# Parallel Programming Using CUDA

***BE Final Year Project***
***By***

**Anurag Kumar Jaiswal  Roll No: 001311001048**

**Satish Kumar  Roll No: 001311001002**

**Supervisor:**
**Mr. Utpal Kr. Ray**
Assistant Professor


**Department of Information Technology**
**Jadavpur University**

**Kolkata**

# ACKNOWLEDGEMENT

We take this opportunity to express our sincere thanks to our guide **Prof. Utpal Kr. Ray** for his support, encouragement and advice. His constant guidance was invaluable in the realization of this report.

**Anurag Kumar Jaiswal & Satish Kumar**
**Jadavpur University**
May 15, 2017

# ABSTRACT

This report documents our final year project, which is about parallel programming using CUDA, the NVIDIA GPU architecture with support for general purpose computing. The purpose of this report is to uncover the qualities of CUDA as a parallel computing platform, determining the performance of an algorithm when it is implemented using parallel programming approach.

We examine this by implementing an algorithm to find a minimum in an array of elements using parallel programming approach. We measure the time the algorithm takes to complete its task for different size of an array and made a time-vs.-size graph.

We also implement parallel version of a standard Matrix Inversion algorithm and study the performance of the parallel algorithm when it is executed in CPU and also when it is executed in the GPU using CUDA.

# TABLE OF CONTENTS

# 1. INTRODUCTION

## 1.1 Graphic Processor Units

A graphic processor unit is simply a processor attached to the graphics card used in video games, PlayStations and computers. The way they are different from the CPUs, the central processing units, are that they are massively threaded and parallel in their operations. This is because of the nature of their work – they are used for fast rendering; same operation is carried out for each of the pixels in the image. Thus, they have more transistors devoted to data processing rather than flow control and data caching.

Today graphic processor units have outdated their primary purpose. They are being used and promoted for scientific calculations all over the world by the name of GPGPUs or general purpose GPUs; engineers are achieving several times speed-up by running their programs on GPUs. Applications fields are many: image processing, general signal processing, physics simulations, computational biology, etc.

## 1.2 Motivation

Graphic Processor Units can speed-up the computation manifolds over the traditional CPU implementation. Image processing, being inherently parallel, can be implemented quite effectively on a GPU. Thus, many applications which are otherwise run slowly can be fastened up, and can be put to useful real-time applications. This was the motivation behind my final year project.

NVIDIA CUDA is a parallel programming model and software, which has developed specifically to address the problems of efficiently program the GPU as well as be compatible with a wide variety of GPU cores available in the market. Further, being an extension to the standard C language, it presents a low-learning curve for the programmers, as well giving them flexibility to put in their creativity in the parallel codes they write.

# 2. PARALLEL PROGRAMMING

In computing, a parallel programming model is an abstraction of parallel computer architecture, with which it is convenient to express algorithms and their composition in programs. A parallel program is composed of simultaneously executing processes.

- There are two different approaches of achieving parallelism while solving a problem.
- They are CONTROL PARALLEL approach and DATA PARALLEL approach.
- In Control Parallel approach, parallelism is achieved by applying different operations to the data set simultaneously. It is about attaching computations to the data set. Sometimes it is easier to implement this approach, but speedup achieved is limited. Pipelining is a special case of Control Parallel Approach.
- In Data Parallel approach, parallelism is achieved by applying same operation to the different data set simultaneously. It is about attaching different data set to the computations. Implementing this is more complex, but more speedup can be achieved through this method.

## 2.1 A Sample Problem

Find the ratio of the product and the sum of all the elements of a given matrix.

### 2.1.1 The Sequential Solution

To calculate Tseq; let's assume every addition takes x amount of time, every multiplication takes 2x amount of time and the division takes 3x amount of time. So, Tseq = $N^2 2x + N^2 x + 3x$.

## 2.1.2 The Control Parallel Solution

There are three different distinct computation type here - Multiplication, Addition and Division. Out of these, first two can run in parallel; and the last one has to start once the first two are complete. So a two processor machine is good enough for running this solution.

Out of this Multiplication and Addition; multiplication will take more time ($N^2 2x$) to complete than the addition ($N^2 x$). So, the processor running the addition (after his addition operation gets completed) has to wait for the other processor to complete multiplication. Once both these operation are over, either of the processor can take up the division which takes 3x amount of time to complete.

So, $T_{par} = N^2 2x + 3x$

Speedup $S = (N^2 2x + N^2 x + 3x) / (N^2 2x + 3x)$

$= (3N^2 + 3) / (2N^2 + 3)$

$= 1.5$ (N being large)

And, Efficiency $E = S/P = 1.5/2 = 0.75$

## 2.1.3 The Data Parallel Solution

For the sake of simplicity, let's assume there are only two processors available. So, the total data (here, it's a matrix) need to be divided into two halves (row wise division is good). The first half goes to the processor 1 and the second half goes to the processor 2. Each of this processor will run the computations (multiplication and addition) on their own data set. Once, the partial products and sums are available; either of the processor can compute the total and then the division.

So, the time taken to compute the partial product by any of the processor is ($N^2 2x$)/2. And the time take to compute the partial sum by any of the processor is ($N^2 x$)/2. So, the total time taken by any of the processor to run his own part of

computation is $(N^2 2x + N^2 x)/2$ or $(3N^2 x)/2$.

After this one of the processor will compute the total product (takes 2x) and total sum (takes x). It is to be noted that this work cannot be parallelized. And the last piece of computation done by one of the processor is division (takes 3x).

So, Tpar = $(3N^2 x)/2 + 2x + x + 3x$
= $(3N^2 x)/2 + 6x$
= $(3N^2 x + 12x)/2$

Speedup S = $((N^2 2x + N^2 x + 3x) / (3N^2 x + 12x)) * 2$
= $((3N^2 + 3) / (3N^2 + 12)) *2$
= 2 (N being large)

And, Efficiency E = S/P = 2/2 = 1.0

## 2.2 Multithreaded Programming – A Type of Parallel Programming

The threads model of parallel programming is one in which a single process (a single program) can spawn multiple, concurrent "threads" (sub-programs). Each thread runs independently of the others, although they can all access the same shared memory space (and hence they can communicate with each other if necessary). Threads can be spawned and killed as required, by the main program.

A challenge of using threads is the issue of collisions and race conditions, which can be addressed using synchronization. If multiple threads write to (and depend upon) a shared memory variable, then care must be taken to make sure that multiple threads don't try to write to the same location simultaneously. There are mechanisms when using threads to implement synchronization, and implement mutual exclusivity (mutex variables) so that shared variables can be locked by one thread and then released, preventing collisions by other threads. These mechanisms ensure threads must "take turns" when accessing protected data.

# 3. GPU PROGRAMMING USING CUDA

GPU computing is possible because today's GPU does much more than render graphics: It sizzles with a teraflop of floating point performance and crunches application tasks designed for anything from finance to medicine.

## 3.1 What is CUDA?

CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing – an approach known as GPGPU. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

The CUDA platform is designed to work with programming languages such as C, C++ and Fortran. This accessibility makes it easier for specialists in parallel programming to utilize GPU resources, as opposed to previous API solutions like Direct3D and OpenGL, which required advanced skills in graphics programming.

## 3.2 How CUDA works?

The GPU programming model is very different from a CPU programming model as it inherently supports parallel programming. A usual template followed to program a GPU is as below:

- Transfer the data to the GPU memory.
- Launch the GPU kernel from the host program.
- Wait till GPU finishes off the calculations.
- Transfer the results back to CPU memory.

Thus, a CUDA program consists of two parts in its code: a) **host program** which runs on CPU and b) **kernel** which does the calculations on GPU. A brief description of the various programming features is as below:

- The execution of the kernels is done in threads and blocks. The structure of the grid of threads and blocks is passed in the host code while calling the kernel using a unique <<<…>>> syntax. A typical kernel call looks like:

  **funcAdd<<<Grid,Blocks,1>>>(A, B, C)**

- These threads and blocks are executed on the various multiprocessors of the GPU. The threads of each block are executed concurrently in one multi-processor; each of them may process some maximum number of blocks depending on the resources used by each thread. As the blocks get executed, new blocks are launched on the vacated processors.

- The threads are processed in groups of 32 called warps. The splitting into warps is done according to increasing thread IDs.

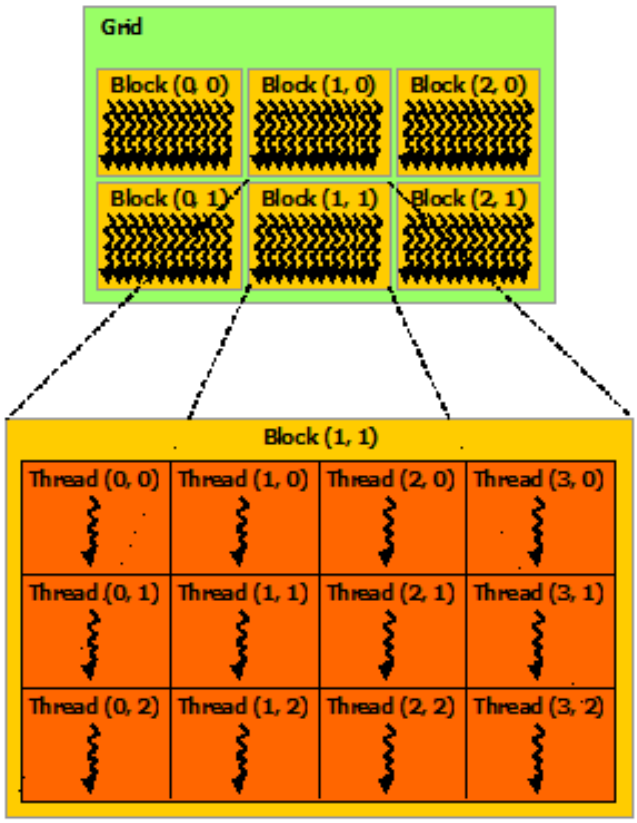
- Each thread and block has a unique ID, which is defined by a 3 component vector namely **threadIdx** and **blockIdx** respectively. The thread IDs are with respect to their position in the block. A typical grid of threads and blocks is shown in figure1.

- Threads within a block can synchronize using a barrier synchronization function **__syncthreads()**. This function stops the execution of threads until all the threads have reached that point.

- The threads are executed in parallel unless their paths diverge or they read/write from the same memory location. In the former case, the processor executes each

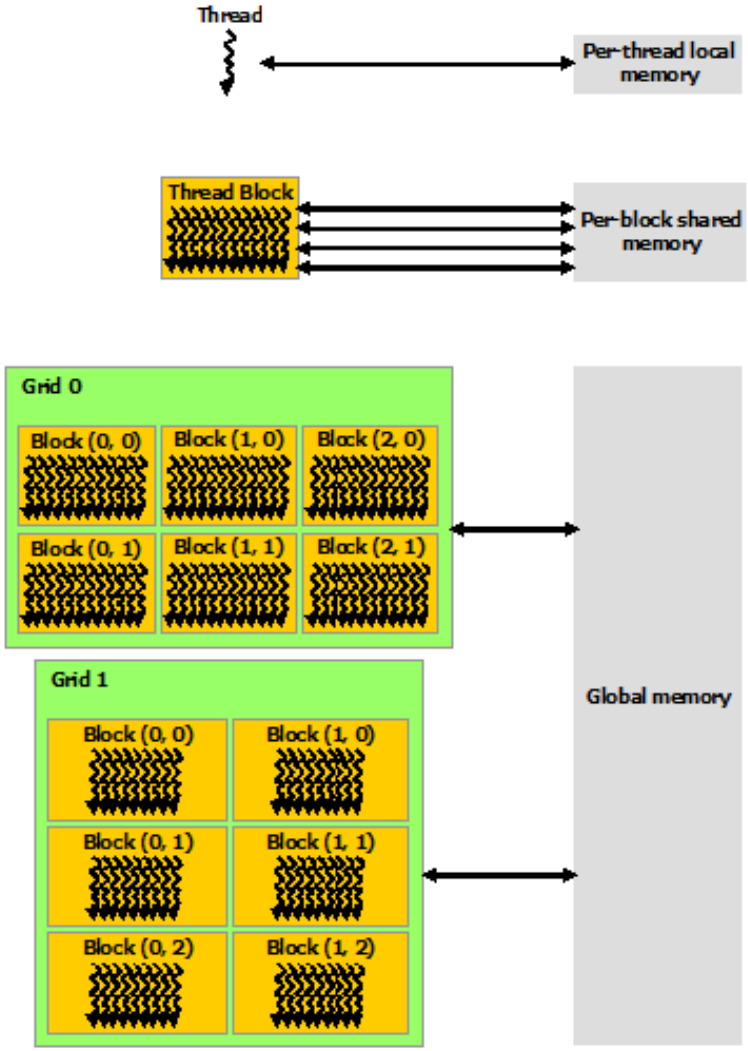
path serially, disabling threads on other paths until the paths converge again. In the latter case too, the access is serialized, but in the case of non-atomic write, the outcome of the write is not guaranteed.

GPU derives its computing from an array of Streaming Multiprocessors (SMs). Each multiprocessor consists of eight Scalar Processor cores and a multithreaded instruction unit. The multiprocessor handles multiples threads using the architecture called SIMT, which maps each thread to a scalar processor. Further each multiprocessor has on-chip memory of different types: a set of 32-bit registers, an on-chip shared memory shared by all the scalar processors, a constant cache to speed up reading from constant memory and a texture cache which speeds up read from the texture memory space. A schematic of the GPU hardware is shown in figure 3.




## 3.3 GPU Memory Hierarchy

GPU has different levels of memory - global memory, shared memory, constant memory, texture memory and local memory and  registers.


Thread
Per-thread local memory
Thread Block
Per-block shared memory
Grid 0
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 1
Block (0, 0)
Block (1, 0)
Block (0, 1)
Block (1, 1)
Block (0, 2)
Block (1, 2)
Global memory

### 3.3.1 Registers and local memory

Each multiprocessor has certain number of 32-bit registers. They are allocated per thread and only that thread can access the contents of the register. These are the fastest accessible memories, and are allocated at the compile time depending on the requirement of a thread. If the threads in a block need more registers than provided in a multiprocessor, the kernel fails to launch. Local Memory is also available per thread and the compiler automatically allocates certain variables on the local memory, like a big array. Local memory is not cached and hence its access is as costly as global memory.

### 3.3.2 Shared Memory

Shared memory is an on-chip memory and hence, it is faster than local or global memory. Every multiprocessor has 16kB of shared memory, and is accessible by all the threads in a block. The shared memory of one block cannot be accessed by another block even if they are running on the same multiprocessor. To achieve high bandwidth, the shared memory is divided into memory banks, which can be accessed simultaneously. If threads in the same half-warp access the same bank address, then there is a bank conflict and the access is serialized, thus reducing the effective bandwidth.

### 3.3.3 Global Memory

It is an off-chip memory and is accessible to all threads. The data transferred from CPU is stored in the global memory initially. The memory is connected to all the processors through a wide bus, which can be up to 512 bits wide. Since the memory is not cached, access to the global memory is expensive and should be avoided as far as possible. The memory bandwidth is effectively utilized when the threads in a half-warp access the memory in a sequence, resulting in coalescing of the memory transactions.

### 3.3.4 Constant Memory

Used for storing constant values that does not change over time. Constant memory resides in the global memory. Constant memory accesses are cached in constant cache. A separate constant cache exists for each multiprocessor and it is a read only cache.
For example the constant memory on any architecture is 64KB
Cache size per multiprocessor on Fermi and Kepler are 8KB

### 3.3.5 Texture Memory

CUDA also supports a part of the texturing hardware which is used for graphics processing. This allows the texture memory to be read using device functions called texture fetches. As the texture memory is cached, it is much faster than global memory access. It also supports various attributes like texture coordinates, which can be used to access the texture elements, interpolation options, normalization, etc.

# 4. ADD TWO VECTORS USING CUDA

The following is an insight into the effectiveness of CUDA in solving a general problems  like adding two vectors of equal sizes.

## 4.1 The Problem Statement

The aim of this problem is to add corresponding elements of two vectors of equal sizes using the CUDA framework by using the computation power of the highly parallel structure of the GPU. The aim of this task is to compare the time required by CPU based sequential program with GPU based parallel program.

## 4.2 Approach

The domain of GPU Programming using the CUDA framework is relatively new, so extensive materials for study were not available. The approach towards solving the problem was hence somewhat  intuitive.

```
__global__ void VecAdd(const int* A,const int* B,int* C,int N)
{
        int i = blockDim.x * blockIdx.x + threadIdx.x;
        if(i<N)
                C[i] = A[i] + B[i];
}
```

## 4.3 Results

| Array Size | Memory Required | Time(in µsec) for CPU | Time(in µsec) for GPU |
|---|---|---|---|
| 2000 | 0.034 | 28 | 20 |
| 4000 | 0.069 | 59 | 19 |
| 8000 | 0.137 | 117 | 20 |
| 16000 | 0.275 | 237 | 21 |
| 32000 | 0.549 | 527 | 21 |

The execution time was recorded for different sizes of the array and the following table was obtained.

**Table 4.1** Time Required to add two vectors of different sizes using both CPU & GPU

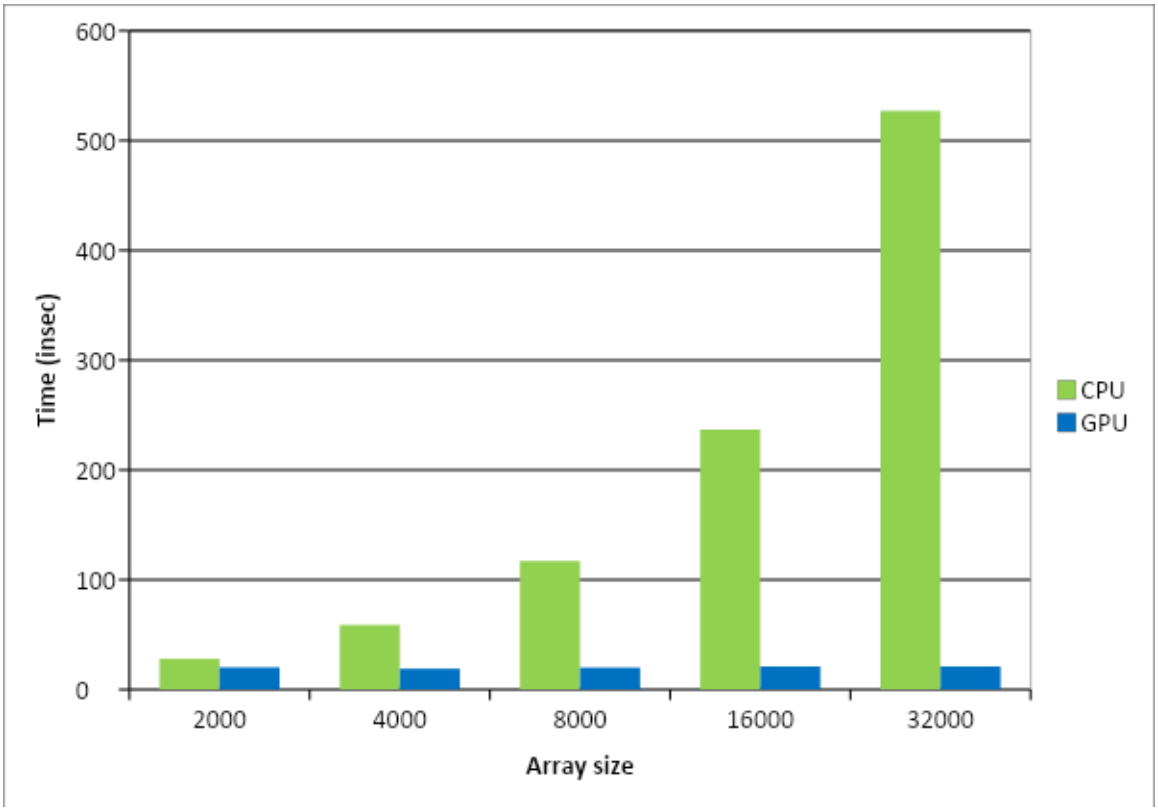



***Figure 4.1*** *Comparison between times taken to add two vectors using CPU & GPU*

# 5. GPU BASED MATRIX MULTIPLICATION USING CUDA

## 5.1 The Problem Statement

The aim of this problem is to multiply two vectors of equal dimensions using the CUDA framework by using the computation power of the highly parallel structure of the GPU. The aim of this task is to compare the time required by CPU based sequential program with GPU based parallel program with and without shared memory.

## 5.2 Approach

**Without using shared memory**

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
        ...
        for (int e = 0; e < A.width; ++e)
                Cvalue += A.elements[row * A.width + e] * B.elements[e * B.width + col];
        ...
}
```

The lines in loop over the entries of the row of A and the column of B (these have the same size) needed to compute the (row, col)-entry of the product, and the sum of these products are accumulated in the Cvalue variable. Matrices A and B are stored in the device's global memory in row major order, meaning that the matrix is stored as a one-dimensional array, with the first row followed by the second row, and so on. Thus to _nd the index in this linear array of the (i; j)-entry of matrix A, for example, we compute (i_width of A) to second the starting index of the ith row, and then add j to go to the jth entry in that row. Finally, the last line of the kernel copies this product into the appropriate element of the product matrix C, in the device's global memory.

There are three functions :

1. main()

This function generates two random matrices with dimensions that are read from the commandline. Matrix A is filled randomly with numbers and Matrix B. Then main() calls MatMul() which multiplies these and places the product in Matrix C.

2. MatMul(Matrix A, Matrix B, Matrix C)

This function takes two matrices A and B as input, and fills the entries of matrix C with the product. It first allocates memory on the device for matrix A, and copies A

onto the device's global memory. Then it does the same for matrix B. It allocates space on the device for C, the matrix product, computes the number of blocks needed to cover the product matrix, and then launches a kernel with that many blocks. When the kernel is done, it reads matrix C o_ of the device, and frees the global memory.

3. MatMulKernel(Matrix A, Matrix B, Matrix C)

This runs on the device and computes the product matrix. It assumes that A and B are already in the device's global memory, and places the product in the device's global memory, so that the host can read it from there.

The code just given works and is fast, but could be so much faster if we take advantage of shared Memory.

**Using shared memory**

A Common Programming Strategy::

- Global memory is much slower than shared memory
- So, a profitable way of performing computation on the device is to tile data to take advantage of fast shared memory:
    - Partition data into subsets that fit into shared memory
    - Handle each data subset with one thread block by:
        - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
        - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
        - Copying results from shared memory to global memory

Looking at the loop in the kernel code, we notice that each thread loads (2*A.width) elements from global memory | two for each iteration through the loop, one from matrix A and one from matrix B. Since accesses to global memory are relatively slow, this can bog down the kernel, leaving the threads idle for hundreds of clock cycles, for each access.

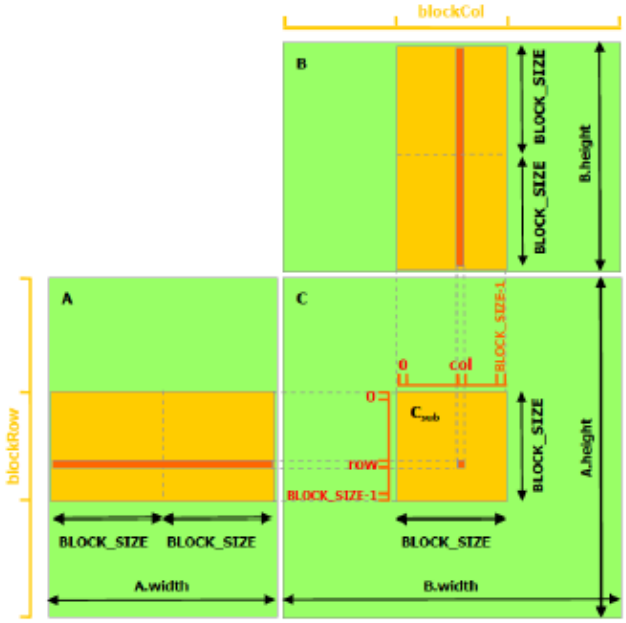



**FIG 5.1**

One way to reduce the number of accesses to global memory is to have the threads load portions of matrices A and B into shared memory, where we can access them much more quickly. Ideally, we would load both matrices entirely into shared memory, but unfortunately, shared memory is a rather scarce resource, and won't hold two large matrices. So we will content ourselves with loading portions of A and B into shared memory as needed, and making as much use of them as possible while they are there.

One way of doing this is suggested by Figure5.1. Matrix A is shown on the left and matrix B is shown at the top, with matrix C, their product, on the bottom-right. This is a nice way to lay out the matrices visually, since each element of C is the product of

the row to its left in A, and the column above it in B. The row and column sort of aim for their product in C. In that figure, and in our sample code, we will use square thread blocks of dimension BLOCK_SIZE * BLOCK_SIZE and will assume that the dimensions of A and B are all multiples of BLOCK_SIZE.

Again, each thread will be responsible for computing one element of the product matrix C. For reference, consider the item highlighted in red in the matrix C, in Figure 5.1 . (Note that the yellow square in matrix C represents one thread-block's-worth of elements, whereas the red box inside the yellow square represents a single entry in C, and hence a single thread.) Our thread computes this element in C by multiplying together the red row shown in A, and the red column shown in B, but it will do it in pieces, as we will now discuss.

We may decompose matrices A and B into non-overlapping submatrices of size BLOCK_SIZE*BLOCK_SIZE. If we look at our red row and red column, they will pass through the same number of these submatrices, since they are of equal length. If we load the left-most of those submatrices of matrix A into shared memory, and the top-most of those submatrices of matrix B into shared memory, then we can compute the first BLOCK_SIZE products and add them together just by reading the shared memory. But here is the benefit: as long as we have those submatrices in shared memory, every thread in our thread block (computing the BLOCK_SIZE*BLOCK_SIZE submatrix of C) can compute that portion of their sum as well from the same data in shared memory.

When each thread has computed this sum, we can load the next BLOCK_SIZE * BLOCK_SIZE submatrices from A and B, and continue adding the term-by-term products to our value in C. And after all of the submatrices have been processed, we will have computed our entries in C. The kernel code for this portion of our program is shown below

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C){

        // Block row and column
        int blockRow = blockIdx.y, blockCol = blockIdx.x;

        // Each thread block computes one sub-matrix Csub of C
        Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

        // Each thread computes 1 element of Csub accumulating results into Cvalue
        float Cvalue = 0.0;

        // Thread row and column within Csub
        int row = threadIdx.y, col = threadIdx.x;

        // Loop over all the sub-matrices of A and B required to compute Csub
        for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {

                // Get sub-matrices Asub of A and Bsub of B
                Matrix Asub = GetSubMatrix(A, blockRow, m);
                Matrix Bsub = GetSubMatrix(B, m, blockCol);

                // Shared memory used to store Asub and Bsub respectively
                __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
                __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

                // Load Asub and Bsub from device memory to shared memory
                // Each thread loads one element of each sub-matrix
                As[row][col] = GetElement(Asub, row, col);
                Bs[row][col] = GetElement(Bsub, row, col);
                __syncthreads();

                // Multiply Asub and Bsub together
                for (int e = 0; e < BLOCK_SIZE; ++e)
                        Cvalue += As[row][e] * Bs[e][col];
                __syncthreads();
        }

        // Each thread writes one element of Csub to memory
```

## Calling the Kernel

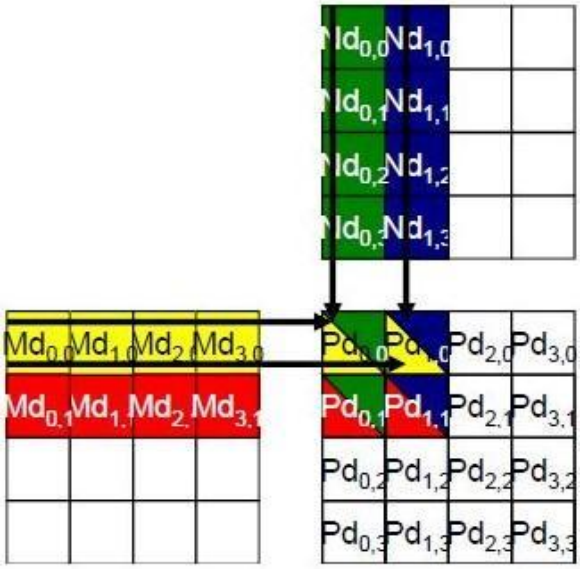
- The kernel in the MatMulKernel() kernel makes use of the __syncthreads() call. Whenever a thread calls __syncthreads(), all threads in that thread's block must compute up to this point before any thread is allowed to pass that point. With the first call to __syncthreads() we thus insure that every entry of the submatrices of A and B have been loaded into shared memory before any thread begins its computations based on those values. The second call to __syncthreads() ensures that every element of the submatrix of C has been processed before we begin loading the next submatrix of A or B into shared memory. Note that while the __syncthreads() primitive enables this sort of inter-thread synchronization, its use does minimize parallelism and may degrade performance if not used wisely and sparingly

  Let's take a look at the host code related to invocation of the kernel:

```
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
        ...
        // Invoke kernel
        dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
        dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
        MatMulKernel<<>>(d_A, d_B, d_C);
        err = cudaThreadSynchronize();
        printf("Run kernel: %s\n", cudaGetErrorString(err));
        ...
}
```

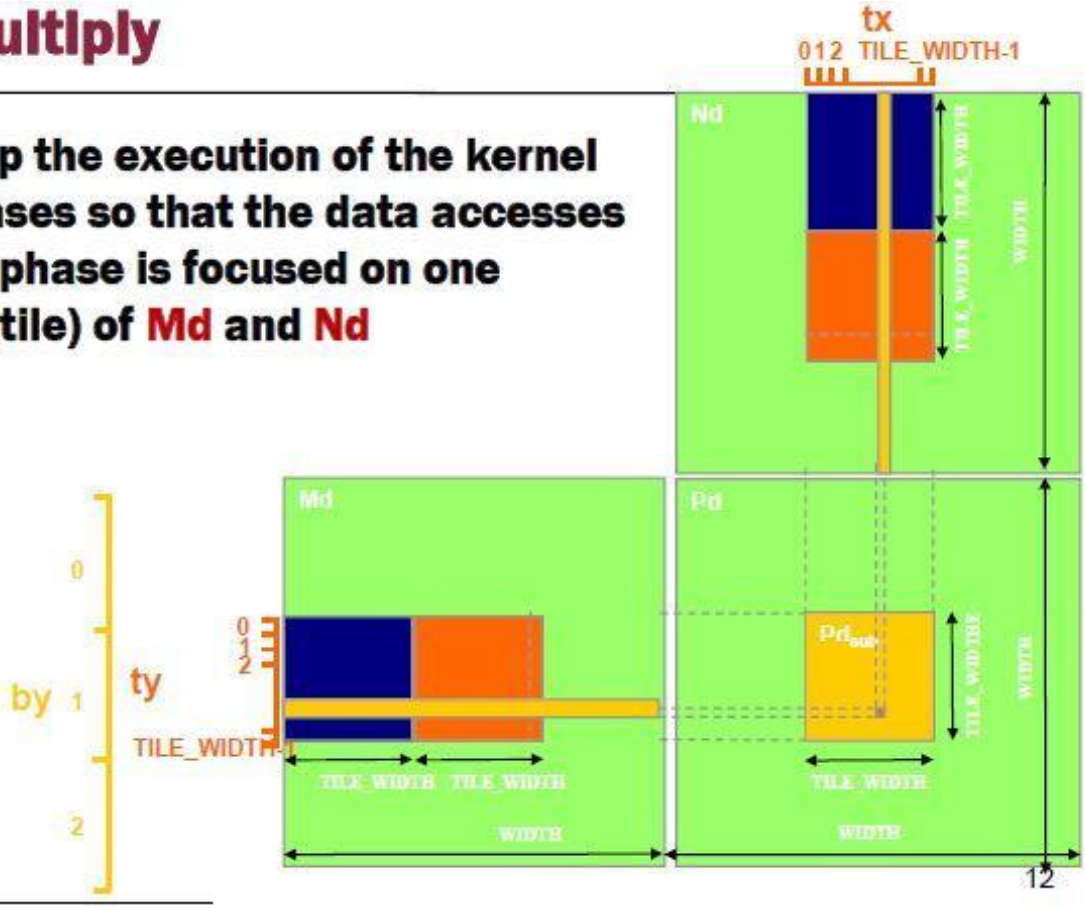
The dim3 structure defined in the CUDA libraries is very useful. It defines a triple of unsigned ints used to hold the dimensions of a grid or block.

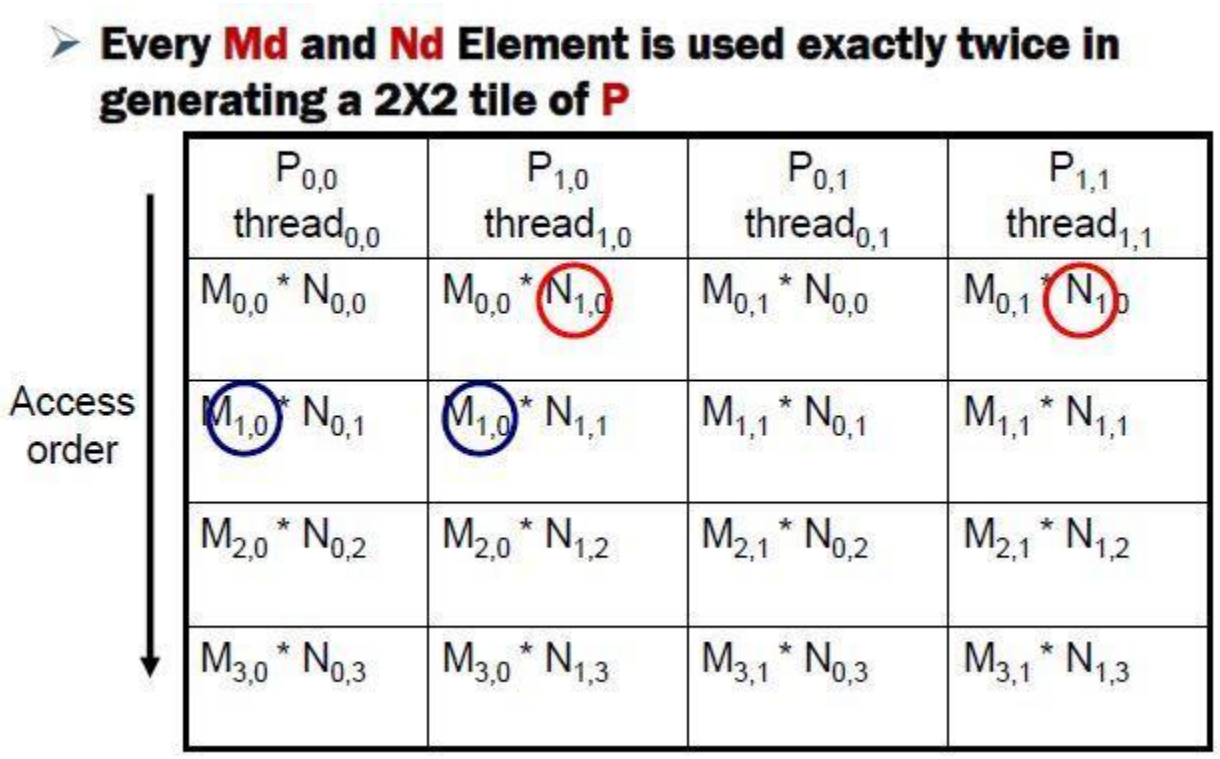
**EXAMPLE**::




# Tiled Multiply

- **Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd**






## 5.3 Results

The execution time was recorded for different sizes of the array and the following table was obtained.

| Matrix Size | Memory Required (in MB) | CPU Time (in sec) | Multiplier | GPU Time (in sec) | Multiplier | GPU Time using shared memory | Multiplier |
|---|---|---|---|---|---|---|---|
| 500*500 | 1.43 | 0.473709 | ----- | 0.122799 | ----- | 0.112445 | ----- |
| 1000*1000 | 5.722 | 3.912552 | 8.26 | 0.249378 | 2.03 | 0.159023 | 1.41 |
| 2000*2000 | 22.888 | 34.003586 | 8.69 | 1.180480 | 4.73 | 0.478651 | 3.01 |
| 4000*4000 | 91.5527 | 286.24081 | 8.42 | 8.540837 | 7.235 | 2.768426 | 5.78 |
| 8000*8000 | 366.211 | 2658.7105 | 9.29 | 67.29404 | 7.88 | 21.516653 | 7.77 |

**Table 5.1 Time required by matrix multiplication using CPU and GPU for different sizes of matrix**

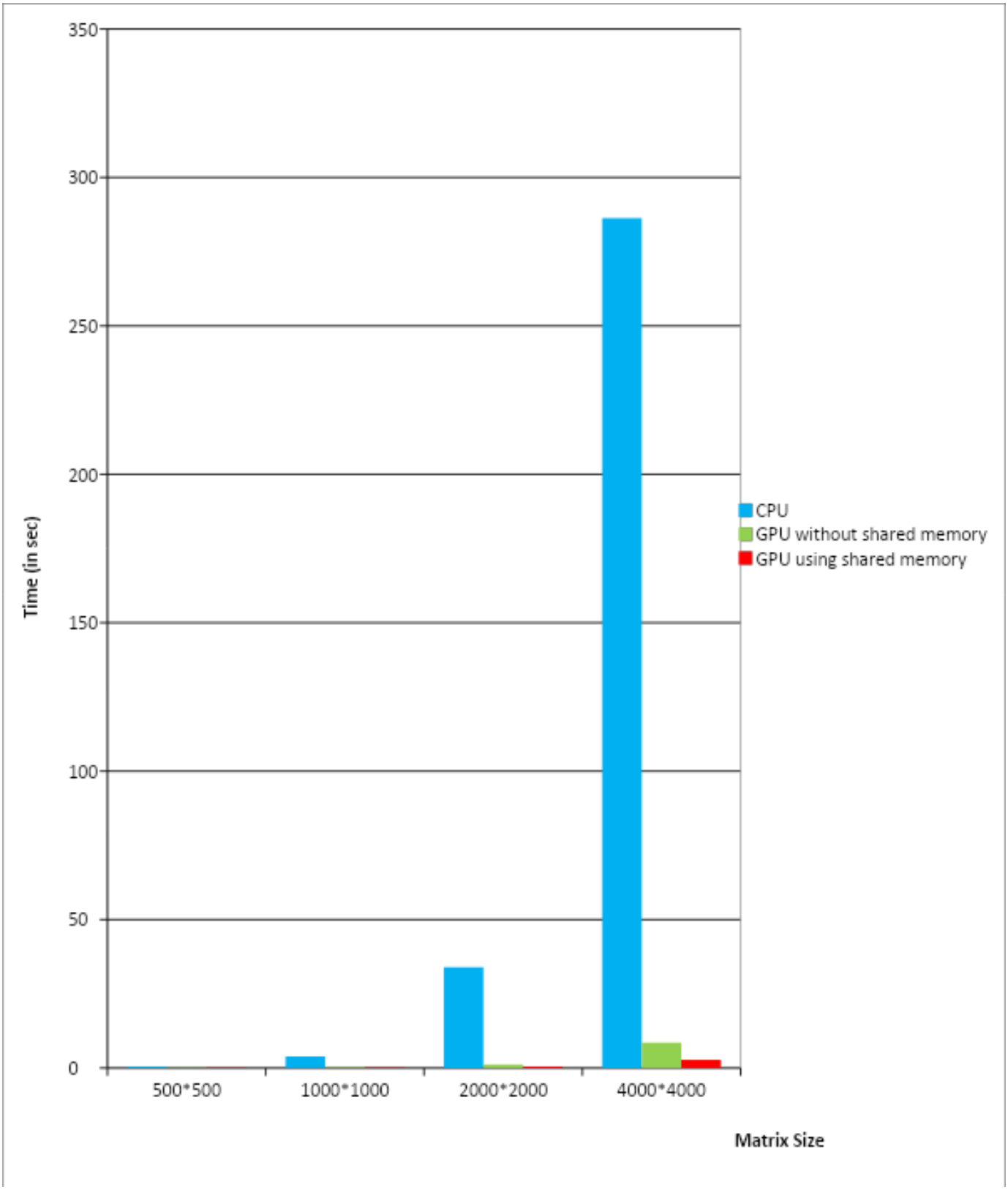



***Figure 5.1*** *Comparison between times taken for matrix multiplication using different methods*

# 6. CONCLUSIONS

The execution time of matrix multiplication reduces to almost one-twentieth when we implement the Matrix Multiplication algorithm in a GPU using CUDA. We can further reduce the execution time if we implement and execute the algorithm using shared memory. The execution time of the parallel program using CUDA in GPU takes less than one-thirtieth of the time taken by a sequential program in a CPU when the order of the input matrix size is 4000 * 4000.

As a conclusion, it can be said that this implementation is successful. The execution time is as low as expected.

# 7. REFERENCES


- https://en.wikipedia.org/wiki/Graphics_processing_unit
- https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units
- https://www.nvidia.com/object/what-is-gpu-computing.html
- http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-microsoft-windows/#axzz3cRosp3v6
- https://developer.nvidia.com/maxwell-compute-architecture
- http://www.notebookcheck.net/NVIDIA-GeForce-840M.105681.0.html
- https://en.wikipedia.org/wiki/Maxwell_(microarchitecture)
- https://en.wikipedia.org/wiki/Gaussian_eli

# 8. APPENDIX A – CUDA INSTALLATION GUIDELINES

The CUDA Framework can be installed on Linux or Windows, and used to program the GPU provided that the GPU is a CUDA Enabled GPU manufactured by NVIDIA. We have installed Linux (Ubuntu 16.04 LTS) and installed CUDA 8.0 in order to write and run CUDA code on the GPU. All the installation files have been submitted to our guide for convenience in future work, and prevention of the version conflict problem in future.

To use CUDA on your system, you will need the following items on your system:

- A CUDA-capable GPU
- The NVIDIA CUDA Toolkit

Below is the step by step guideline for installation of CUDA on LINUX -

## 1. Verify that your GPU is CUDA capable

To verify that your GPU is CUDA-capable, go to your distribution's equivalent of System Properties, or, from the command line, enter:

```
$ lspci | grep -i nvidia
```

If you do not see any settings, update the PCI hardware database that Linux maintains by entering `update-pciids` (generally found in `/sbin`) at the command line and rerun the previous `lspci` command.

If your graphics card is from NVIDIA and it is listed in http://developer.nvidia.com/cuda-gpus, your GPU is CUDA-capable.

## 2. Download the NVIDIA CUDA Toolkit

The NVIDIA CUDA Toolkit is available at http://developer.nvidia.com/cuda-downloads.

Choose the platform you are using and download the NVIDIA CUDA Toolkit.

The CUDA Toolkit contains the CUDA driver and tools needed to create, build and run a CUDA application as well as libraries, header files, CUDA samples source code, and other resources.

# 3. Installing CUDA

- Perform the pre-installation actions.

- **Install repository meta-data**

**Note:** When using a proxy server with aptitude, ensure that `wget` is set up to use the same proxy settings before installing the cuda-repo package.

```
$ sudo dpkg -i cuda-repo-<distro>_<version>_<architecture>.deb
```

- **Update the Apt repository cache**

```
$ sudo apt-get update
```

- **Install CUDA**

```
$ sudo apt-get install cuda
```

- Perform the post-installation actions.

# 4. Post Installation Actions

## 4.1. Environment Setup

The `PATH` variable needs to include `/usr/local/cuda-8.0.61/bin`

To add this path to the `PATH` variable:

```
$ export PATH=/usr/local/cuda-8.0.61/bin${PATH:+:${PATH}}
```

In addition, when using the runfile installation method, the `LD_LIBRARY_PATH` variable needs to contain `/usr/local/cuda-8.0.61/lib64` on a 64-bit system, or `/usr/local/cuda-8.0.61/lib` on a 32-bit system

- To change the environment variables for 64-bit operating systems:

```
$ export LD_LIBRARY_PATH=/usr/local/cuda-8.0.61/lib64\
                         ${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

- To change the environment variables for 32-bit operating systems:

```
$ export LD_LIBRARY_PATH=/usr/local/cuda-8.0.61/lib\
                         ${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

Note that the above paths change when using a custom install path with the runfile installation method.

## 4.2. Verify the Installation

Before continuing, it is important to verify that the CUDA toolkit can find and communicate correctly with the CUDA-capable hardware. To do this, you need to compile and run some of the included sample programs.

**Note:** Ensure the PATH and, if using the runfile installation method, LD_LIBRARY_PATH variables are set correctly.

### 4.2.1. Verify the Driver Version

If you installed the driver, verify that the correct version of it is loaded. If you did not install the driver, or are using an operating system where the driver is not loaded via a kernel module, such as L4T, skip this step.

When the driver is loaded, the driver version can be found by executing the command

```
$ cat /proc/driver/nvidia/version
```

Note that this command will not work on an iGPU/dGPU system.

### 4.2.2. Compiling the Examples

The version of the CUDA Toolkit can be checked by running `nvcc -V` in a terminal window. The `nvcc` command runs the compiler driver that compiles CUDA programs. It calls the `gcc` compiler for C code and the NVIDIA PTX compiler for the CUDA code.

The NVIDIA CUDA Toolkit includes sample programs in source form. You should compile them by changing to `~/NVIDIA_CUDA-8.0.61_Samples` and typing `make`. The resulting binaries will be placed under `~/NVIDIA_CUDA-8.0.61_Samples/bin`.

# 9. APPENDIX B – SYSTEM SPECIFICATIONS

## 8.1 Laptop Specifications

- Model : Sony VAIO SVF15218SN
- Intel® Core™ i5-3337U CPU @ 1.80GHz (4 CPUs), ~1.8GHz
- RAM : 4 GB
- Operating System: Ubuntu 16.04 LTS

## 8.2 GPU Specifications

- NVIDIA GeForce740M GPU
- Driver Version : 375.39
- CUDA Cores : 384
- Total Memory : 1 GB

## 8.3 CUDA Specifications

- CUDA Version : 8.0

# 9. APPENDIX C – SOLUTION CODES

## 9.1 Add two vectors of equal sizes

```
/* CUDA program to add two vectors*/
#include <stdio.h>
#include<iostream>
#include <sys/time.h>
#include <time.h>
#include <cuda_runtime.h>
using namespace std;

__global__ void VecAdd(const int* A,const int* B,int* C,int N)
{
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   if(i<N)
          C[i] = A[i] + B[i];
}
timeval t0,t1;
int main()
{
          int N;
          cin>>N;
          int A[N],B[N],C[N],i;
          cudaError_t err = cudaSuccess;
          size_t size = N * sizeof(int);

          for(i=0;i<N;i++){
                    A[i]=(rand() % 10);
                    B[i]=(rand() % 10);
          }
           /*for (int i = 0; i < N; ++i)
                    printf("%d ",A[i]);
          printf("\n");
          for (int i = 0; i < N; ++i)
                    printf("%d ",B[i]);
          printf("\n");*/
          // Allocate the device input vector A
           int *d_A = NULL;
          err = cudaMalloc((void **)&d_A, size);
           if (err != cudaSuccess)
          {
```

```
                fprintf(stderr, "Failed to allocate device vector A (error code %s)!\n",
                cudaGetErrorString(err));
                exit(EXIT_FAILURE);
        }
        // Allocate the device input vector B
        int *d_B = NULL;
        err = cudaMalloc((void **)&d_B, size);
        if (err != cudaSuccess)
        {
                 fprintf(stderr, "Failed to allocate device vector B (error code %s)!\n",
                cudaGetErrorString(err));
                 exit(EXIT_FAILURE);
        }

        // Allocate the device output vector C
        int *d_C = NULL;
        err = cudaMalloc((void **)&d_C, size);
         if (err != cudaSuccess)
        {
                fprintf(stderr, "Failed to allocate device vector C (error code %s)!\n",
        cudaGetErrorString(err));
                exit(EXIT_FAILURE);
        }
        // Copy the host input vectors A and B in host memory to the device input
        // vectors in device memory
        err = cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
if (err != cudaSuccess)
        {
                fprintf(stderr, "Failed to copy vector A from host to device (error code %s)!\n",
cudaGetErrorString(err));
                exit(EXIT_FAILURE);
        }
        err = cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
        if (err != cudaSuccess)
        {
                fprintf(stderr, "Failed to copy vector B from host to device (error code %s)!\n",
cudaGetErrorString(err));
                exit(EXIT_FAILURE);
         }
        int threadsPerBlock = 256;
        int blocksPerGrid =(N + threadsPerBlock - 1) / threadsPerBlock;
        gettimeofday(&t0, 0);

        // Kernel invocation with N threads
        VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
```

```
	gettimeofday(&t1, 0);
	double elapsed = (t1.tv_sec-t0.tv_sec)*1000000+ t1.tv_usec-t0.tv_usec;
	elapsed/=1000000;
	printf("Time elapsed for array of size %d is %f seconds.\n",N,elapsed);
		err = cudaGetLastError();
	if (err != cudaSuccess)
	{
		fprintf(stderr, "Failed to launch vectorAdd kernel (error code %s)!\n",
cudaGetErrorString(err));
		 exit(EXIT_FAILURE);
	}
	printf("Copy output data from the CUDA device to the host memory\n");
	err = cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
	if (err != cudaSuccess)
	{
		fprintf(stderr, "Failed to copy vector C from device to host (error code %s)!\n",
cudaGetErrorString(err));
		 exit(EXIT_FAILURE);
	}
	/*for(i=0;i<N;i++)
		printf("%d ",C[i]);
	printf("\n");*/
	cudaFree(d_A);
	cudaFree(d_B);
	cudaFree(d_B);
}
```

## 9.2 Matrix Multiplication without shared memory

```
/* CUDA program to multiply two matrices*/
#include <stdio.h>
#include <assert.h>
#include <sys/time.h>
#include <time.h>
#include <cuda_runtime.h>

#define BLOCK_SIZE 16

typedef struct {
	int width;
	int height;
	int* elements;
} Matrix;

__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);
```

```
timeval t0,t1;
void MatMul(const Matrix A, const Matrix B, Matrix C) {

        // Allocate A in device memory
        Matrix d_A;
        d_A.width = A.width;
        d_A.height = A.height;
        size_t size = A.width * A.height * sizeof(int);
        cudaError_t err = cudaMalloc(&d_A.elements, size);
        err = cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);

        // Allocate B in device memory
        Matrix d_B;
        d_B.width = B.width;
        d_B.height = B.height;
        size = B.width * B.height * sizeof(int);
        err = cudaMalloc(&d_B.elements, size);
        err = cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);

        // Allocate C in device memory
        Matrix d_C;
        d_C.width = C.width;
        d_C.height = C.height;
        size = C.width * C.height * sizeof(int);
        err = cudaMalloc(&d_C.elements, size);

        // Invoke kernel
        dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
        dim3 dimGrid((B.width + dimBlock.x - 1) / dimBlock.x,
        (A.height + dimBlock.y - 1) / dimBlock.y);
        MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

        err = cudaThreadSynchronize();

        // Read C from device memory
        err = cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);

        // Free device memory
        cudaFree(d_A.elements);
        cudaFree(d_B.elements);
        cudaFree(d_C.elements);

}
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
```

```
	// Each thread computes one element of C by accumulating results into Cvalue
	int Cvalue = 0;
	int row = blockIdx.y * blockDim.y + threadIdx.y;
	int col = blockIdx.x * blockDim.x + threadIdx.x;
	if(row > A.height || col > B.width) return;

	for (int e = 0; e < A.width; ++e)
	Cvalue += (A.elements[row * A.width + e]) * (B.elements[e * B.width + col]);
	C.elements[row * C.width + col] = Cvalue;
}

int main(int argc, char* argv[])
{
	Matrix A, B, C;
	int a;
	a = atoi(argv[1]);
	A.height = A.width = a;
	A.elements = (int*)malloc(A.width * A.height * sizeof(int));
	B.height = B.width = a;
	B.elements = (int*)malloc(B.width * B.height * sizeof(int));
	C.height = C.width = a;
	C.elements = (int*)malloc(C.width * C.height * sizeof(int));
	for(int i = 0; i < A.height; i++)
		for(int j = 0; j < A.width; j++)
			A.elements[i*A.width + j] = (rand() % 10);
	for(int i = 0; i < B.height; i++)
		for(int j = 0; j < B.width; j++)
			B.elements[i*B.width + j] = (rand() % 10);
	gettimeofday(&t0, 0);
	MatMul(A, B, C);
	gettimeofday(&t1, 0);
	double elapsed = (t1.tv_sec-t0.tv_sec)*1000000+ t1.tv_usec-t0.tv_usec;
	elapsed/=1000000;

	printf("Time elapsed for matrix of size %d*%d is %f seconds.\n",a,a,elapsed);
	/*printf("Matrix A\n==========\n");
	for(int i = 0; i < A.height; i++){
		for(int j = 0; j <A.width; j++)
			printf("%d ", A.elements[i*A.width + j]);
		printf("\n");
	}
	printf("\n");
	printf("Matrix B\n==========\n");
	for(int i = 0; i < B.height; i++){
		for(int j = 0; j <  B.width; j++)
```

```
                        printf("%d ", B.elements[i*B.width + j]);
                printf("\n");
        }
        printf("\n");
        printf("Matrix C\n==========\n");
        for(int i = 0; i < C.height; i++){
        for(int j = 0; j < C.width; j++)
        printf("%d\t", C.elements[i*C.width + j]);
        printf("\n");
        }
        printf("\n");*/

}
```

## 9.3 Matrix Multiplication using shared memory

```
/* CUDA program to multiply two matrices using shared memory*/
#include <stdio.h>
#include <assert.h>
#include <sys/time.h>
#include <time.h>
#include <cuda_runtime.h>

#define BLOCK_SIZE 16

typedef struct {
        int width;
        int height;
        float* elements;
        int stride;
} Matrix;

__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);
timeval t0,t1;

void MatMul(const Matrix A, const Matrix B, Matrix C) {
        // Allocate A in device memory
        Matrix d_A;
        d_A.width = d_A.stride = A.width;
        d_A.height = A.height;
        size_t size = A.width * A.height * sizeof(float);
        cudaError_t err = cudaMalloc(&d_A.elements, size);
        cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);
        // Allocate B in device memory
```

```
        Matrix d_B;
        d_B.width = d_B.stride = B.width;
        d_B.height = B.height;
        size = B.width * B.height * sizeof(float);
        err = cudaMalloc(&d_B.elements, size);
        cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);

        // Allocate C in device memory
        Matrix d_C;
        d_C.width = d_C.stride = C.width;
        d_C.height = C.height;
        size = C.width * C.height * sizeof(float);
        err = cudaMalloc(&d_C.elements, size);

        // Invoke kernel
        dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
        dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
        MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
        err = cudaThreadSynchronize();

        // Read C from device memory
        err = cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);

        // Free device memory
        cudaFree(d_A.elements);
        cudaFree(d_B.elements);
        cudaFree(d_C.elements);
}

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col) {
        return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col, float value) {
        A.elements[row * A.stride + col] = value;
}




__device__ Matrix GetSubMatrix(Matrix A, int row, int col) {
        Matrix Asub;
        Asub.width = BLOCK_SIZE;
```

```
	Asub.height = BLOCK_SIZE;
	Asub.stride = A.stride;
	Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row + BLOCK_SIZE * col];
	return Asub;
}

// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
	int blockRow = blockIdx.y;
	int blockCol = blockIdx.x;
	Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
	float Cvalue = 0.0;
	int row = threadIdx.y;
	int col = threadIdx.x;
	for(int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
		Matrix Asub = GetSubMatrix(A, blockRow, m);
		Matrix Bsub = GetSubMatrix(B, m, blockCol);
		__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
		__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
		As[row][col] = GetElement(Asub, row, col);
		Bs[row][col] = GetElement(Bsub, row, col);
		__syncthreads();
		for (int e = 0; e < BLOCK_SIZE; ++e)
			Cvalue += As[row][e] * Bs[e][col];
		__syncthreads();
	}
	SetElement(Csub, row, col, Cvalue);
}

int main(int argc, char* argv[])
{
	Matrix A, B, C;
	int a;
	a = atoi(argv[1]);
	A.height = A.width = a;
	A.elements = (float*)malloc(A.width * A.height * sizeof(float));
	B.height = B.width = a;
	B.elements = (float*)malloc(B.width * B.height * sizeof(float));
	C.height = C.width = a;
	C.elements = (float*)malloc(C.width * C.height * sizeof(float));

	for(int i = 0; i < A.height; i++)
		for(int j = 0; j < A.width; j++)
			A.elements[i*A.width + j] = (rand() % 10);
```

```
	for(int i = 0; i < B.height; i++)
		for(int j = 0; j < B.width; j++)
			B.elements[i*B.width + j] = (rand() % 10);

	gettimeofday(&t0, 0);
	MatMul(A, B, C);
	gettimeofday(&t1, 0);
	double elapsed = (t1.tv_sec-t0.tv_sec)*1000000+ t1.tv_usec-t0.tv_usec;
	elapsed/=1000000;
	printf("Time elapsed for matrix of size %d*%d is %f seconds.\n",a,a,elapsed);/*

	for (int i = 0; i < A.height ; i++){
		for(int j = 0; j <  A.width; j++)
			printf("%f ", A.elements[i*A.width + j]);
		printf("\n");
	}
	printf("\n");
	for(int i = 0; i < B.height; i++){
		for(int j = 0; j <  B.width; j++)
			printf("%f ", B.elements[i*B.width + j]);
		printf("\n");
	}
	printf("\n");
	for(int i = 0; i <  C.height; i++){
		for(int j = 0; j <C.width; j++)
			printf("%f ", C.elements[i*C.width + j]);
		printf("\n");
	}
	printf("\n");*/

}
```