. A lot of Neural net architectures have been introduced after the initial success of Multi-layered perceptron (MLP)[1]. Convolutional neural networks, Deep belief networks, Recurrent neural networks were some of the deep learning architectures which were introduced to make deep learning applicable and useful for a wider range of applications.

With convolutional neural networks arguably being the most widely used and effective deep learning architecture for image detection there has been a consistent effort from the research community to come up with newer, more accurate and less compute-intensive models for convolutional neural nets. The paper titled "ImageNet Classification with Deep Convolutional Networks" introduced Alex net model which created a large, deep convolutional neural network for image detection. It was one of the most influential publication in the field of image classification with deep convolutional neural nets. The Alex net model is shown in Fig 1.1

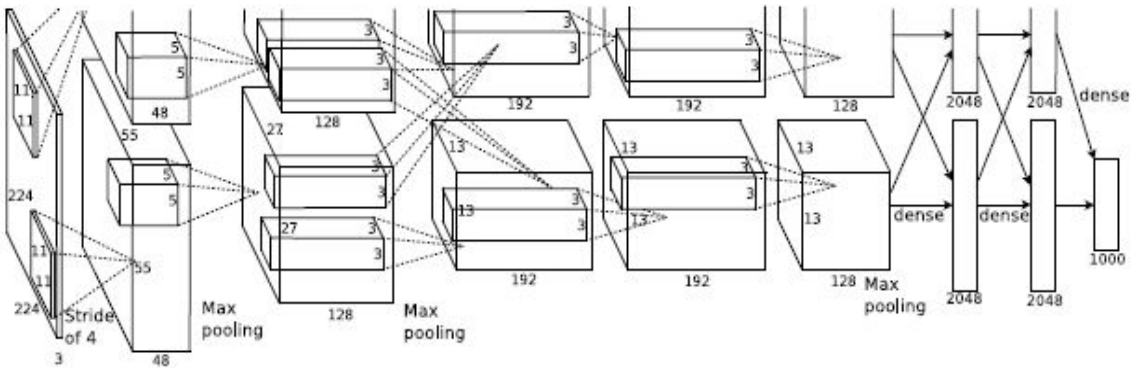


Figure 1.1: Alex Net Architecture

Source-https://medium.com/@smallfishbigsea/a-walk-through-of-alexnet-6cbd137a5637

Squeezenet [25] which was introduced just 3 years after Alex net had the same accuracy as Alex Net.[2] with almost 50 times fewer parameters involved. Such advances in convolutional neural networks model made them feasible for FPGA and embedded systems deployment.

Along with making convolutional neural network model's better for accuracy and reducing the parameters involved, there is also an active interest in the chip manufacturing community to make better hardware for deep learning applications. Chip manufacturing giants like Intel, Google and Nvidia are coming up with new design architecture and hardware specifically focused on deep learning. Along with having single precision and double precision cores for arithmetic operation hardware like Google's Tensor processing unit shown in Fig 2.1 includes a dedicated unit for each functional layer of a convolutional neural network. Even GPU design has been modified with the surge in deep learning, Nvidia's Volta GPUs [3] come with specialized Tensor cores which are useful in accelerating the convolutional layer of deep neural networks. Fig 1.2 shows the Volta Streaming Multiprocessor architecture. So, in order to have low-power real-time image detection, we must support an efficient convolutional neural network model with equally efficient hardware.

Figure 1.2: Nvidia Volta Streaming Multiprocessor Microarchitecture

Source+http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

1.1 Motivation

For real-time object detection in self-driving cars and drones, we need hardware which can do faster computations with low-power requirements. Traditional server class GPUs are capable of doing millions of computations within quick time but their high-power requirements and bulky size makes them inefficient to employ in drones or embedded devices which need real-time object detection. Embedded GPUs are low power, light-weight portable GPUs with respectable performance compared to the server class GPUs. There is always a trade-off between the processing capability and power consumption while designing embedded GPUs and in most cases, performance is sacrificed to abide with the power constraint. The main aim of embedded GPUs is to provide a considerable performance in a low power environment.

Deep learning consists of two phases, training and inference. In the training phase shown in Fig 1.3 the network tries to learn with the help of data. We have multiple forward and backward propagations during the training phase. In a forward pass with the help of the random weights assigned we predict the class labels and scores. In a backward pass, we compare the class labels with actual labels and the error calculated is backpropagated. Training is computationally intensive because only a single weight update is done in a complete pass through all samples.

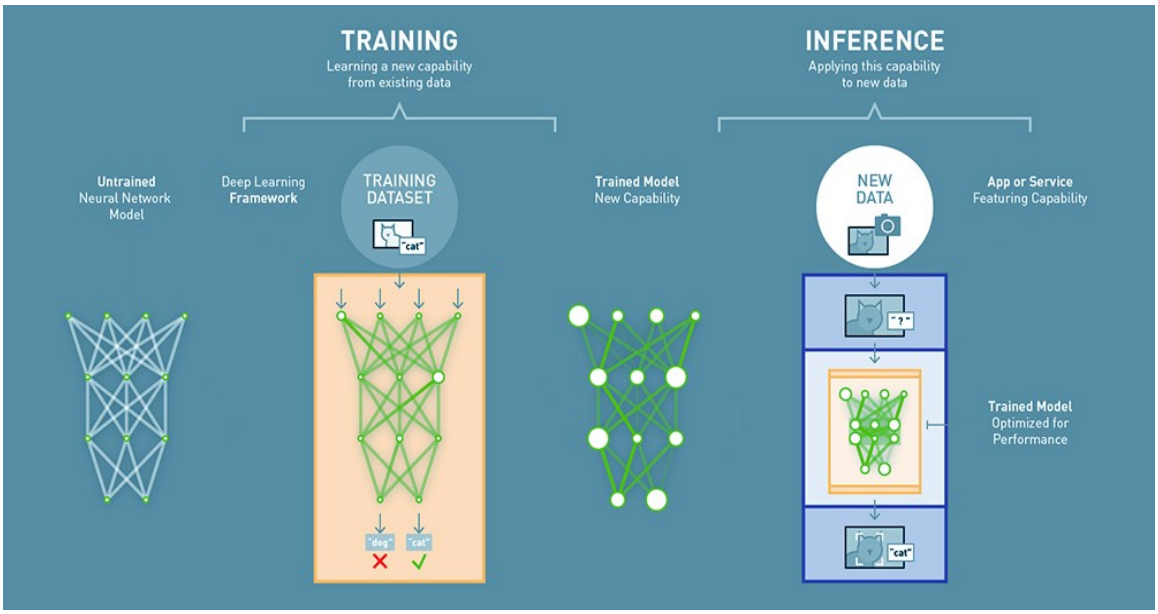


Figure 1.3: Deep Learning Training Phase

Source-http://blog.exxactcorp.com/discover-difference-deep-learning-training-inference/

Inference shown in fig 1.4 requires only one forward pass as we are already working on a trained model. We do not have a backward pass in inference phase as we do not calculate or backpropagate any error.

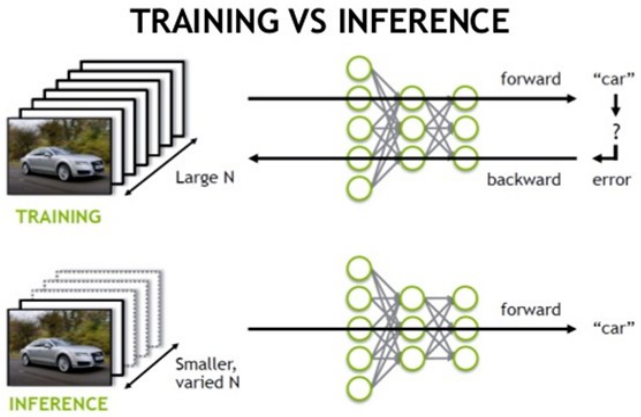


Figure 1.4: Deep Learning Training vs Inference

Source-http://blog.exxactcorp.com/discover-difference-deep-learning-training-inference/

Embedded GPUs have only 1 or 2 (depending on the architecture) streaming multiprocessors (explained in section 3.1.2) to account for low power usage. Training is generally done on a batch of images at a time, embedded GPUs because of their low memory can only process a few images per second. Also, the training phase requires a lot of forward and backward passes, embedded GPUs overall performance falls short of the server class GPUs owing to the large amount of computations involved. For inference, the performance goals are different. In inference phase only one forward pass is required, to minimize the end to end times, inference uses small batch sizes ideally 1 image at a time. Inference focuses on reducing the latency than throughput. The following table shown in Fig 1.5 is taken from the Nvidia's "GPU-Based Deep Learning Inference" white paper [4] .

Network: GoogLeNet	Batch Size	Titan X (FP32)	Tegra X1 (FP32)	Tegra X1 (FP16)
Inference Performance	1	138 img/sec	33 img/sec	33 img/sec
Power		119.0 W	5.0 W	4.0 W
Performance/Watt		1.2 img/sec/W	6.5 img/sec/W	8.3 img/sec/W
Inference Performance	128 (Titan X) 64 (Tegra X1)	863 img/sec	52 img/sec	75 img/sec
Power		225.0 W	5.9 W	5.8 W
Performance/Watt		3.8 img/sec/W	8.8 img/sec/W	12.8 img/sec/W

Figure 1.5: Performance comparison of Embedded and Server class GPUs for GoogLeNet inference

Source- https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson-tx1-whitepaper.pdf

This table shows the results of GoogLeNet inference on Jetson TX1 an embedded GPU and Titan X a server class GPU. Even though the number of images processed per second is not comparable, the power consumed by the embedded GPUs is far less and also, the performance per watt is much better. Hence embedded GPUs can be a good platform for applications involving low-power but when we need an extremely fast inference for applications such as real time object detection, embedded GPUs performance seems incompetent. To make embedded GPUs fast enough for applica-

tions like real-time object detection, hardware bottlenecks affecting the performance should be recognized. This motivates my research objective of running convolutional neural networks on embedded GPUs simulated GPU simulator, observing the hardware bottlenecks affecting the performance and to come up with architectural changes to accelerate deep learning inference on embedded GPUs.

1.2 Problems

1. GPU simulator is a good place to start for modeling different GPUs and modifying their architecture. We use GPGPU sim [5], a cycle-level GPU performance simulator that focuses on "GPU computing" for our research. CUDA [6] is a parallel computing platform which was created by Nvidia to program their GPUs, CUDA C is the programming language used for working with Nvidia GPUs. GPGPU sim relies on extracting the PTX (Nvidia's pseudo assembly language) out of the Cuda C source code as shown in fig 1.6. Nvidia's closed source libraries used by various deep learning frameworks comes in the form of a binary which cannot be disassembled for extracting the PTX. The inability of extracting PTX out of the Nvidia's provided binaries makes GPGPU sim incompatible for running deep learning frameworks.

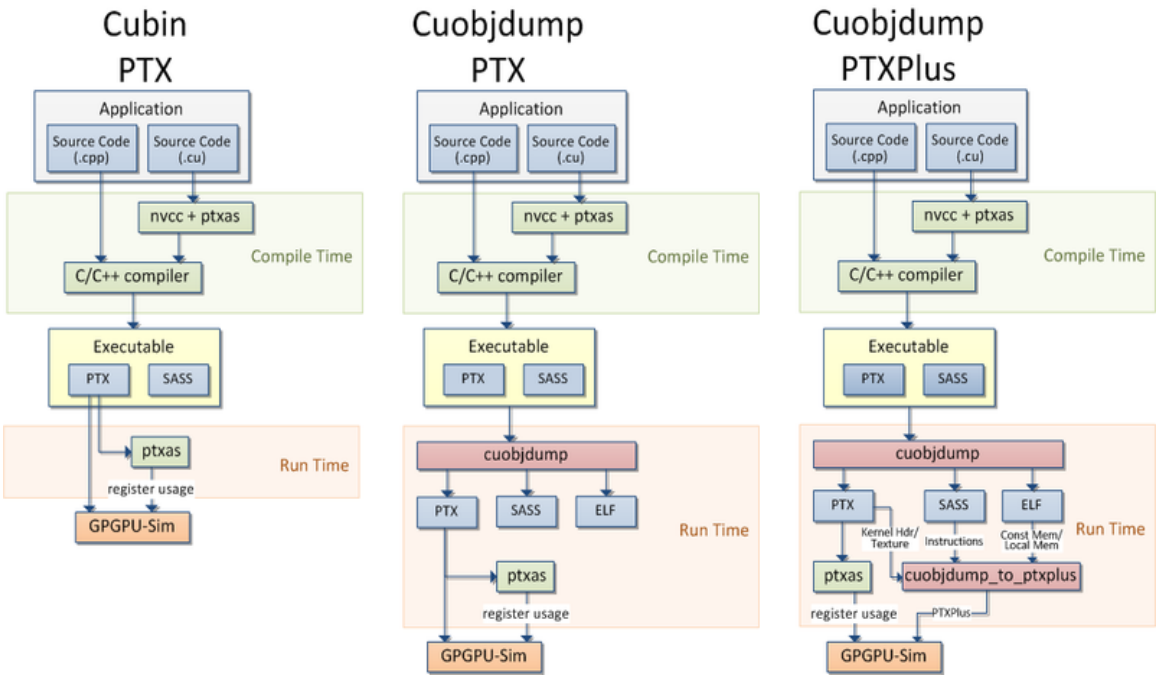


Figure 1.6: Compilation flow of PTX and PTX plus

Source-http://gpgpu-sim.org/manual/index.php/Main-Page

GPGPU simulators cannot run a deep learning framework which uses Python

or Java wrappers.

2. GPGPU sim developers till now have limited their architecture modeling for gaming and high-end GPUs like Nvidia GeForce GTX 1080 TI, thus there is no support for modeling embedded GPUs.

1.3 Contribution

1. As GPU simulators are incompatible with deep learning frameworks because of Nvidia's closed source libraries as discussed in section 1.2, my first contribution was to come up with open source libraries to replace Nvidia's closed source one. The custom library which was developed was programmed in CUDA C and the code was optimized for performance. The replacement of Nvidia's closed source library with custom library paved the way for enabling GPU architecture research using GPU simulators for deep learning applications.

2. We integrated our open source implementations of the library and CLblast [7], the only open source CUDA GEMM library (General Matrix Multiplication) publicly available with darknet deep learning framework. Extensive performance benchmarking was done for darknet's performance with the custom library, CLblast and Nvidia's closed source cuBLAS library on Nvidia's Jetson TX1 and Jetson TX2 embedded GPUs. The results obtained are explained in the section 5.2.1.

3. Introduced the GPU simulator to embedded domain by doing architecture modeling of Nvidia Jetson TX1 and Jetson TX2 GPUs which is explained in detail in the section 4.5. Yolo Net[8] for image detection was the application we chose to benchmark on the actual hardware platform of Jetson TX1 and Jetson TX2 as well the simulated TX1 and TX2 platform.

CHAPTER 2: RELATED WORK

2.1 Architectural Advancements for Accelerating Deep Learning Applications

This research has been inspired by many current and previous work done on accelerating deep learning for real time inference. With ever so increasing demand for faster deep learning hardware, ASICs (application-specific integrated circuits) specially designed for deep learning applications are employing novel architectural ideas for faster computations with lower power consumption.

Companies like Google uses deep learning for almost all their important services like search, street view, photos etc. With heavy reliance on deep learning for almost all its services, Google launched Tensor Processing Unit (TPU) [9] a customized ASIC to accelerate the convolutional neural network computations. TPUs are tailor made for deep learning applications such that it has a dedicated hardware unit for each unique layer in a convolutional neural network. TPU is shown in fig 2.1.

TPUs have a customized Matrix multiplier Unit (MXU) specially dedicated for Matrix multiplication operation which considerably accelerates the convolutional layer. Activation Unit is specifically customized to apply various activation functions on the input parameters while the pooling unit is dedicated to accelerating the pooling layer of convolutional neural networks. With the help of dedicated units for each layer TPU achieves much better energy efficiency and performance than the conventional chips used for deep learning. TPUs have their own instruction set which follows the CISC (Complex instruction set) architecture.

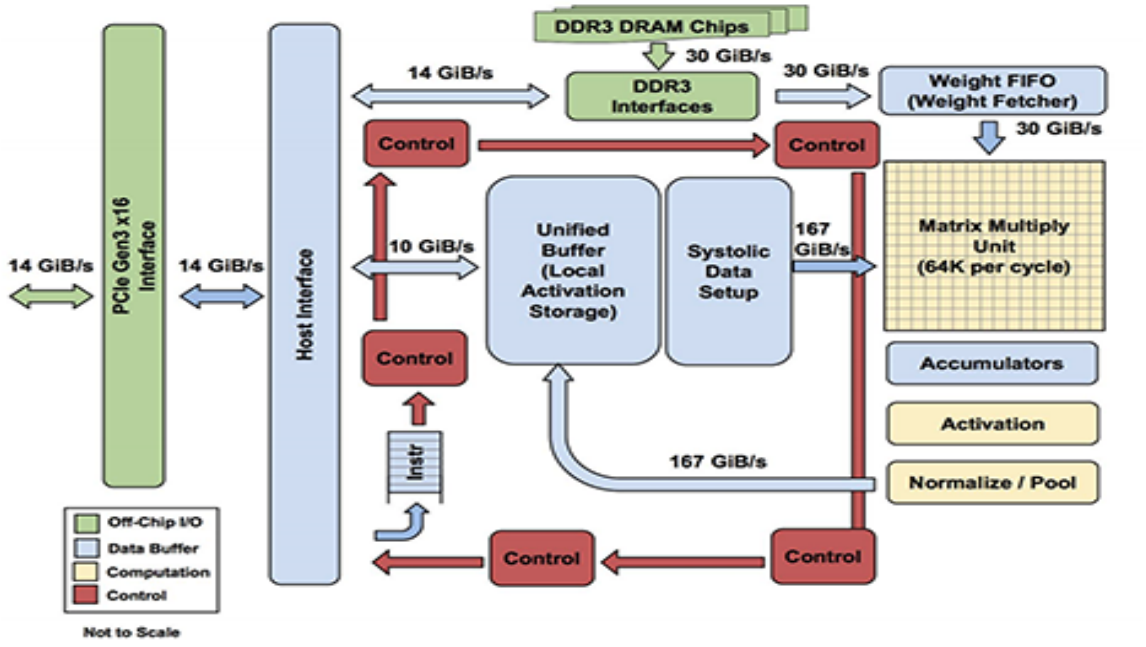


Figure 2.1: TPU Architecure in-depth

Source-https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

Even though ASIC's have a better performance and energy efficiency than GPUs for deep learning, designing an ASIC is a very expensive process and not feasible at the academia level. The chip would become redundant if there are models implementing convolutional neural network in a slightly different way with new layers. Nvidia's Volta architecture shown in Fig 1.2 combines the benefits of an ASIC and general purpose GPUs. As convolution layer is the most compute intensive layer it has a special unit called Tensor cores which accelerates the convolutional layer. Along with the tensor cores it also has the single precision and double precision cores which can be used for general computing. So, Nvidia's GPU Volta architecture and Google's TPU highlights some of the architectural advancements done for deep learning.

2.2 GEMM Libraries

Convolutional layer in a convolutional neural network is used for extracting the low-level features of an image. Each filter used in the convolution layer is responsible for extracting a certain feature from the image. Convolutional layer is the most compute intensive layer amongst all the other layers because multiple filters convolve across the image to extract features. Direct convolutions are not used in convolutional layer because of the irregular access patterns, instead a convolution operation between a kernel and image, Im2col function converts both of them to a form that matrix multiplication between the image and kernel gives us the convolution output. This conversion certainly causes data redundancy, but the performance benefit achieved due to the regular access patterns of a matrix multiplication operation outweighs the memory wastage. Another reason for converting a convolution operation to matrix multiplication operation is that GPUs, in general are optimized for matrix multiplication.

As General Matrix Multiplication is one of the most important part and time expensive part of deep learning we investigate the available GEMM APIs for CPUs and GPUs. General Matrix Multiplication(GEMM) APIs are generally a part of Basic Linear Algebra Subprograms (BLAS) library [10]. For CPUs we have ACML, C++ AMP BLAS, ATLAS libraries which have an optimized GEMM functions. Nvidia's cuBLAS library is the most optimized Cuda library for GPUs but is closed source. CLblast library is the only publicly available open source cuda library which has an optimized GEMM. We develop our own optimized GEMM APIs because neither cuBLAS or CLblast are compatible with GPU simulator.

CHAPTER 3: BACKGROUND

This section talks about the rise of heterogeneous computing, GPU architecture and the Cuda programing model.

3.1 GPU as general-purpose processor

Invention of Graphic processing unit brought a renaissance in the field of computer vision and graphics. They were used as a co-processor alongside CPUs for handling the graphics workload over the larger part of the previous decade. Early GPUs had a fix rendering pipeline in which each step was designated to a particular task like rasterization or vector transformation. During the end of previous decade, GPUs started becoming more capable and several programmable stages were added into their pipeline which allowed simple programs to be directly executed on GPUs. Initially, different types of shaders existed for the individual steps of the graphics pipeline. With the addition of more and more shader instructions, running general-purpose algorithms on GPUs became feasible[12].

Current GPUs consists of thousands of simple processing cores grouped into streaming multiprocessors rather than fixed rendering pipeline or different shader types. Millions of threads are launched while calling a GPU function also called as a GPU kernel which runs on different cores in parallel. With the help of thousands of simple processor cores, GPUs achieve enormous level of parallelism which increase performance by orders of magnitude compared to CPUs for well-suited workloads. The Fig shown in 3.1 shows the basic arrangement and components of a contemporary embedded GPU.

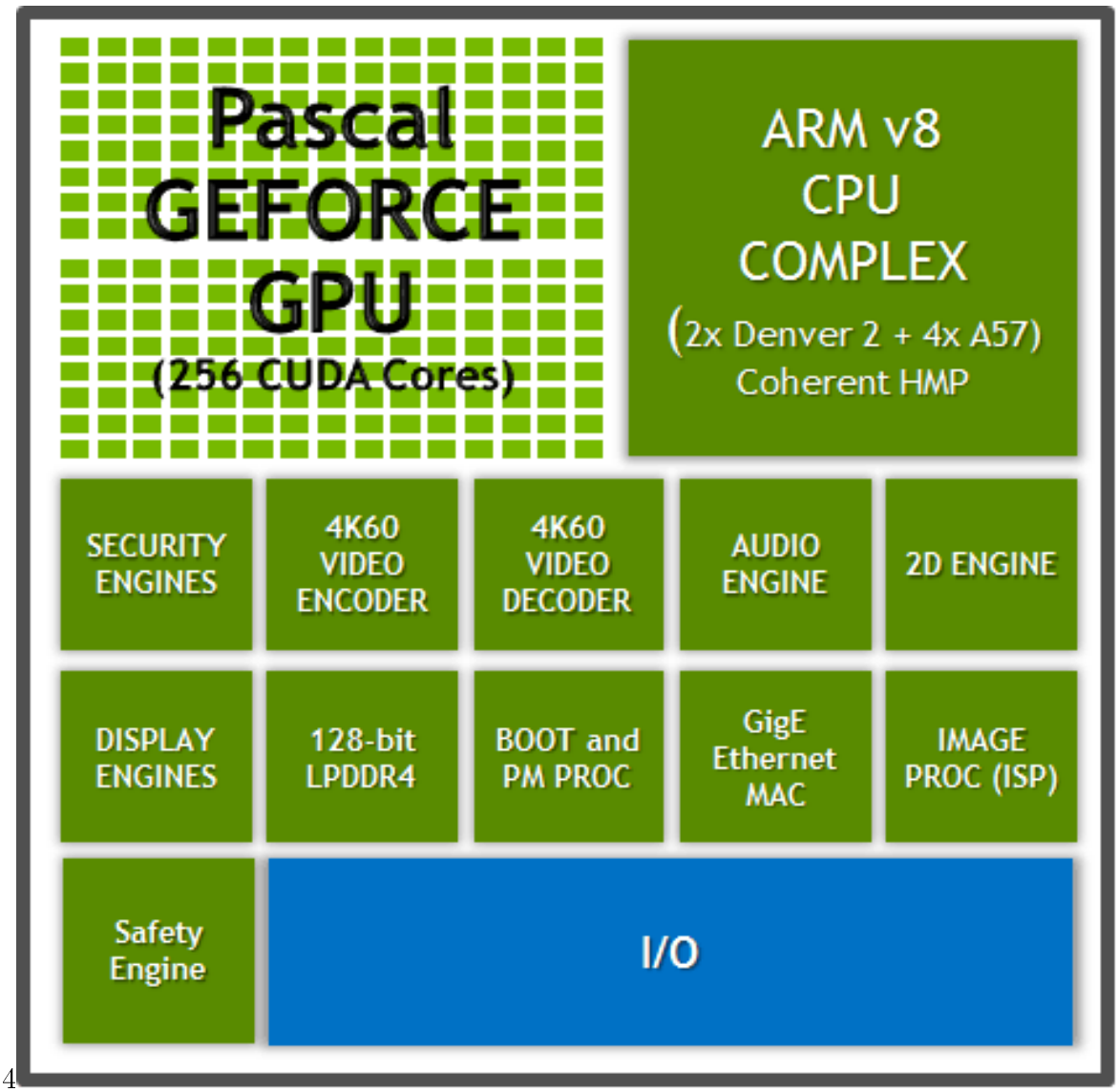


Figure 3.1: A simple GPU Block Diagram

3.1.1 Nvidia GPU Architecture

Nvidia is one of the biggest GPU providers in the world and its CUDA-capable GPUs are of great performance in the field of GPU computing. The Fig 3.2 shows the general GPU architecture. Generally, a Nvidia GPU consists of Graphic processing clusters, Memory controllers and Streaming multiprocessors. The Streaming multiprocessors contain CUDA cores in it and are connected to the L2 Cache and main memory through an interconnect.

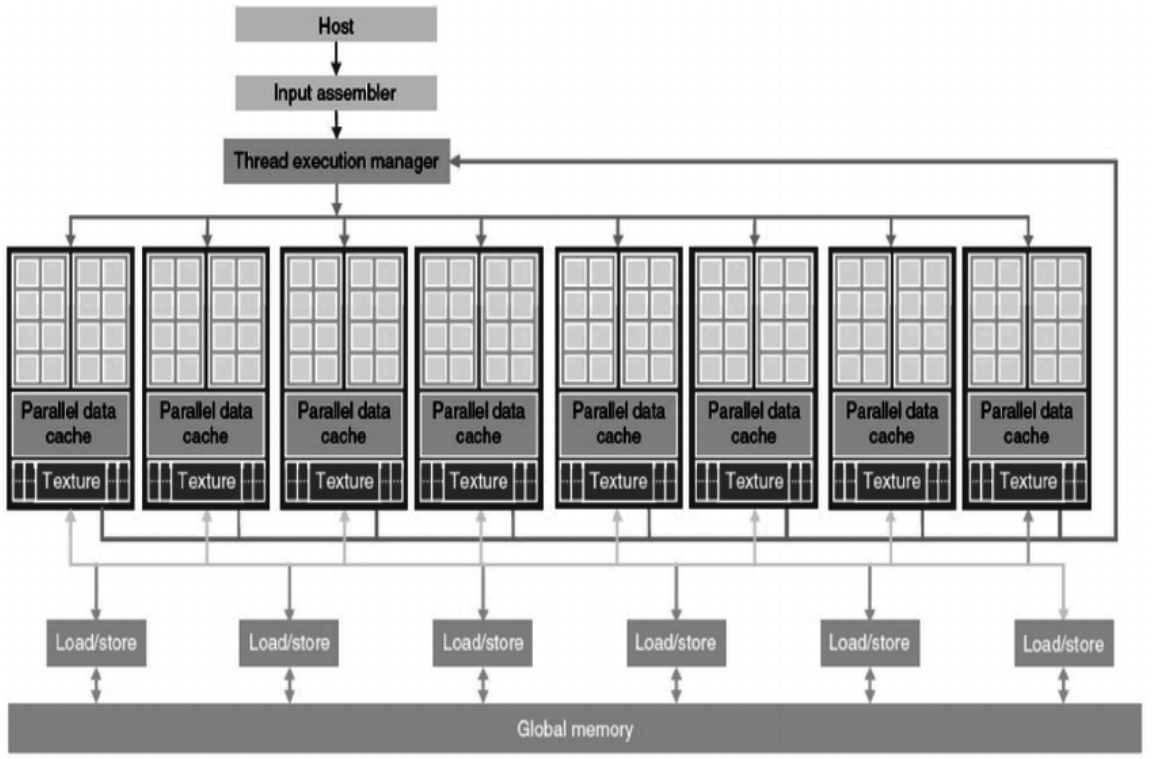


Figure 3.2: GPU Architecture

Source-https://www.researchgate.net/figure/Architecture-of-a-CUDA-capable-GPU-based-on-the-Tesla-architecture-20-22-fig2-51041816

3.1.2 Nvidia Streaming Multiprocessor

Nvidia Streaming Multiprocessor shown in Fig 4.8 is the most fundamental component of a Nvidia GPU architecture. A Nvidia SM consists of single/double precision CUDA cores which are used for executing arithmetic instructions, texture engine to modify a bitmap image be placed onto an arbitrary plane of a given 3D model as a texture and polymorph engine. The section 3.1.6 will help us understand how millions of threads are grouped into thread blocks and scheduled across the SMs. Inside an SM we have warp schedulers and dispatch units which picks up warps from active pool and send it to cores for execution. Each SM has its local low-latency memory for context switching and faster computations.

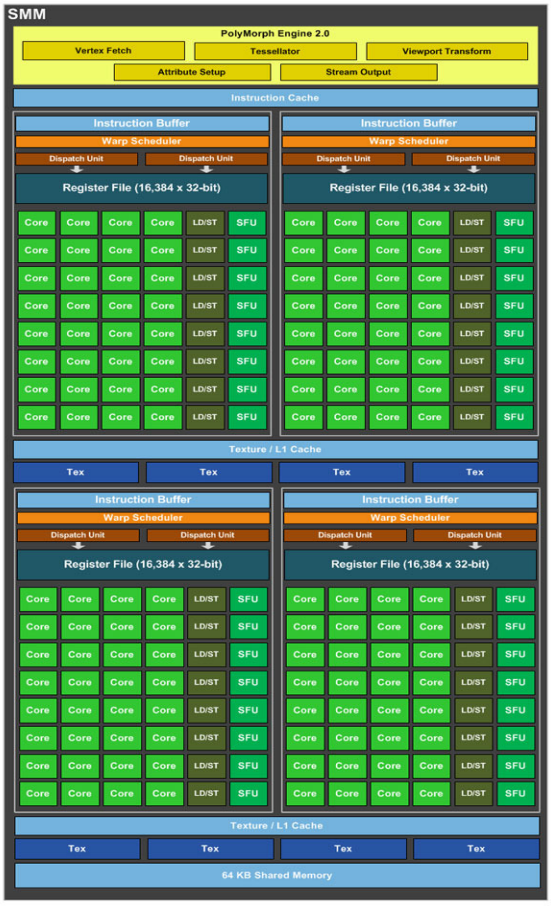


Figure 3.3: Maxwell Streaming Multiprocessor Architecture

Source-https://devblogs.nvidia.com/maxwell-most-advanced-cuda-gpu-ever-made/

3.1.3 GPU Memory Hierarchy

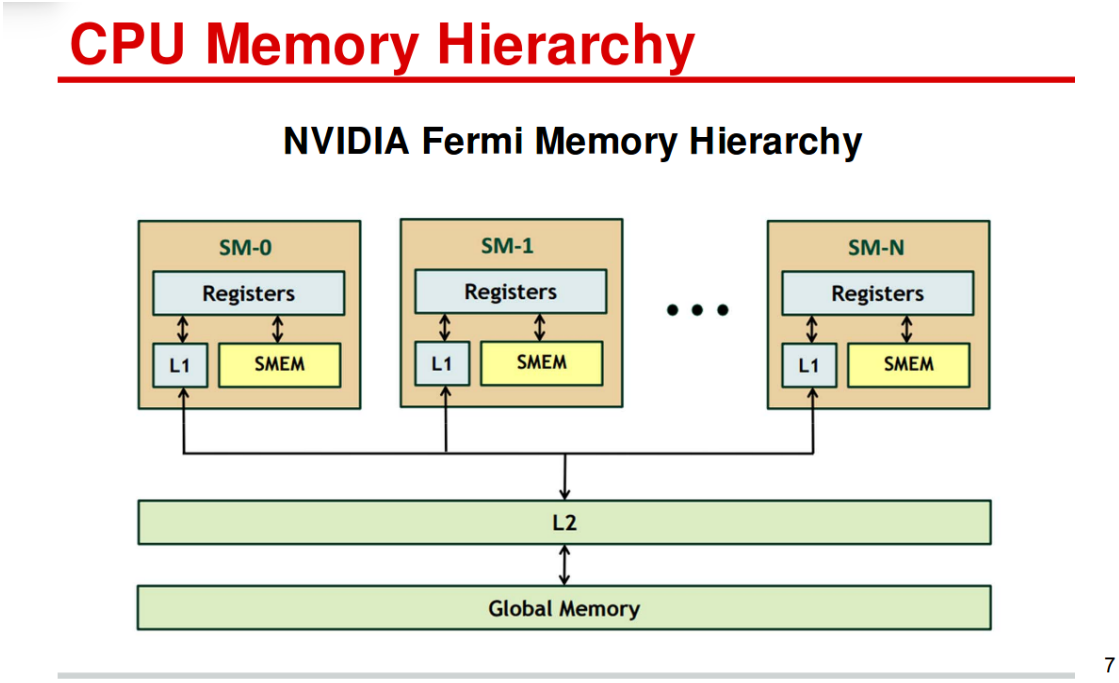


Figure 3.4: GPU Memory Hierarchy

Source-http://www.orangeowlsolutions.com/archives/388

Fig 3.4 shows the memory hierarchy of a Nvidia GPU. Registers are private to each Streaming multiprocessor and are used to store context of a warp. Each warp scheduled on the SM is allocated some register file space. If a warp is stalled for a memory access or cache miss, the stalled warp is replaced by a different warp while the swapped-out warp is waiting for its data. The warp switching occurs quickly because their context is stored in the register files. Thus register files help in **hiding** memory access latency.

L1 cache and Shared memory are both on chip low latency memories. But not all architecture has both the memories, some Nvidia GPU architecture give programmer a choice between using L1 cache and Shared memory for their program. The main difference between L1 cache and Shared memory is that unlike L1 cache Shared

memory can be explicitly programmed for accelerating Cuda code. Shared memory shares data and computation results by all the threads in the same block, called the cooperative thread array (CTA) or block of threads in the CUDA. So it is recommended not to allocate a large amount of shared memory per thread block as it affects GPU occupancy. This allows massive parallel-reduction processes for computing and synchronizing the block results that can be accessed using each CTA index or block index (for example, bid=blockIdx.x). This is the second (middle) layer of the parallel granularity in CUDA

All the SM in a GPU also share a common L2 cache and Global memory, the latency associated with both these memories is relatively high. So, as a CUDA programmer you must ensure not to have a lot of global memory accesses in the code. As threads within a thread block can communicate with each other using Shared memory, all the threads launched for a kernel can communicate using Global memory. This memory allows the grids communication and the synchronization of the results among all the CUDA blocks and to share a large data set and global computation results in global memory.

3.1.4 CUDA and OpenCL

With GPUs becoming suitable to run arbitrary code for GPGPU applications, the need for easy access to the GPUs vast computing power arose. This section introduces the two most popular frameworks for GPGPU scenarios: NVIDIA's CUDA [6] and the vendor-independent Open Computing Language (OpenCL)[13].

CUDA which stands for Compute Unified Device Architecture, is a parallel computing platform specifically for Nvidia GPUs. CUDA framework allows a programmer to use an Nvidia GPU for general purpose computing. CUDA platform extends the industry-standard C language to create a new parallel programming language called CUDA C which is compiled using a nvcc compiler. CUDA Fortran is also available for Fortran users. The CUDA [6] framework includes two separate APIs to access

the GPU [3],the CUDA Driver API and the CUDA Runtime API. The CUDA Driver API implements basic primitives to access the GPU that allow a high degree of direct control over the device.

3.1.5 Cuda Program Structure

A CUDA program can usually be divided into two parts, host code and device code. Host code, as the name suggests, is the code that runs on the host, which is usually a traditional CPU. It is the serial part of the program which is written in straight ANSI C/C++. Device code, on the other hand, is the parallel part of the program which is written in ANSI C extended with CUDA keywords. A complete Cuda program is mixed source code with both host code and device code. The nvcc compiler will separate them during compilation. The ANSI C/C++ code will be compiled with host's standard C++ compiler and run on the CPU. The GPU code, also known as GPU kernels, will be compiled by the nvcc compiler and mapped to the GPU device.

GPU is generally used as a co-processor; host code is responsible for allocating both host and device (GPU) memory and to transfer the data from host to device. After receiving data from the host device GPU does its computation and the host copies the data back, this is shown in Fig 3.5. The global keyword indicates that the following function will run on the GPU.

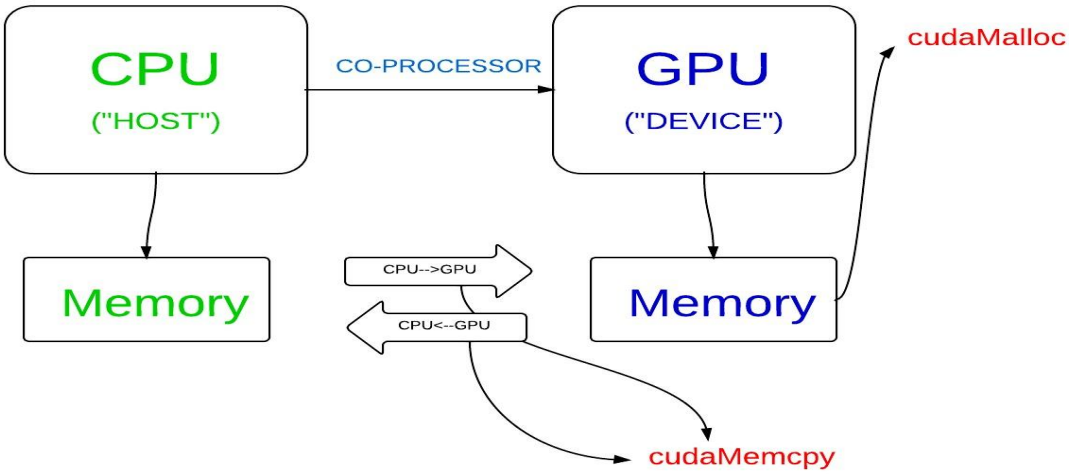


Figure 3.5: Host-Device Interaction

Source-https://blogandcode.wordpress.com/author/neelbommisetty/

3.1.6 Cuda Program Execution

Analogous to a function call in C/C++ we have a kernel call in CUDA C whenever we have to invoke a device function. We need to configure the execution parameters before call the kernel.

Kernel Name«< NUMBER OF BLOCKS, NUMBER OF THREADS PER BLOCK, SHARED MEM PER BLOCK»>(Param1,Param2) is the general prototype of a device function. As GPUs work on a SIMT model, programmer is responsible for configuring number of threads for a kernel. In CUDA, the kernel is executed with the aid of threads. The thread is an abstract entity that represents the execution of the kernel. We will talk about how threads are grouped into thread blocks and warps later in this section. Fig 3.6 shows how a single CPU thread spawns multiple GPU threads.

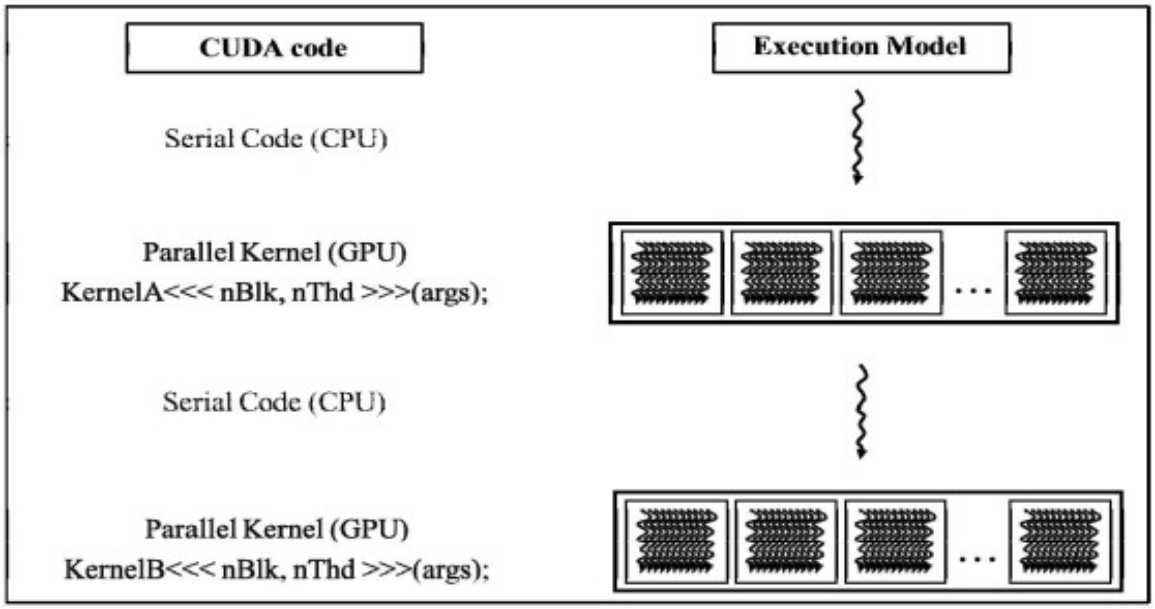


Figure 3.6: Execution of a CUDA Program

Source-https://www.researchgate.net/figure/Programming-and-execution-model-of-CUDA-fig5-273478745

3.1.7 Cooperative Thread Arrays (Thread Blocks)

With millions of threads being launched with each GPU kernel call, thread block is an abstraction used for grouping threads such that the process of scheduling thread across SM and data mapping becomes easy. Basically, thread block is a group of threads that can execute either serially or in parallel. There is a limit on the amount of threads that can be grouped into a thread block with 1024 being the number for most Nvidia GPU architectures. As shared memory is allocated per thread block it becomes a very useful abstraction while synchronizing between different thread block of the same kernel. All the threads inside a SM are guaranteed to run on a single SM, the thread block is swapped out when all its threads are done with their execution.

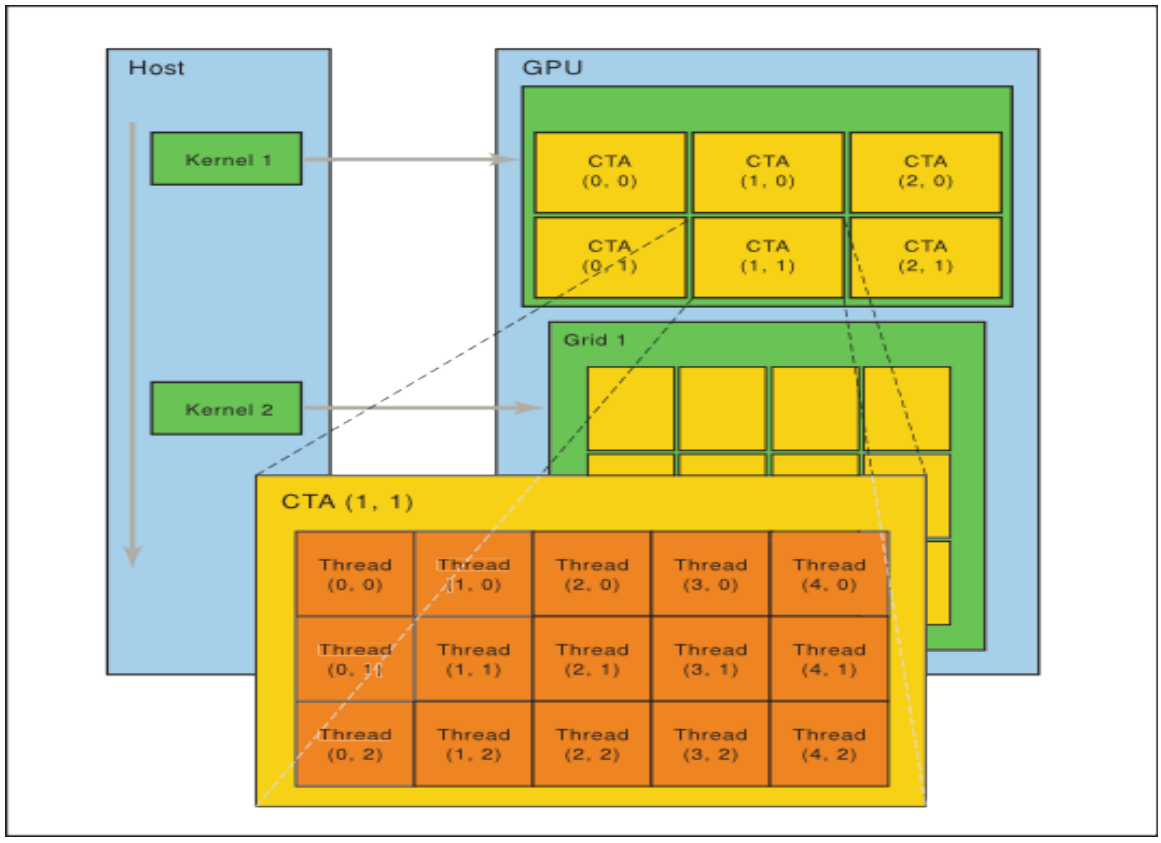


Figure 3.7: A cooperative thread array (CTA) is a set of concurrent threads that execute the same kernel program. A grid is a set of CTAs that execute independently.

Source-https://www.researchgate.net/figure/The-CUDA-execution-model-and-thread-addressing-scheme-fig9-237570681

Each thread has it's unique Id inside the thread block, that Id is used to determine the assigned roles or fetch data from a specific positions or write the data back to specific position. The thread identifier is a three-element vector tid, (with elements tid.x, tid.y, and tid.z) that specifies the thread's position within a 1D, 2D, or 3D thread block as shown in Fig 3.7. Threads within a thread block execute in SIMT (Single-Instruction Multiple Thread) fashion in groups called warps. A warp is a group of 32 threads which execute in a lock-step mode. In every cycle the instructions scheduler selects the warps from active pool and dispatch unit dispatches it for execution.

3.1.8 GPGPU-Sim A Cycle-Level GPU Performance Simulator

GPGPU-Sim is a cycle-level GPU performance simulator that focuses on "GPU computing" (general purpose computation on GPUs). GPGPU-Sim runs program binaries that are composed of a CPU portion and a GPU portion. However, the microarchitecture (timing) model in GPGPU-Sim 3.x reports the cycles where the GPU is busy it does not model either CPU timing or PCI Express timing (i.e. memory transfer time between CPU and GPU).

Top-Level Organization

GPGPUsim consists of SIMT cores which Nvidia calls as Streaming Multiprocessor which are connected to the memory partition with the help of interconnection network. Fig 3.8 shows the top level organization of the GPU architecture as modeled by GPGPU sim.

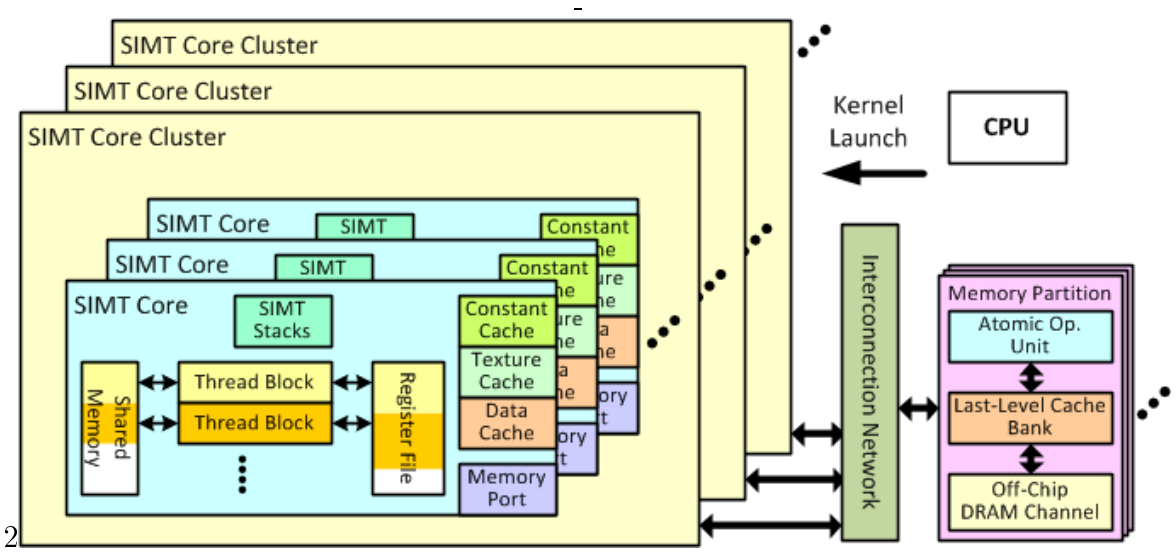


Figure 3.8: Overall GPU Architecture Modeled by GPGPU-Sim

Source-http://gpgpu-sim.org/manual/index.php/Main-Page

Each SIMT Core has its own shared memory,register files,constant cache, data cache and texture cache.

Clock Domains

GPGPU-Sim supports four independent clock domains: (1) the SIMT Core Cluster clock domain (2) the interconnection network clock domain (3) the L2 cache clock domain, which applies to all logic in the memory partition unit except DRAM, and (4) the DRAM clock domain.3.8

SIMT core clusters

Multiple SIMT cores are stacked together in SIMT core clusters. All the SIMT cores inside SIMT Core Cluster share the same connection port to interconnection network as shown in Fig 3.9.

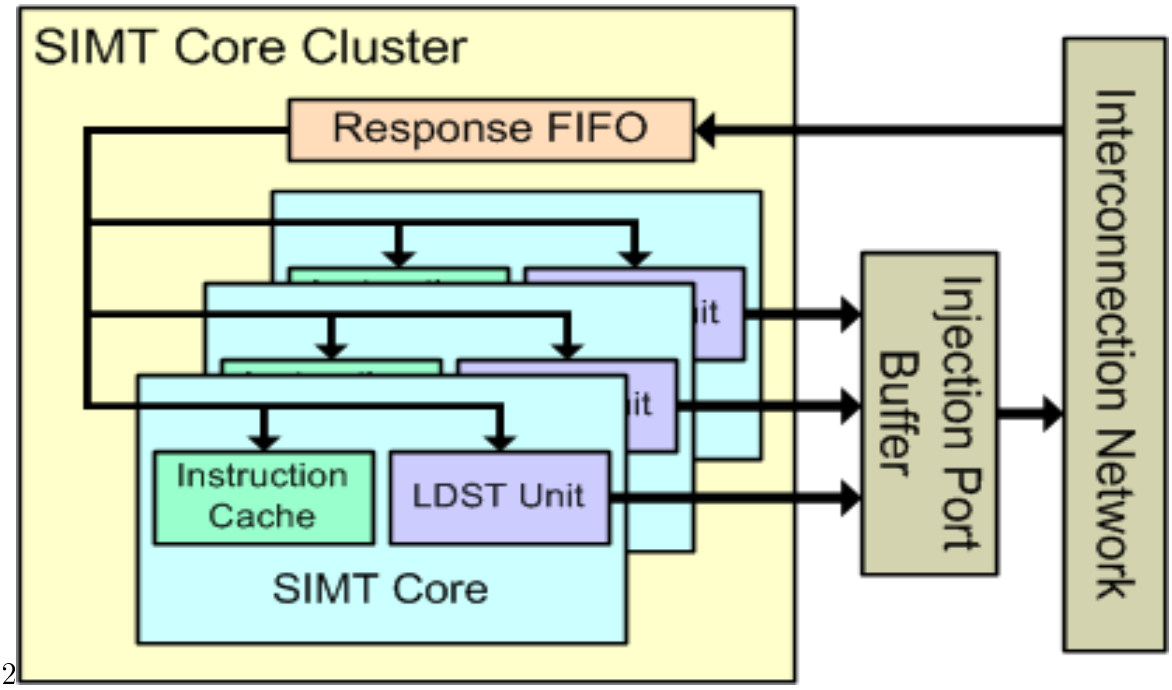


Figure 3.9: SIMT Core Clusters

Source-http://gpgpu-sim.org/manual/index.php/Main-Page

CHAPTER 4: OPEN SOURCE GEMM KERNELS FOR ARCHITECTURE EXPLORATION

The motivation behind writing a custom GEMM kernel stems from the fact that the Nvidia cuBLAS GEMM kernel [15] is not compatible with the GPU simulator as Nvidia does not expose the source code. Our aim is not to match the performance of Nvidia cuBLAS GEMM but to provide an accurate functionality for all the GEMM configuration with a decent speed. This section goes over 3 different kernels which were developed to replace Nvidia's cuBLAS GEMM in the darknet framework [16]. The GEMM kernels are optimized using standard CUDA performance optimization techniques. Some of the optimizations are inspired by Cedric Nutregen's work on accelerating Matrix multiplication for OpenCL devices.

4.1 Base Line Model

General Matrix Multiply (GEMM)

Matrix to matrix multiplication (GEMM) is one of the most important operation in many engineering, and machine learning applications. Before developing our own kernels for GEMM, we will take a look at the functionality of Nvidia's cuBLAS GEMM kernel and also its parameters.

Nvidia cuBLAS GEMM

This is the function prototype of cuBLAS GEMM API

```
cublasStatus-t cublasSgemm(cublasHandle-t handle,
cublasOperation-t transa,\newline cublasOperation-t transb,
int m, int n, int k,
const float *alpha,
const float *A, int lda,
const float *B, int ldb,
const float *beta,
float *C, int ldc)\newline
```

This function performs the matrix-matrix multiplication

C = ALPHA. op (A). op (B) + BETA C ALPHA and BETA are scalar quantities and matrices A, B and C are stored in Column Major format.

1. handle - It is the handle to cuBLAS library context.
2. transa - It specifies whether operation op(A) is non- or (conj.) transpose.
3. transb - It specifies whether operation op(B) is non- or (conj.) transpose.
4. m - It specifies the number of rows of matrix op(A) and C.
5. k - It specifies the number of columns of op(A) and rows of op(B).
6. n - It specifies the number of columns of matrix op(B) and C.
7. alpha - It specifies the <type> scalar used for multiplication.
8. A - It specifies the <type> array of dimensions lda x k with lda>=max(1,m) if transa == CUBLAS-OP-N and lda x m with lda>=max(1,k) otherwise.

9. lda - It specifies the leading dimension of two-dimensional array used to store the matrix A.

10. B - It specifies the <type> array of dimension ldb x n with ldb>=max(1,k) if transb == CUBLAS-OP-N and ldb x k with ldb>=max(1,n) otherwise.

11. ldb - It specifies the leading dimension of two-dimensional array used to store matrix B.

12. beta - It specifies the <type> scalar used for multiplication. If beta==0, C does not have to be a valid input.

13. C - It specifies the<type> array of dimensions ldc x n with ldc>=max(1,m).

14. ldc - It specifies the leading dimension of a two-dimensional array used to store the matrix C.

C style Matrix-Multiplication

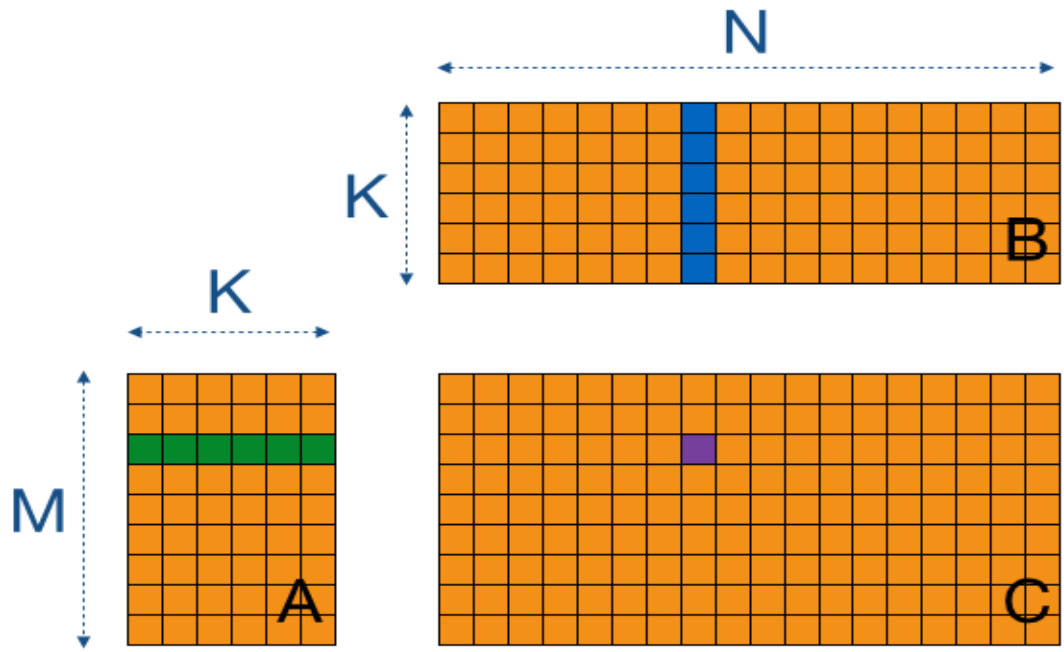
The matrix multiplication of two matrices is computed as following:

```
C := alpha * A * B + beta *C
```

Matrix A is of size M*K while matrix B is of size K*N hence the size of matrix C is M*N as the number of columns of matrix A should be equal to the number of rows in matrix B. For simplicity we will assume the scalar quantities alpha and beta to be 1 which gives us a simple equation:
C+=A*B.

This computation is illustrated in the Fig 4.1: to compute a single element of C (in purple), we need a row of A (in green) and a column of B (in blue).

-



2

Figure 4.1: Matrix Multiplication

Source-https://cnugteren.github.io/tutorial/pages/page1.html

```
int row, col, k;
for (row = 0; i < M; row++)
{
for (col = 0; col < N; col++)
{
c[i][j] = 0;
for (k = 0; k < N; k++)
{
c[row][col] += a[row][k]*b[k][col];
}
    }
     }
```

This version of SGEMM is implemented in plain C using 3 nested loops.

4.2 Optimization 1: Baseline Model

Our first implementation of Matrix Multiplication on GPUs is pretty simple, but it gives us a good amount of speedup over the CPU GEMM code which employs 3 nested for loops. In this approach we launch, M*N threads for our kernel one thread for each of our output pixel. Each thread is responsible for calculating one pixel of the output matrix by looping over matrix A and matrix B, K times.

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;


if( col < N && row < M)
{
  for(int i = 0; i < K; i++)
 {
 sum += (ALPHA *A_gpu[row * K + i]) * B_gpu[i * N + col];
 }
 C_gpu[row * N + col]+= sum;
}
```

Fig 4.2 shows the graph of time taken for execution by this kernel in ms for different matrix sizes . We ran this Test on Nvidia Jetson TX1 GPU. The matrices are square matrices.

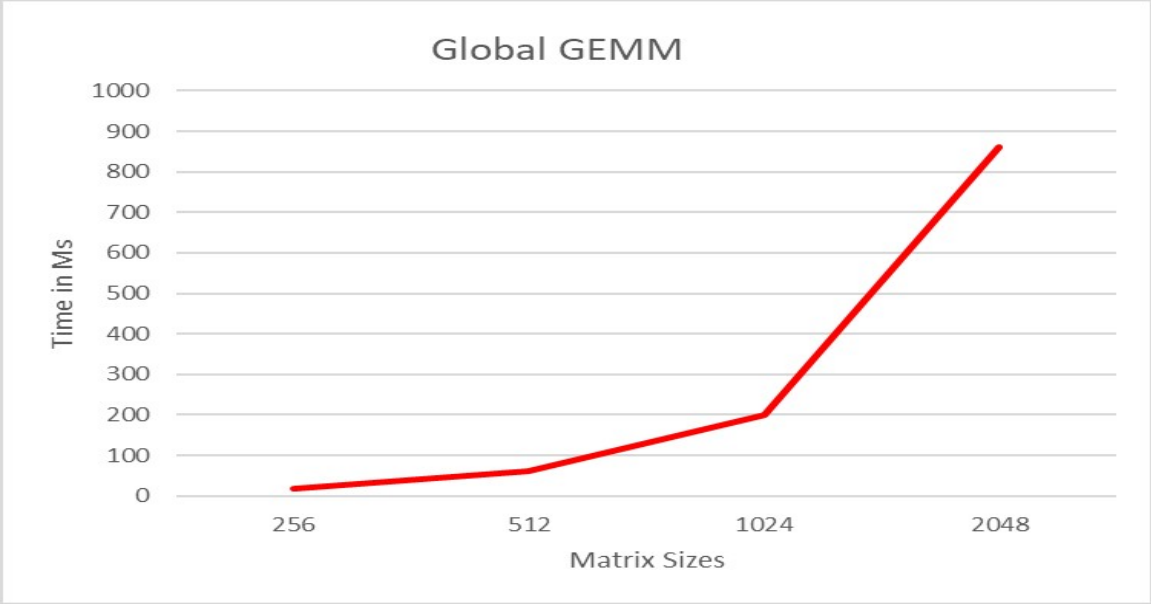


Figure 4.2: Time Taken in ms by Global Memory Kernel over different Matrix sizes

Even though this code gives us a substantial speed up over the CPU Matrix multiplication code, it has a lot of global memory accesses in it. As we studied in the section 3.4 that accesses to global memory are very costly, this results in poor performance. If somehow, we can reduce the global memory accesses we can definitely have some more speed-up.

4.3 Optimization 2: Shared Memory Tiling

Global memory accesses are always very costly, it was the major bottleneck in the previous kernel. We had M*N*K*2 Global memory loads and M*N Global memory stores in the previous kernel. In order to avoid these many Global memory accesses, we use GPUs on-chip, low-latency Shared memory to store the values of matrix A and matrix B. In this kernel every thread block would be responsible for bringing tiles of matrix A and matrix B from global memory to the Shared memory. The matrix multiplication is done inside the Shared memory which leads to a lot of data reuse within a tile.

As shown in Fig 4.3 to compute the purple tile in matrix C we bring in tiles of data from Matrix A(green sub tile) and also from matrix B(blue sub tile). Now we can iteratively update the values in C sub tiles by adding the results of matrix multiplication of A sub tile and B sub tile.

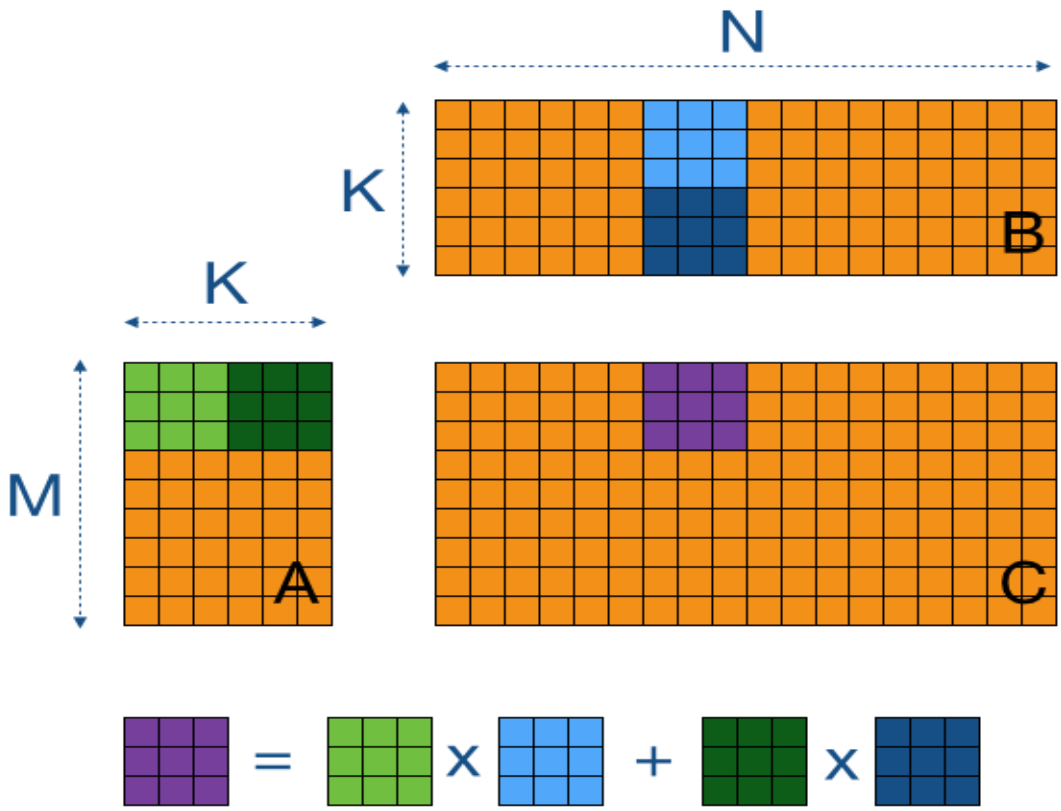


Figure 4.3: GEMM using Shared Memory Tiling

Source-https://cnugteren.github.io/tutorial/pages/page4.html

Well, if we take a closer look at the computation of a single element (in the image below), we see that there is lots of data re-use within a tile. For example, in the 3x3 tiles of the image below, all elements on the same row of the purple tile (Csub) are computed using the same data of the green tiles (Asub).

Fig 4.4 highlights the data re-use within a tile.

Figure 4.4: Data reuse in GEMM using Shared Memory Tiling

Source-https://cnugteren.github.io/tutorial/pages/page4.html

The algorithm for implementing Matrix multiplication using shared memory tiling is described below. For implementing Shared memory tiling, we used a 32X32 sized shared memory array per thread block.

```
\label{sm_algo}
  // Local memory to fit a tile of TS*TS elements of A and B
  #define  TS 32
 __shared__ float sA[32][32];   // Tile size of 32x32
 __shared__ float sB[32][32];
  int Row = blockDim.y*blockIdx.y + threadIdx.y;
  int Col = blockDim.x*blockIdx.x + threadIdx.x;

  //Fetch the subtiles A and B in shared memory
  for (int k=0; k<(k/TS); k++) {
   if ( (Row < arows) && (threadIdx.x + (k*32)) < acols)
  {
   sA[threadIdx.y][threadIdx.x] = A[(Row*acols) + threadIdx.x + (k*32)];
  }
   if ( Col < bcols && (threadIdx.y + k*32) < brows)
  {
   sB[threadIdx.y][threadIdx.x] = B[(threadIdx.y + k*32)*bcols + Col];
  }
  //Multiplication within Shared Memory
  for (int j = 0; j < 32; j++)
  {
  sum += sA[threadIdx.y][j] * sB[j][threadIdx.x];
  }
```

The algorithm is divided into two parts

1. Load the tiles from matrix A and matrix B.

2. Synchronize

3. Sub tile multiplication in Shared memory.

As each thread now performs only two global loads, we reduce the access to Global memory by a factor of 32 which along with the data reuse in Shared memory is responsible for the speedup over the baseline model . Fig 4.5 shows the graph of time taken for execution by this kernel in ms for different matrix sizes . We ran this test on Nvidia Jetson TX1 GPU. The matrices are square matrices.

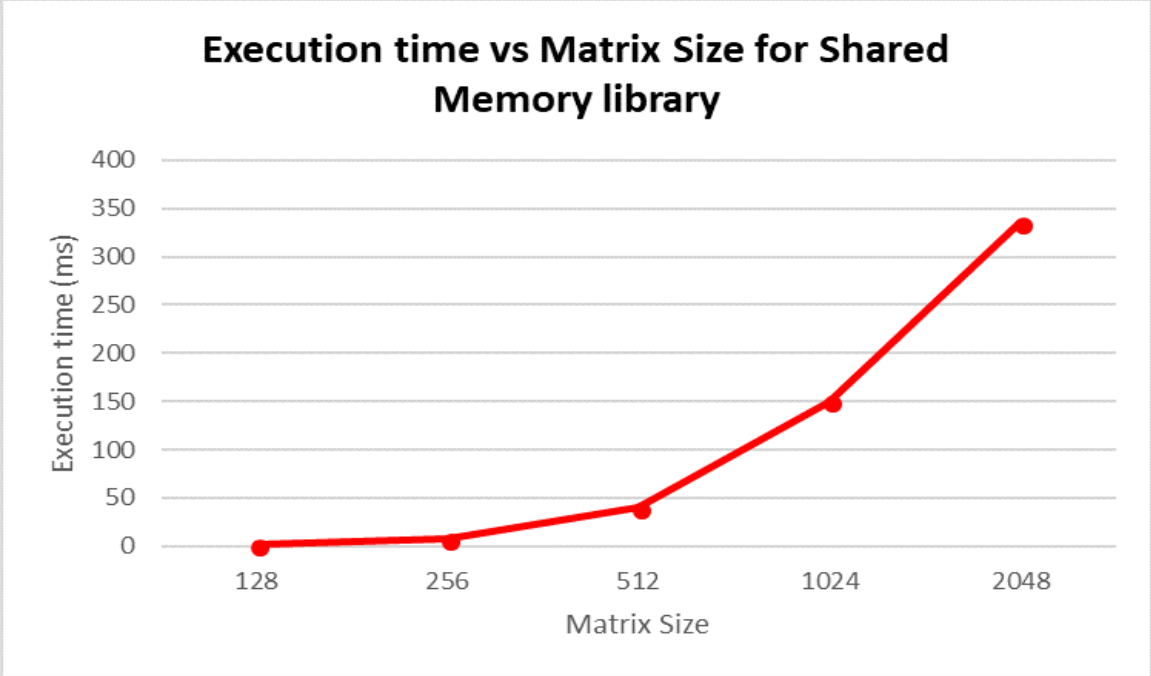


Figure 4.5: Time Taken in ms by Shared Memory Kernel over different Matrix sizes

4.4 Optimization 3: More Work per Thread

The previous approach had some significant speed-up over the C code and base-line implementation as we reduced the number of Global memory accesses. In this kernel along with reducing accesses to Global memory we will also be reducing the accesses to Shared memory by increasing the amount of work done per thread.

```
 for (int j = 0; j < 32; j++)
{
     sum += sA[threadIdx.y][j] * sB[j][threadIdx.x];
}
```

This code is taken from the previous kernel and it does the multiplication of two sub matrices(tiles) in shared memory. For doing a single multiplication operation we need to have two Shared memory load operations and one Shared memory store. Only one out of the four instructions are a useful fused multiply add instruction. We can reduce the loads from shared memory if we increase the amount of data brought by one thread from Global memory to Shared memory and let one thread compute 8 elements arranged in consecutive columns of matrix C . So, in this scenario we are not reducing Global memory accesses but in turn reducing the shared memory loads by a factor of WPT which is variable. WPT should be chose carefully, we do not want the WPT value to be high as it would significantly reduce the amount of threads launched violating the basic principle of GPUs parallelism. Low WPT values would not give us significant performance benefits.

Fig 4.6 shows Matrix multiplication using Shared memory and increasing the work done per thread.

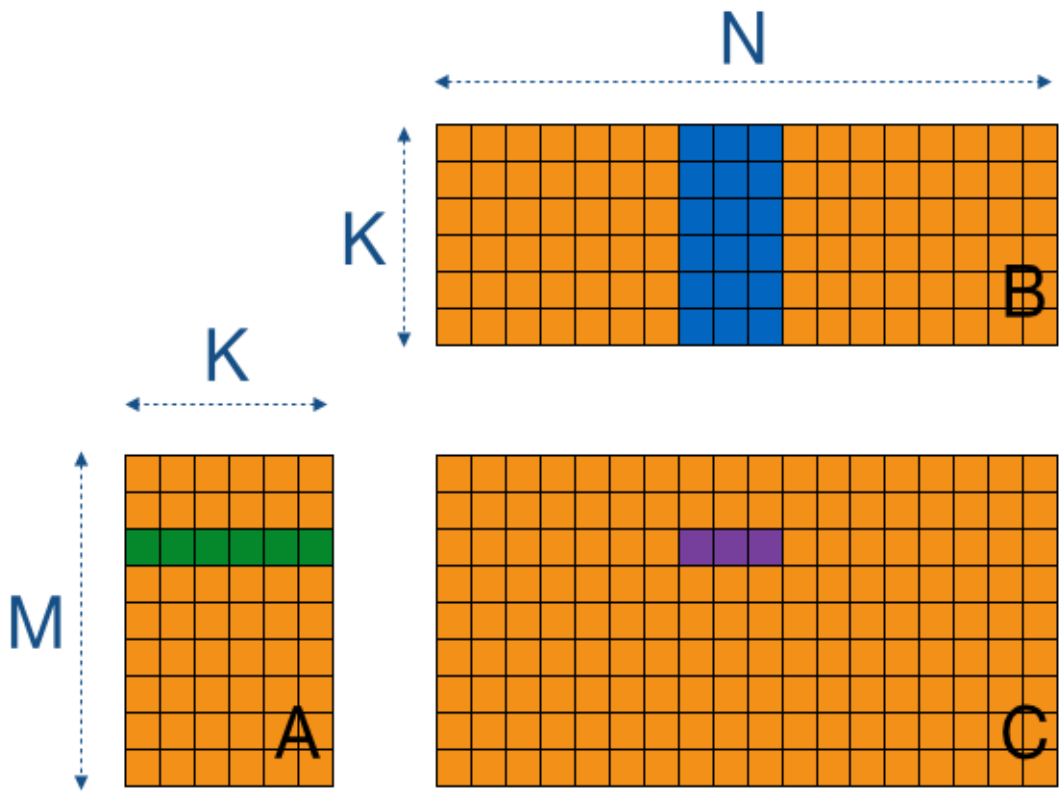


Figure 4.6: GEMM using Shared Memory Register Sub Tiling and increasing the work done per thread

The algorithm for the kernel includes three major steps

1. Each thread should fetch now fetches WPT values of matrix A and B into the shared memory.

2. Synchronize

3. Each thread will do WPT amount of Matrix multiplications inside the shared memory.

4. Each thread is responsible for storing WPT amount of values in the C matrix.

```
#define WPT 8
A factor of WPT registers are initialised to zero for each thread
float acc[WPT];            .
for (int w=0; w<WPT; w++)
{
acc[w] = 0.0f;
}

Each thread now loads WPT values of A and B into the local memory.
for (int t=0; t<t/TS; t++) {
// Load one tile of A and B into local memory
for (int w=0; w<WPT; w++) {

if ( (Row <= (arows)) && (threadIdx.x + (k*TS)) < acols)
{
    sA[threadIdx.y+ w*RTS][threadIdx.x] = A[(Row+w*RTS)*acols
    + threadIdx.x + (k*TS)];
}
   if ( Col < bcols && (threadIdx.y + k*TS) < brows)
  {
  sB[threadIdx.y+ w*RTS][threadIdx.x] = B[((threadIdx.y + k*TS)
  +(w*RTS))*bcols + Col];
  }

}
```

Inner-most loop over WPT doesn't require a new value from Asub each time,

```
saving precious local memory loads
for (int j = 0; j <(TS); j++)
{
  for (int w=0; w<WPT; w++)
  {
  acc[w] +=   (sA[threadIdx.y + w*RTS][j]) * (sB[j][threadIdx.x]);
  }
 }
```

The number of threads spawned in our program should be reduced as a single thread does WPT amounts of loads from shared memory and also WPT amount of multiplications.

Fig 4.7 shows the graph of time taken for execution by this kernel in ms for different matrix sizes . We ran this test on Nvidia Jetson TX1 GPU.The matrices are square matrices.

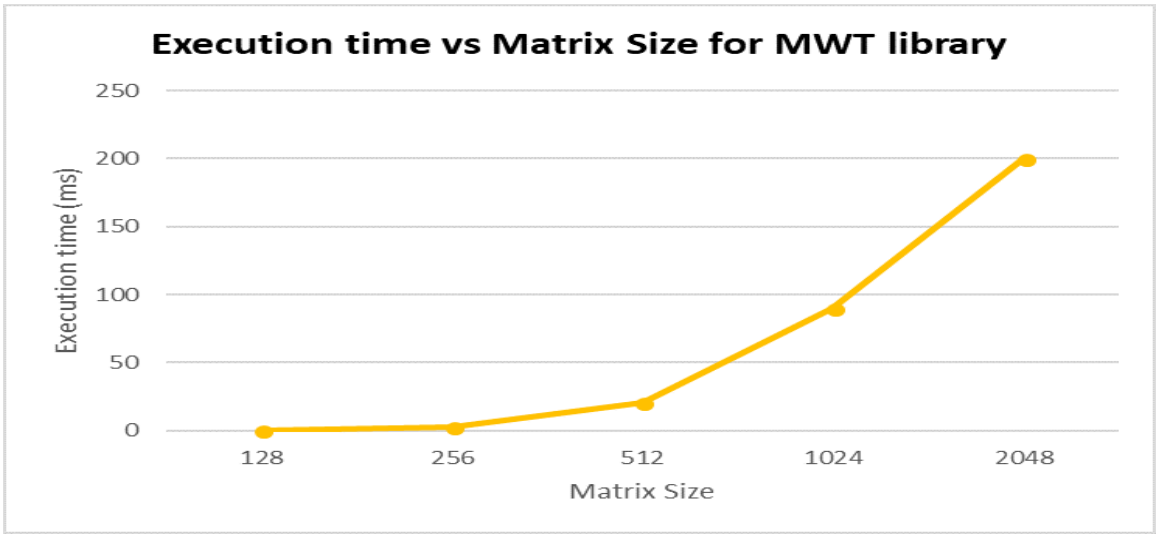


Figure 4.7: Time Taken in ms by More Work Per Thread Kernel over different Matrix sizes

4.5 Architecture Modeling of Embedded GPUs

Nvidia Jetson TX1 (Maxwell architecture) and Jetson Tx2 (Pascal architecture) are Nvidia's latest embedded GPUs specifically designed for low-power embedded applications. Publicly available information on these hardware chips is limited so the first step for architecture modeling of embedded GPUs was to collect the available literature from web, Nvidia's Cudadeviceprop structure is also a good source to collect information about Jetson TX1 and TX2.

4.5.1 GPGPU sim Parameters

Before we move to the Architecture modeling of embedded GPUs on the GPU simulator, let us understand some GPGPU sim parameters which we will be configuring. Section 3.1.8 will help in understanding these parameters.

1. -gpgpu-ptx-sim-mode : To Select Functional Mode or Performance Mode
2. -gpgpu-ptxsave-converted-ptxplus : Allows Converting the Code to Ptxplus instead of PTX
3. -gpgpu-n-clusters : Each Cluster has its own path to Interconnect Network
4. -gpgpu-ncores-per-cluster :Number of SM per cluster.
5. -gpgpu-clock-domains <Core Clock>:<Interconnect Clock>:<L2 Clock>:<DRAM Clock : Clock Domains of various subsytems.
6. -gpgpu-shader-registers : Number of registers per shader core.
7. -gpgpu-shader-core-pipeline: Shader core pipeline configuration
8. - gpgpu-pipeline-widths : Pipeline width of various functional units inside GPU.
9. -gpgpu-num-sp-units : Number of Single Precision Units.
10. -gpgpu-cache:dl1 : Setting L1 Cache size

11. -gpgpu-shmemsize : Setting Shared Memory size
12. -gpgpu-cache:dl2 : Setting L2 Cache size
13. -ptx-opcode-latency :Setting Instruction Latency
14. -gpgpu-operand-collector-num-units-sp : Single precision operand collectors
15. -gpgpu-shmem-num-banks : Number of Shared Memory Banks
16. -gpgpu-num-sched-per-core : Number of schedulers per core.
17. -gpgpu-scheduler : GPU scheduling policy.

4.5.2 Jetson TX1

NVIDIA Tegra X1 is an embedded GPU which includes power efficient Maxwell GPU architecture. In the section 1.1 we compared the performance per watt of Jetson Tx1(embedded GPU) with Titan GPU(server class GPU), Embedded GPUs being low-power devices had a better performance per watt than server class GPUs. Maxwell GPU cores in TX1 also have support for 16-bit floating point calculations for embedded applications.

Jetson TX1 shown in Fig 4.8 contains 2 Streaming Multiprocessors with each having 128 cores. The Streaming multiprocessor is divided into four blocks with each block containing 32 Cuda cores (total 128 cores per SM). Each Streaming multiprocessor has 4 dedicated warp schedulers and 8 dispatch units such that each each warp scheduler is capable of dispatching two instructions per warp every clock.

Figure 4.8: Maxwell SM Architecture

Source-https://devblogs.nvidia.com/maxwell-most-advanced-cuda-gpu-ever-made/

4.5.3 Jetson TX2

NVIDIA Tegra X2 is an embedded GPU which includes high performance, power efficient Pascal GPU architecture. It also contains 2 Streaming Multiprocessors with each having 128 cores. Similar to Pascal GPU architecture, The Streaming multiprocessor is divided into four blocks with each block containing 32 Cuda cores (total 128 cores per SM). Each Streaming multiprocessor has 4 dedicated warp schedulers and 8 dispatch units such that each each warp scheduler is capable of dispatching two instructions per warp every clock.

Nvidia does not release the entire details of hardware specification of any GPU. They do not document the overall instruction execution flow, scheduling policies used for choosing thread blocks and warps and how branch divergence is handled. So, before modeling the GPUs in simulator we will look into some of the hardware specification of the devices provided to us Nvidia.

4.5.4 Hardware Configuration of TX1 and TX2

The table 4.1 compares Hardware specifications of Nvidia Jetson TX1 and Nvidia Jetson Tx2 GPUs while the table 4.2 compares the Architecture modeling differences between Jetson TX1 and TX2.

Table 4.1: Jetson TX1 and TX2 Hardware Configuration

Hardware Specification	Jetson TX1	Jetson TX2
Architecture	Maxwell	Pascal
CPUs	ARM A57 and A53	ARM Cortex-A57 and Denver
Number of SM	2	2
Number of Cuda Cores	256	256
Number of Schedulers/SM	4	4
L1 Cache size	Unknown	Unknown
L2 Cache size	256kb	512kb
Reg file size	64kb	32kb
Max Tb/SM	32	32
Shared Mem Size	64kb	48kb
Instruction Latency	Undefined	Undefined
Scheduling Policy	Undefined	Undefined
No. of Sp collector units	Undefined	Undefined

Table 4.2: Architecture Modeling

Parameters	Jetson TX1	Jetson TX2
-gpgpu-ptx-sim-mode	Performance	Performance
-gpgpu-n-clusters	2	2
-gpgpu-ncores-per-cluster	1	1
-gpgpu-clock-domains	1137.0:2700.0	1481.0:2750.0
-gpgpu-shader-registers	65536	32768
-gpgpu-shader-core-pipeline	2048:32	2048:32
-gpgpu-pipeline-widths	2,1,1,2,1,1,2	4,1,1,4,1,1,6
-gpgpu-num-sp-units	8	4
-gpgpu-cache:dl1	32:128:4	64:128:6
-gpgpu-shmemsize	65536	49152
-gpgpu-cache:dl2	128:128:8	128:128:16,
-gpgpu-operand-collector-num-units-sp	6	20
-gpgpu-shmem-num-banks	32	32
-gpgpu-num-sched-per-core	4	4
-gpgpu-scheduler	gto	gto

CHAPTER 5: RESULTS

This chapter goes over the experimental setup and the various results.

5.1 Experimental Setup

The deep learning framework we use for our experiment is darknet [16]. Darknet [16] is an open source neural network framework written in C and CUDA. The architecture we ran on the top of deep learning framework was Yolo [8] architecture shown in the Fig 5.1. The deep learning application was image detection.

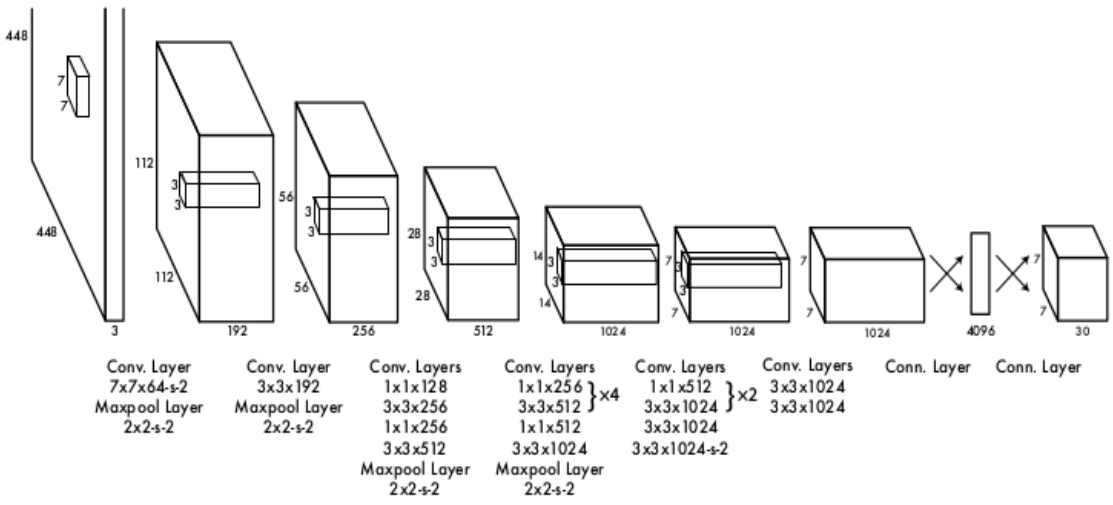


Figure 5.1: Yolo Architecture

Source-http://jderobot.org/Ni9elf-colab

5.2 Performance Comparison

In this section we will start off by comparing the time-efficiency of the GEMM kernels over different square matrix sizes. In the next section we will be integrating our GEMM kernels with darknet framework to record the execution time of each kernel along with their IPC's. We measure the execution time for the 1st layer, first

3 layers and all the layers of Yolo net, the reason behind recording metrics for the 1st layer and first 3 layers is that we have used GPGPU simulator to simulate 1 and first 3 layers of Yolo Net as simulation of more than 3 layers takes extremely large amount of time.

5.2.1 Comparing performances of all the Libraries over Different Sizes

The motivation of this research was to Enable GPU architecture exploration on embedded GPUs for deep learning applications by writing customized GEMM kernels compatible with GPU simulator. Graph 5.2 compares the time taken by the customized GEMM kernels for multiplying two square matrices along with commercially available GEMM APIs which are not compatible with GPU simulator. This experiment was done on Nvidia Jetson TX1 GPU.

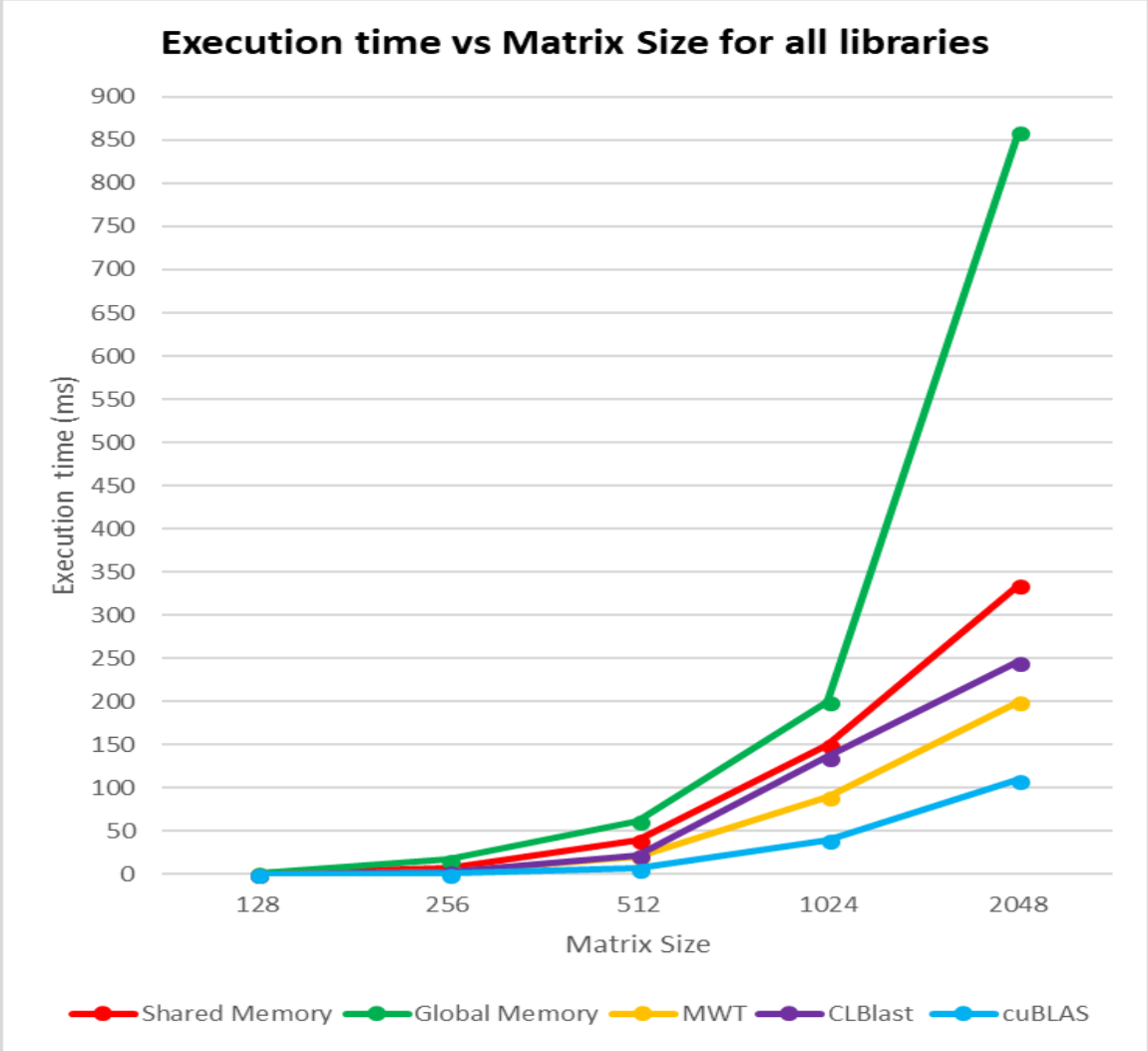


Figure 5.2: Time Taken in ms by GEMM kernels over different matrix sizes

Nvidia's cuBLAS GEMM API is extremely optimized and fine tuned for faster performance, it is optimized at assembly level for all the architectures. As Nvidia's cuBLAS GEMM API is closed source we do not exactly know how the optimizations technique are employed for achieving faster performance. 2D register blocking and architecture specific instructions like warp shuffle instructions might be some of the possible optimizations Nvidia uses. More work done per thread kernel also shows impressive results and is faster than CLblast library for small matrix sizes. Global

Memory kernel is the slowest amongst all.

5.3 Detailed Analysis

5.3.1 Comparing Per Kernel Execution Time

In this subsection with the help of bar chart 5.3 and 5.4 we show the percentage of time required to execute each kernel compared to the total execution time. We conduct our experiments on Yolo net with different GEMM kernels and different amount of layers.

The 3 Yolo net configurations with different GEMM libraries are

1. Yolo net with cuBLAS GEMM kernel
2. Yolo net with Global memory kernel.
3. Yolo net with Shared memory kernel

Yolo net in total has 24 convolutional layers followed by 2 fully connected layers. We ran our tests on

1. Yolo net with only one convolution layer which we call as 1 layer.
2. Yolo net with first 3 convolution layers which we call as 3 layer.
3. All Yolo net layers which we call as entire network.

Bar graph 5.3 shows the percentage of time required to execute each kernel compared to the total execution time. This graph helps us to visualize the percentage of time spent on each kernel for Yolo net with 1,3 and all layers. The graph is divided in to 3 sections based on the GEMM library integared with darknet. The first section is darknet framework integrated with cuBLAS GEMM while the second section is darknet integrated with Global memory GEMM and the third section is darknet with Shared memory GEMM. In each section we have

1. The red line shows the percentage of time consumed by a kernel compared to the percentage of total time spent in execution of all kernels for all the layers of Yolo net.

2. The green line shows the percentage of time consumed by a kernel compared to the percentage of total time spent in execution of all kernels for first 3 layers of Yolo net.

3. The yellow line shows the percentage of time consumed by a kernel compared to the percentage of total time spent in execution of all kernels for 1 layer of Yolo net.

The graph 5.3 shows per kernel execution time percentage for Jetson TX1.

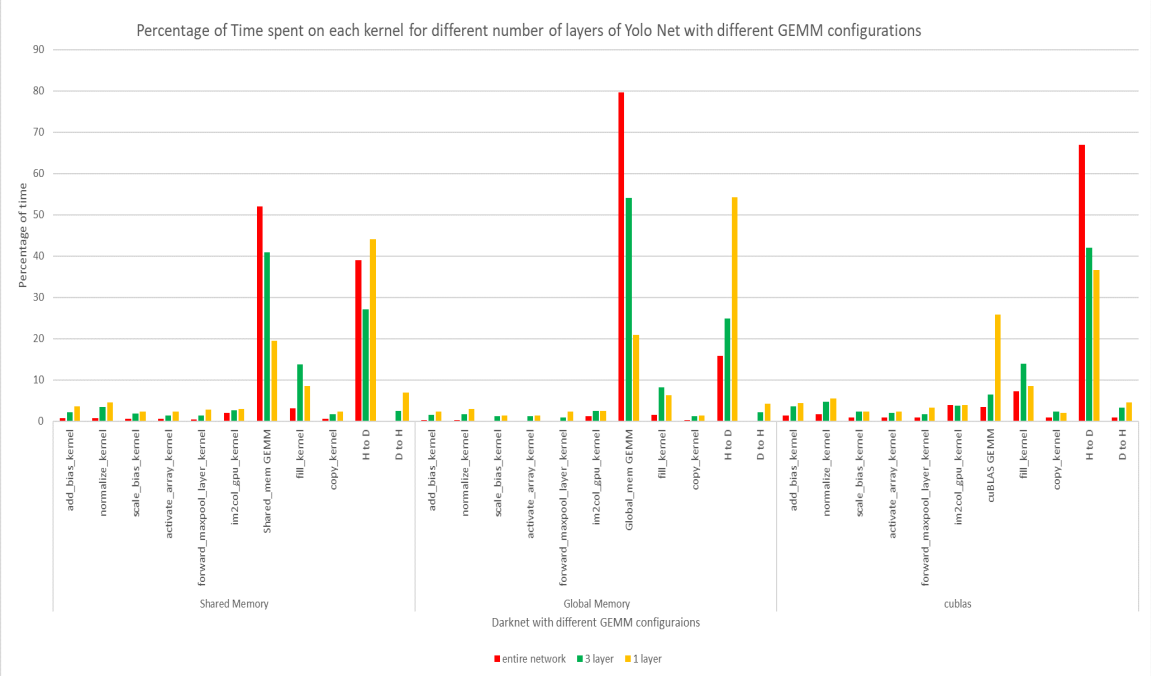


Figure 5.3: Time taken per kernel execution for different layers of Yolo Net on Jetson TX1

The graph 5.4 shows per kernel execution time percentage for Jetson TX2.

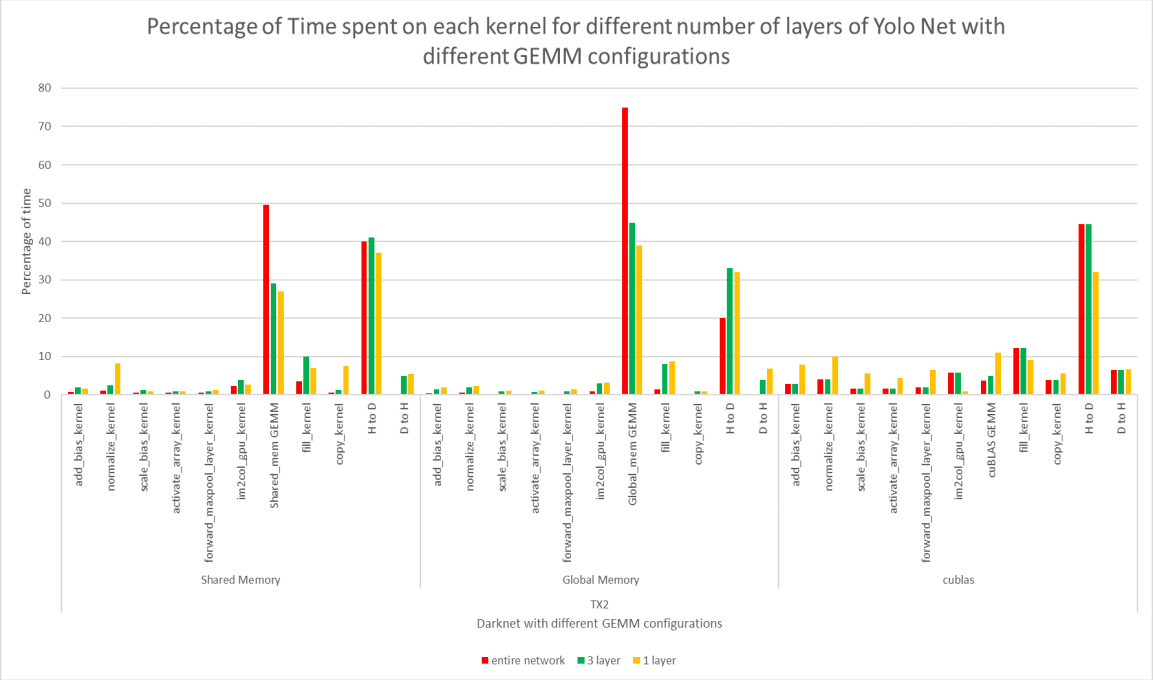


Figure 5.4: Time taken per kernel execution for different layers of Yolo Net on Jetson TX2

Time consumed by the GEMM kernel and the memory copy kernel seems to be extremely high compared to other kernels. Memory copy from host to device is dependent on the PCI-bus or Nvlink bandwidth. GEMM kernel seems to the performance bottleneck for accelerating deep learning on GPUs. So, optimizing Gemm kernels can lead us to better and faster Yolo Net implementation. Let us look into the details of GEMM kernel.

5.3.2 Comparison of IPC and Time Efficiency over GEMM Kernels

From the previous graphs we identified that GEMM is the most time-intensive kernel in our application. For making our GEMM kernel fast we employed certain optimization techniques.In this section we will see how our optimization techniques have had an effect on optimizing GEMM and thus the entire application. We will be comparing the IPC and Time Efficiency for GEMM kernels on TX1 and TX2 Hardware.

The graph 5.5 shows the percentage of time taken by different gemm kernels compared to the total execution time for different layers of Yolo Net on TX1.

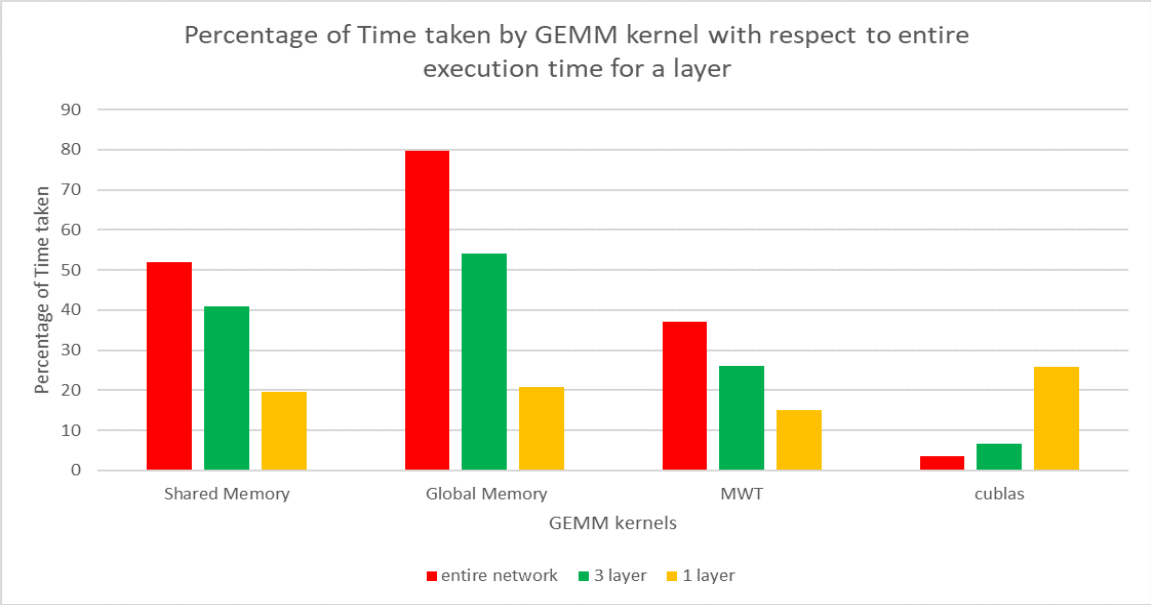


Figure 5.5: Percentage of execution time taken by gemm kernels compared to total execution time for different layers of Yolo Net on TX1

The graph 5.6 shows the IPC of different gemm kernels for different layers of Yolo Net on TX1.

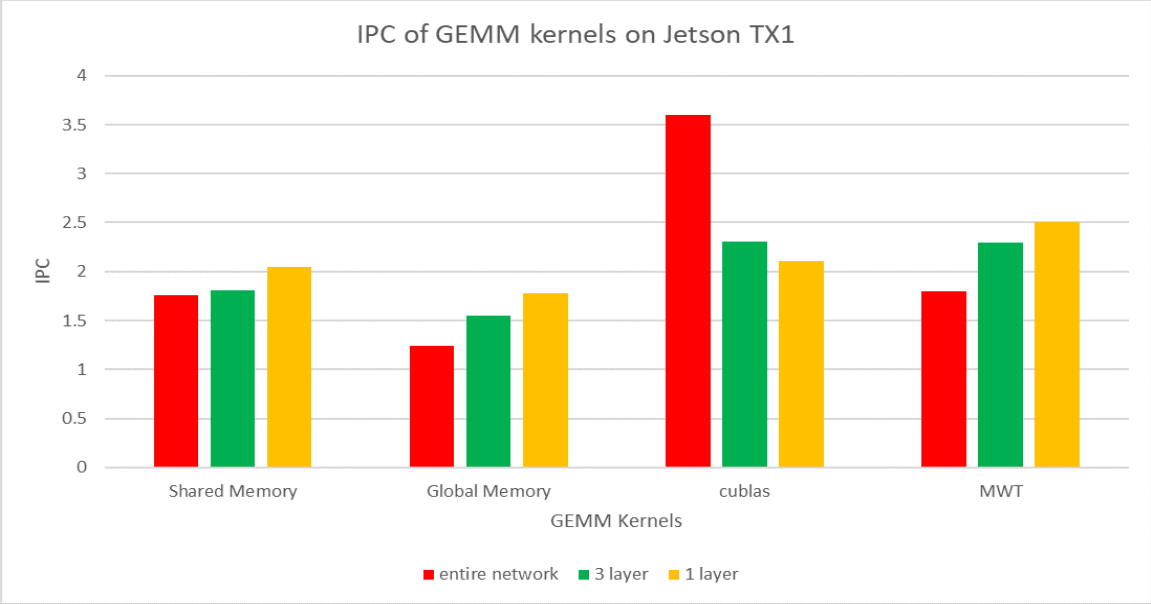


Figure 5.6: IPC of gemm kernels for different layers of Yolo Net on TX1

The graph 5.7 shows the time taken by different gemm kernels for different layers of Yolo Net on TX2.

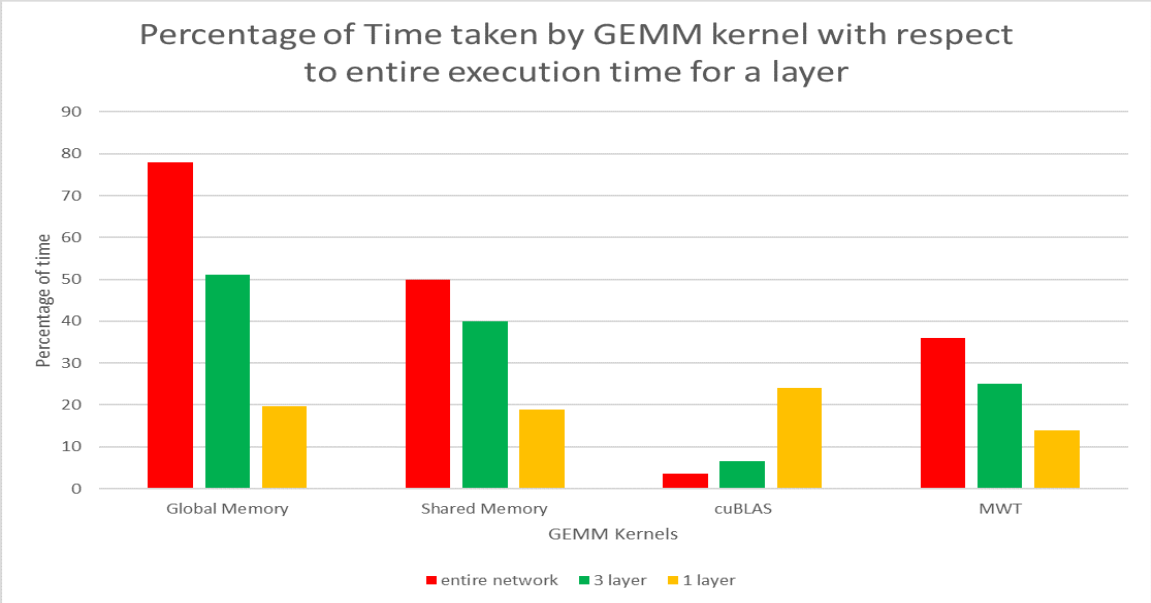


Figure 5.7: Percentage of execution time taken by gemm kernels compared to total execution time for different layers of Yolo Net on TX2

The graph 5.8 shows the IPC of different gemm kernels for different layers of Yolo Net on TX2.

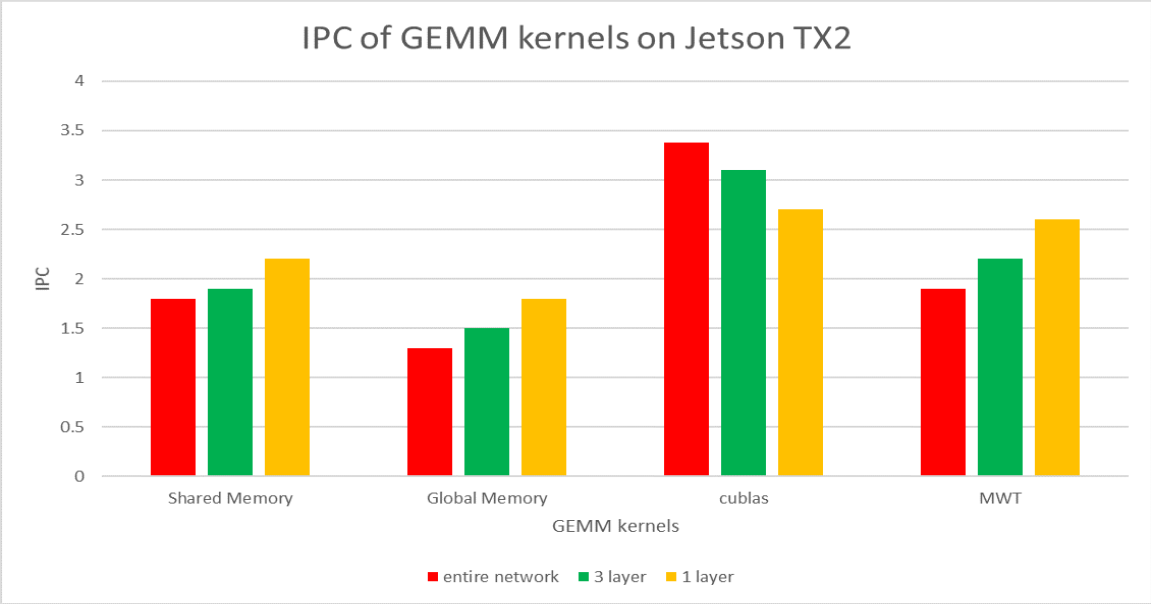


Figure 5.8: IPC of gemm kernels for different layers of Yolo Net on TX2

IPC metrics is defined as Instructions executed per cycle. Nvidia's profiler nvprof reports IPC in terms of number of instructions executed by an SM per cycle. As each scheduler has a dual dispatch unit, in an ideal scenario we can have an IPC of 8 for both Pascal and Maxwell Architecture. The Execution Time and IPC are directly proportional metrics. The faster instructions are executed the more number of instructions can be scheduled and executed per cycle. Except for the cuBLAS GEMM kernel the relative time spent on GEMM kernels increases as we go from 1 layer to the entire network. cuBLAS is highly optimized such that it takes only 8 percent of the total execution for the entire Yolo network. The possible reason behind cuBLAS GEMM being slow for the 1st layer is that, for the first layer it uses a shared memory of size 32x32 and possible the work done per thread is not a lot. For further cuBLAS GEMM calls after the first it was observed that the shared memory size used was 128x128 and it arguable used 2D register blocking for increasing work done per thread in both dimensions, this can be confirmed by low achieved occupancy of cuBLAS GEMM kernels.

5.4 Architecture Modeling Results

In the section 4.5 above we discussed modeling of embedded GPUs, Jetson Tx1 and Tx2 by studying the Maxwell, Pascal architecture and GPGPU sim internal architecture. In this section we will compare the IPC results obtained by running Yolo net on GPGPU simulator and real hardware. We ran the first 3 layers of Yolo net with Global and Shared memory GEMM kernel on the GPU simulator and hardware, the graph 5.9 shows the IPC correlation for Jetson TX1 and 5.10 shows IPC correlation for Jetson TX2 .

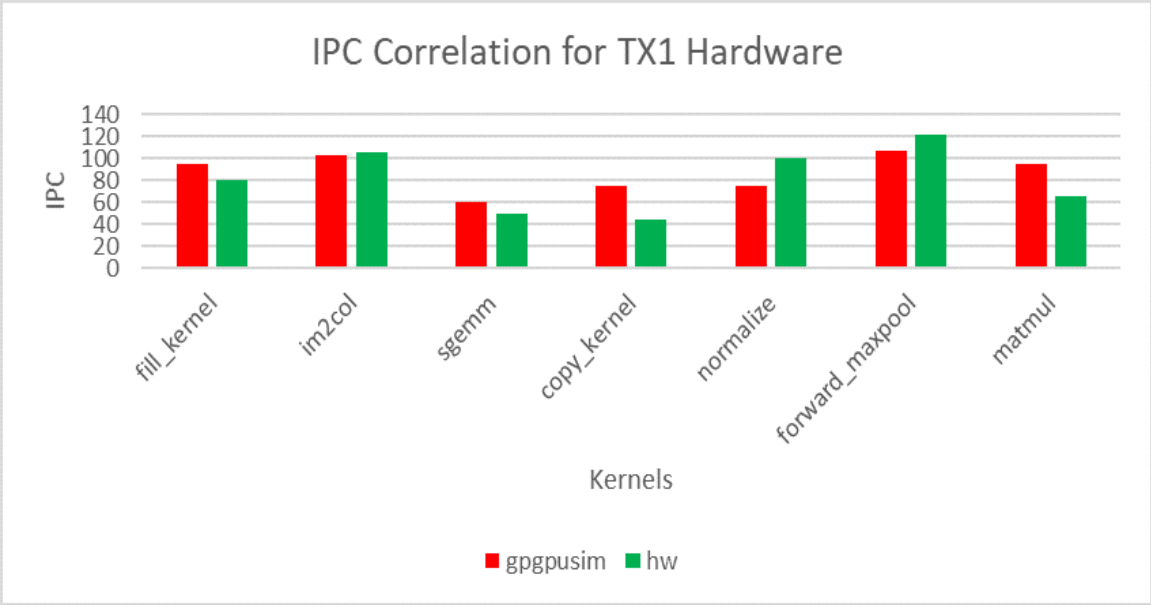


Figure 5.9: IPC Correlation between Hardware and GPU simulator for Yolo Net 3 layer on Jetson TX1

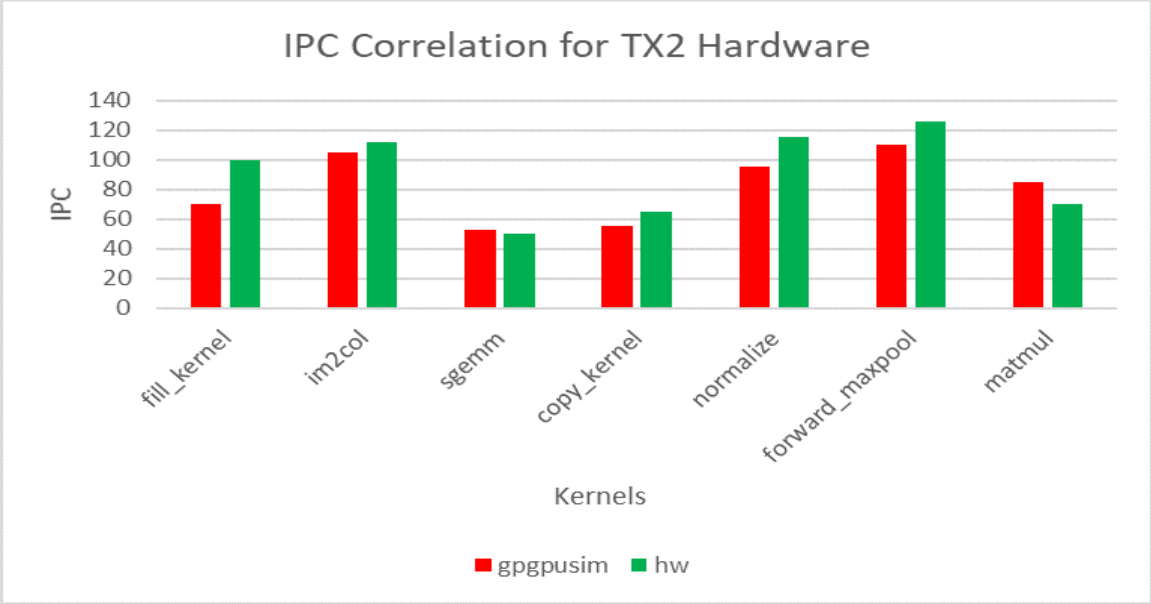


Figure 5.10: IPC Correlation between Hardware and GPU simulator for Yolo Net 3 layer on Jetson TX2

For IPC comparison of an embedded GPU with server-class GPU we ran Yolo net on Tesla c2050 GPU simulated by GPGPU sim, Tesla c2050 GPU being a server class GPU has much higher but the trend of IPC reported for kernels is similar to Jetson TX2. The graph 5.11 shows the IPC correlation between Jetson Tx1 and Tesla c2050.

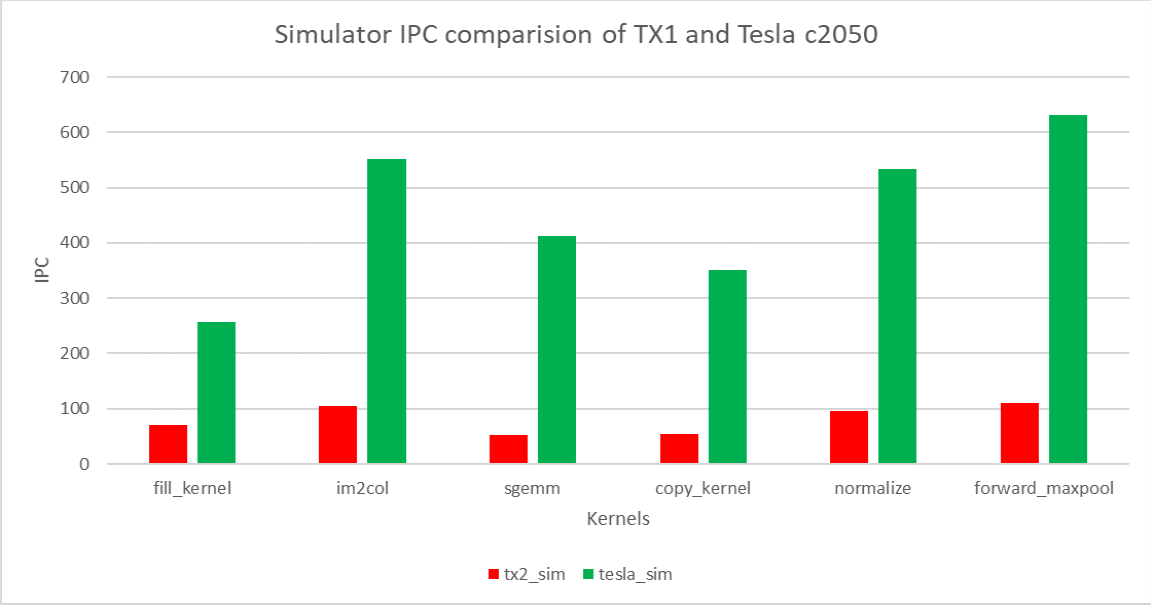


Figure 5.11: IPC Correlation between Jetson TX2 and Tesla c2050

GPGPU-Sim calculates IPC over an SM where the number of instructions being executed is incremented per thread. NVIDIA's profiler program calculates the IPC of the execution of a program run on a GPU by using the formula:

ipc= SUM(sm-inst-executed) / SUM(sm-active-cycles) This results in the average IPC of a single SM. sm-inst+executed - The number of warp instructions executed count at the point where the instruction must complete (cannot be rolled back due to speculative execution). Fully predicated off instructions are count. sm-active-cycles - The number of cycles the SM had at least 1 resident warp.

NVIDIA Perfworks provides the following metrics: sm[sp]-inst-executed-avg, sumper-active, elapsed-cycle. The sum variant would be IPC calculated over the full GPU. The elapsed-cycles variant includes cycles the SM is not active. From the graph 5.9 and 5.10 we can observe that although the GPU sim IPC and hardware IPC do no match completely but the simulator's IPC follows the similar trend as obtained from hardware. Table 5.1 and 5.2 shows the IPC correlation between simulated Jetson TX1 and TX2 with actual hardware.

Table 5.1: IPC correlation for Jetson TX1 Hardware and Simulator

Kernels	Jetson TX1 (H/W)	Jetson TX1(Sim)	Error(percentage)
Fill-kernel	80	95	15
Im2Col-kernel	103	105	2
sgemm-kernel	60	50	16
Copy-kernel	75	45	40
forward-maxpool-kernel	107	121	13

Table 5.2: IPC correlation for Jetson TX2 Hardware and Simulator

Kernels	Jetson TX2 (H/W)	Jetson TX2(Sim)	Error(percentage)
Fill-kernel	70	100	30
Im2Col-kernel	105	112	6
sgemm-kernel	53	50	6
Copy-kernel	55	65	18
forward-maxpool-kernel	110	126	14.5

CHAPTER 6: CONCLUSIONS

This research explored and studied various GPU architectures, GPU simulators and convolutional neural networks. I tried to search for an open source CUDA GEMM library which can be compatible with GPGPU-sim but CLblast is the only publicly available open source CUDA library but is not compatible with GPU simulators. So, we decided to build an optimized library for GEMM but could not match cuBLAS Gemm API performance. Yolo Net for image detection with our customized GEMM kernels successfully ran on Jetson TX1 and Jetson TX2 hardware and the simulated platform as as well. This research opens up GPU architecture exploration for embedded GPUs on GPU simulators specially for deep Learning applications The IPC results from the simulated platform did not exactly match the hardware IPC but results give us a good idea to predict whether a certain architectural change will boost the GPU performance.

CHAPTER 7: Future Work

In this research we explored some optimizations for GEMM kernel and implemented it but the optimized kernels were not able to match the performance of cuBLAS GEMM. So, we can try to improve the performance of our GEMM kernels by using advanced level optimizations like 2D Register blocking and software prefetching. Along with improving GEMM kernels we can improve the Architecture modeling of Jetson TX1 and TX2 to decrease the IPC correlation error. One area to work on is to perform micro benchmarks on the Jetson TX1 and TX2 to obtain accurate latencies for all the execution paths. Other areas that could be worked on include updating GPGPU-Sim's memory hierarchy, updating GPGPU-Sim's top-level organization, and adding new clocking features, such as boost clock and different levels of clock gating. Support for Volta Architecture and Tensor cores in GPU simulator is one thing to look forward to.

REFERENCES

[1] S. K. Pal and S. Mitra, "Multilayer perceptron, fuzzy sets, classifiaction," 1992.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[3] R. Smith, "Nvidia volta unveiled: Gv100 gpup and tesla v100 accelerator announced," 2017.

[4] G.-B. D. L. Inference, "A performance and power analysis," *Whitepaper, November*, 2015.

[5] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 163–174, April 2009.

[6] C. Cuda, "Programming guide," 2012.

[7] C. Nugteren, "Clblast: A tuned opencl blas library," *arXiv preprint arXiv:1705.05249*, 2017.

[8] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," *arXiv preprint*, 2017.

[9] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[10] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.

[11] G. Malhotra, S. Goel, and S. R. Sarangi, "Gputejas: A parallel simulator for gpu architectures," in *High Performance Computing (HiPC), 2014 21st International Conference on*, pp. 1–10, IEEE, 2014.

[12] L. M. P. d. Oliveira, *A framework for scientific computing with GPUs*. PhD thesis, Faculdade de Ciências e Tecnologia, 2012.

[13] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[14] Y. LeCun *et al.*, "Lenet-5, convolutional neural networks," *URL: http://yann. lecun. com/exdb/lenet*, p. 20, 2015.

[15] C. Nvidia, "Cublas library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, no. 27, p. 31, 2008.

[16] J. Redmon, "Darknet: Open source neural networks in c," *h ttp://pjreddie. com/darknet*, vol. 2016, 2013.

[17] D. Mor, "Gpgpu for embedded systems," *White Paper*, 2016.

[18] J. Nijhuis, L. Spaanenburg, and F. Warkowski, "Structure and application of nnsim: a general-purpose neural network simulator," *Microprocessing and Microprogramming*, vol. 27, no. 1-5, pp. 189–194, 1989.

[19] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[20] T. Aaamodt and A. Boktor, "Gpgpu-sim 3. x: A performance simulator for many-core accelerator research," in *International Symposium on Computer Architecture (ISCA), http://www. gpgpu-sim. org/isca2012-tutorial*, 2012.

[21] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, ACM, 2014.

[22] J. Ho and R. Smith, "Nvidia tegra x1 preview and architecture analysis," 2015.

[23] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CoRR*, vol. abs/1311.2524, 2013.

[24] G. E. Hinton, "Deep belief networks," *Scholarpedia*, vol. 4, no. 5, p. 5947, 2009.

[25] T. Greene, "Squeeze your'net links," *Network World*, vol. 14, no. 28, p. 1, 1997.

[26] C. Nugteren, *Improving the programmability of GPU architectures*. PhD thesis, Ph. D. thesis, Department of Electrical Engineering, Eindhoven University of Technology, 2014.

[27] N. C. PTX, "Parallel thread execution," *NVIDIA Corp., Jun*, 2008.