Assignment No.: B7

Roll No.

- **Title :**
  To implement matrix multiplication in CUDA using shared memory

- **Problem Definition :**

  Implement a n x n matrix parallel multiplication using CUDA, use shared memory.

- **Learning Objective :**

  To study matrix multiplication in CUDA using shared memory.

- **Learning Outcome :**

  Successfully implemented matrix multiplication in CUDA using shared memory.

- **Software and Hardware Requirement:**
  - Latest version of 64 Bit Operating Systems Open Source Ubuntu 14.04
  - Multicore CPU equivalent to Intel i5/7 4^{th} generation
  - CUDA enabled machine

- **Theory :**

  1. Introduction

  CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce. CUDA gives developers direct access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs.

  Using CUDA, the GPUs can be used for general purpose processing (i.e., not exclusively graphics); this approach is known as GPGPU. Unlike CPUs, however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly.

2. Related Concepts for Matrix Multiplication

Threads in CUDA each have their own program counter and registers. All threads share a memory address space called "global memory,", and threads within the same block share access to a very fast "shared memory" that is more limited in size. Within the same block threads share the instruction stream and execute instructions in parallel. When thread execution diverges, then the different branches of execution are run serially, until the divergent section has completed, at which point all threads in the block execute in parallel again, until the next divergence within that block. CUDA devices run many threads simultaneously.

3. Concept of programming language use

Threads running under CUDA must be grouped into blocks, and a block can hold at most 512 or 1024 threads. We would thus have to launch multiple blocks to compute the product in the previous example. The phrase compute capability is the term NVIDIA uses to describe the general computing power of its GPUs. The blocks themselves form a two-dimensional grid of blocks.

Kernel functions are specified by declaring them global in the code, and a special syntax is used in the code to launch these functions on the GPU, while specifying the block and grid dimensions. These kernel functions serve as the entry points for the GPU computation, much the same way main() serves as the entry point in an ordinary C program. We launch the global function MatMulKernel on the device, passing parameters d_A, d_B, d_C. The variables dimGrid and dimBlock are of type dim3, which holds a triple of unsigned integers. These are the variables to which the threads have access as described above.

MatMulKernel<<<>>>(d_A, d_B, d_C)

When a kernel is launched on the GPU a grid of thread blocks is created and the blocks are queued to be run on the GPU's multiprocessors. These blocks are given to multiprocessors as they become available, and once started on a multiprocessor, the threads of that block will run to completion on that multiprocessor.

When sharing data between threads, we need to be careful to avoid race conditions, because while threads in a block run logically in parallel, not all threads can execute physically at the same time. Let's say that two threads A and B each load a data element from global memory and store it to shared memory. Then, thread A wants to read B's element from shared memory, and vice versa. Let's assume that A and B are threads in two different warps. If B has not finished writing its element before A tries to read it, we have a race condition, which can lead to undefined behavior and incorrect results.

To ensure correct results when parallel threads cooperate, we must synchronize the threads. CUDA provides a simple barrier synchronization primitive, syncthreads(). A thread's execution can only proceed past a syncthreads() after all threads in its block have executed the syncthreads(). Thus, we can avoid the race condition described above by calling syncthreads() after the store to shared memory and before any threads load from shared memory. It's important to be aware that calling syncthreads() in divergent code is undefined and can lead to deadlock-all threads within a thread block must call syncthreads() at the same point.

4. Algorithm:

1. Include all the header files required for the program.
2. Define value of tile width and matrix width.
3. Write kernel function for multiplication.
4. Take shared array to break the matrices in tile width and fetch them in that array per element.
5. Calculate thread ID.
6. Determine row and column of each element and perform corresponding multiplication.
7. Add all intermediate results to get final value.
8. Allocate memory on host and device using cudaMalloc
9. Initialize values of both input matrices.
10. Copy host array to device array using cudaMemcpy.
11. Call the kernel function.
12. Free the memory on host and device.

5. Flowcharts:

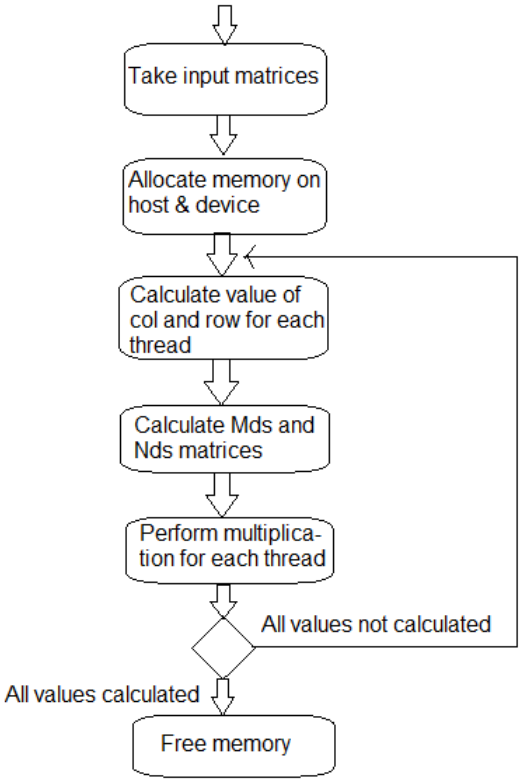

Figure 1: Flow chart

6. Mathematical model:

Aim:

Let system 'S' be the solution for the calculation of matrix multiplication

Initialisation:

Data members:Shared memory.
System Calls=shmget(),shmat(),shmdt().
Input=2 digit number.
Output=Square of number.

Mathematical model using Set theory:

$$L = \{(s, e, i, o, f, DD, NDD, success, failure)\}$$

s= Initial state
e= End of state
i=Input to the system={x | Input matrices Md and Nd }
o=Output given by system={y| Resultant matrix Pd }
f=Transition function

DD=deterministic data
NDD=Non deterministic data
success=Desired outcome generated i.e. Correct resultant matrix obtained
failure=Desired outcome not generated or forced exit due to system error.

Calculation:

$Result = \{(1^{st}\, element\, of\, row\, in\, Md * 1^{st}\, element\, of\, column\, in\, Nd) * (2^{nd}\, element\, of\, row\, in\, Md * 2^{nd}\, element\, of\, column\, in\, Nd)\}$

Venn diagram:

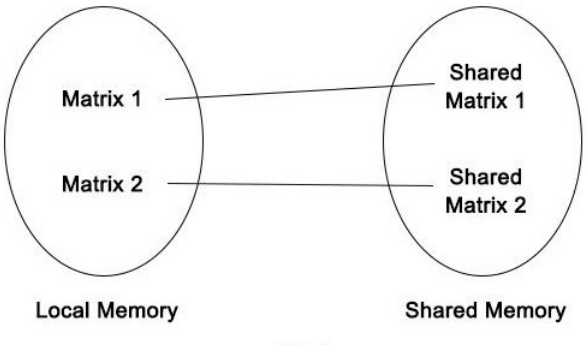

Figure 2: Venn diagram 1

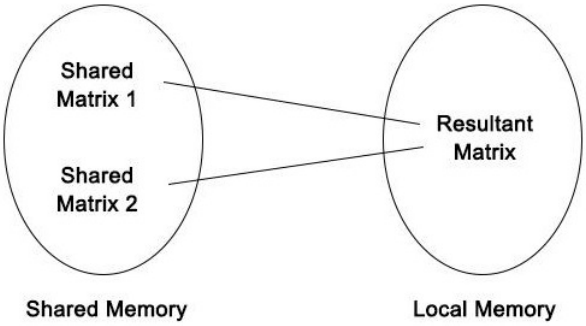

Figure 3: Venn diagram 2

8. UML Diagrams:

Use case Diagram:

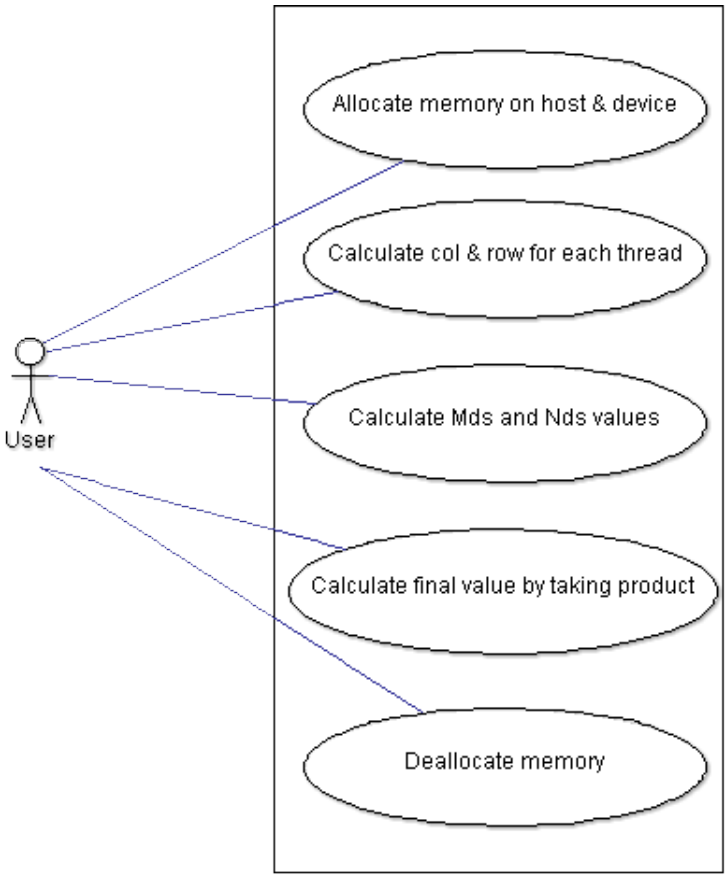

Figure 4: Use case diagram

Sequence Diagram:

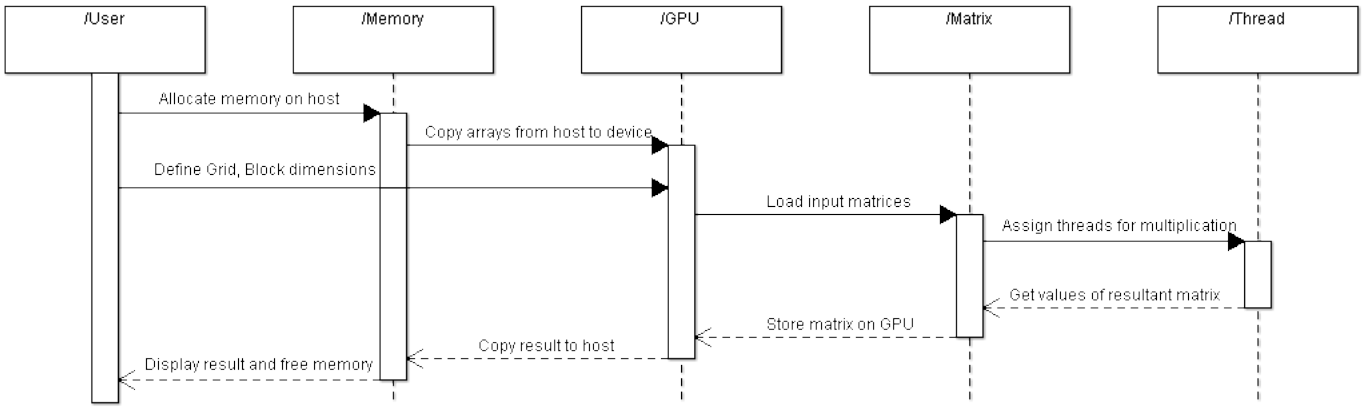

Figure 5: Sequence diagram

Activity Diagram:

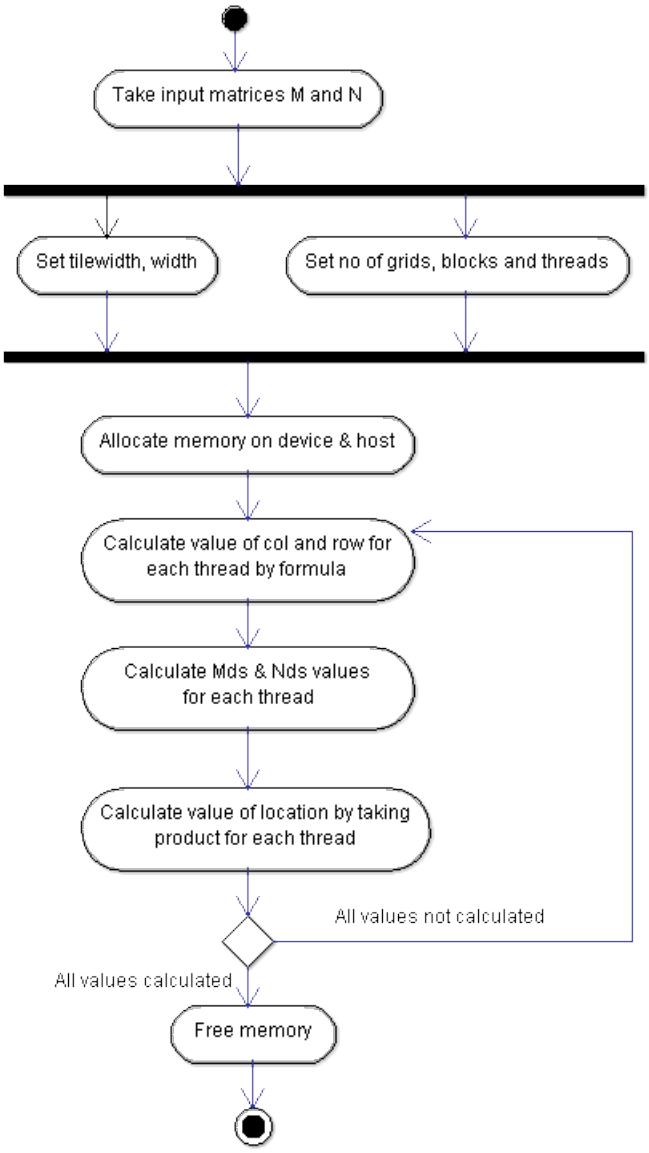

Figure 6: Acitivity diagram

Class Diagram:

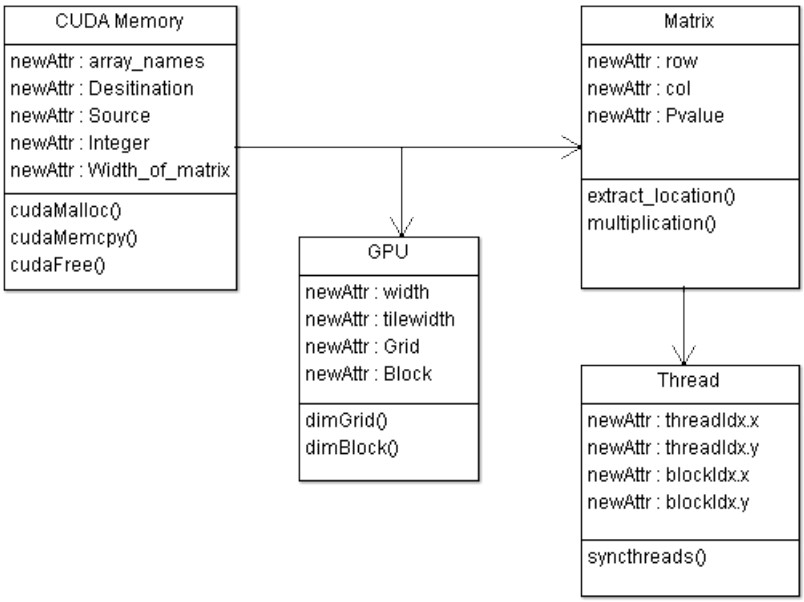

Figure 7: Class diagram

- **Input:**
  Input to this program is are 2 6x6 matrices.
- **Output:**
  Output of this program is a 6x6 matrix.
- **Conclusion :**
  Hence we have studied matrix multiplication in CUDA using shared memory.

Course Outcomes	Achieved Outcome
CO I : Ability to perform multi-core, Concurrent and Distributed Programming.	√
CO II : Ability to perform Embedded Operating Systems Programming using Beaglebone	
CO III :Ability to write Software Engineering Document.	√
CO IV :Ability to perform Concurrent Programming using GPU.	√

FAQ:

- What is shared memory?
- What is CUDA?
- What is kernel function for this problem statement?
- Explain logic of matrix multiplication.
- How CPU allocates memory to variables?
- What is difference between CPU and GPU?