High Performance Scientific Computing
ME 766 : Homework 3 (CUDA)

183079026 - Sandesh Goyal

April 11, 2021

1 Platform

1.1 HOST

Model Name: Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
CPU(s): 40
RAM: 32 GB
OS: CentOS Linux 7 (Core), 64-bit

1.2 Device

— General Information for device 0 —
Name: Quadro K2200
Compute capability: 5.0
— Memory Information for device 0 —
Total global mem: 4239785984
Total constant Mem: 65536
— MP Information for device 0 —
Multiprocessor count: 5
Max threads per block: 1024
Max thread dimensions: (1024, 1024, 64)
Max grid dimensions: (2147483647, 65535, 65535)

nvcc: NVIDIA (R) Cuda compiler driver, Cuda compilation tools, release 8.0, V8.0.61

2 Matrix Multiplication using CUDA C

For Matrix Multiplication using CUDA C, threads are organized in 2D fashion in block which computes a small section(sub-part) of C (result) matrix. Similarly the blocks containing threads are also arranged in a 2D fashion in a grid to cover whole matrix and all together compute the whole C matrix.

For the device platform used, maximum number of threads that can be used per block is 1024, since the square matrix to be computed is of size 100, 1000, 5000, 10000, the block dimension used is 25X25 threads, so that the number threads used in a block do not exceed upper limit of 1024. Also the matrix sizes are exact multiples of 25, therefore the number of blocks required to cover the complete matrix evaluates to proper integer value.

For memory allocation, both at host and device, single dimension (of N*N) or linear allocation is used to store the matrix in row major-wise. Allocation (dynamic using pointers) of 2D array for matrix at host creates array index problem because of pointers when getting converted to 1D at device or vice-versa when using cudamemcpy resulting in segmentation fault. To avoid this issue, 2D allocation is not used at host.

Two versions using CUDA are implemented and their results are included below. Computed matrix C is checked for correctness for smaller N value, print_mat() in the code can be used to display the result in terminal/console.

2.1 Version v1.0 (un-optimized, shared memory not used in device)

Table 1: Timing Analysis for v1.0

Iteration/N	100	1000	5000	10000
1	0.698	0.743	5.958	50.906
2	0.662	0.733	5.605	50.443
3	0.635	0.717	5.671	50.639
4	0.650	0.733	6.058	50.592
5	0.634	0.768	6.018	50.377
Average Computation Time(sec)	**0.6558**	**0.7388**	**5.8620**	**50.5914**

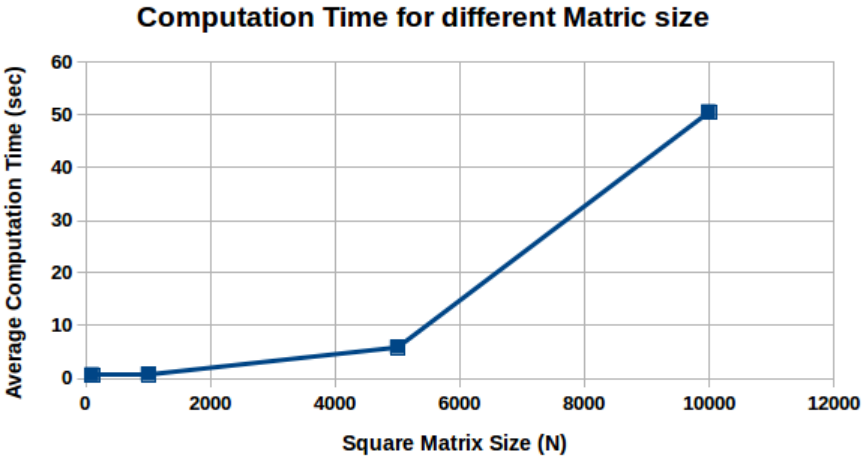

Figure 1: Computation vs Matrix size for v1.0

Results for v1.0: Above data for timing analysis shows that for smaller matrix size, computation time remains almost constant because the resources required to do completely parallel computation are available in device and memory is not bottleneck but as the matrix size increases to higher order, computation time also increases in the same order because the required resources for complete unrolling might not be available in the device and the memory bandwidth also becomes the bottleneck.

2.2 Version v2.0 (optimized using shared memory in device)

In previous implementation, after host data is copied into the global memory of device, for every iteration by each thread, data for A and B is directly fetched from global memory of device which leads to memory bandwidth bottleneck. Also there is a lot of redundancy in the data fetched from global memory by each thread since same location of A and B is fetched by multiple threads. To avoid this redundancy, shared memory is used which is shared among all the threads in a block and since this is local memory, read/write is very fast as compared to the global memory.

Shared memory of dimension same as that of a block is defined both for A and B, all the threads in the block access only a piece/part of A and B and store them in the shared memory. After all the threads complete this step(checked using syncthreads()), computation is performed and the final result is stored back into the global memory.

Table 2: Timing Analysis for v2.0

Iteration/N	100	1000	5000	10000
1	0.802	0.813	2.669	15.385
2	0.681	0.665	2.552	15.296
3	0.633	0.667	2.584	15.429
4	0.634	0.667	2.486	15.296
5	0.633	0.667	2.485	15.061
Average Computation Time(sec)	**0.6766**	**0.6958**	**2.5552**	**15.2880**

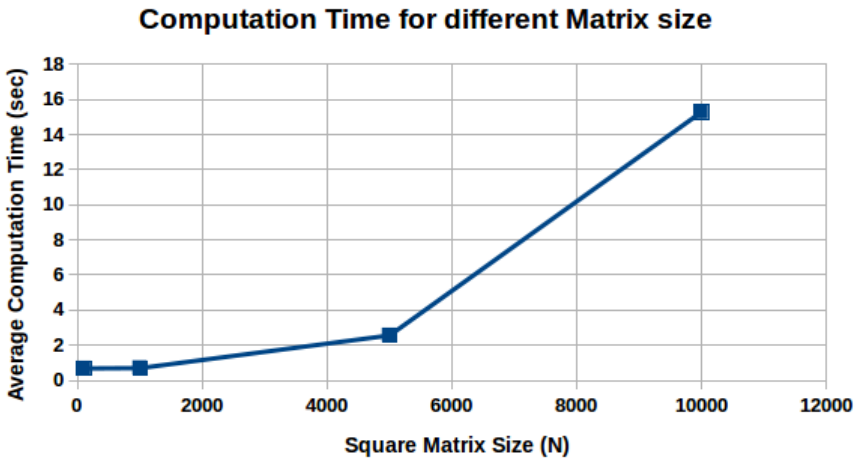

Figure 2: Computation vs Matrix size for v2.0

Table 3: Speedup using MPI

Matrix Size (N)	Speedup
100	0.97
1000	1.06
5000	2.29
10000	3.31

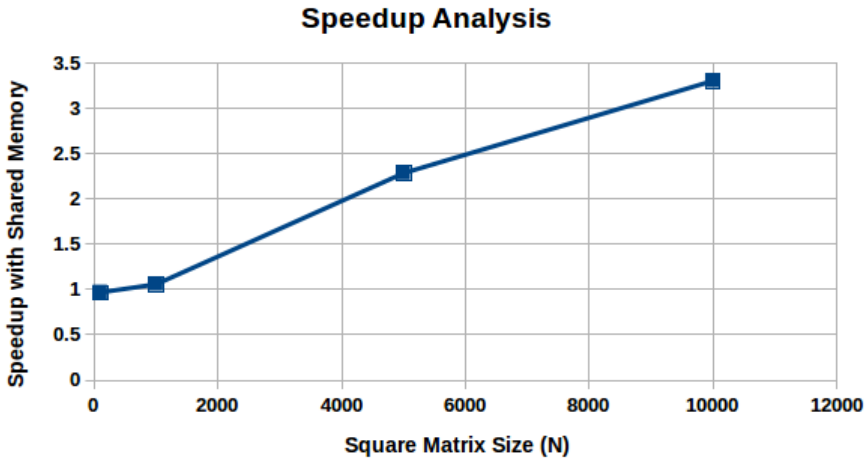

Figure 3: Speedup vs Matrix size using shared memory

Results for v2.0: Above data for timing analysis and speedup shows that compared to un-optimized version, optimized version using shared memory performs much better in terms of computation time. Also for very small matrix size(N=100), performance is a little bit degraded when using shared memory because the overhead added because due to shared memory is more compared the gain obtained by minimizing the global memory fetch for smaller matrix, therefore making use of shared memory is profitable only for large size matrix.

3 Submission

- Sandesh_183079026_HW3.zip
  - v1
    - cuda_mult_v1.cu
    - cuda_mult_v1.sh
    - N_100_cuda_v1.txt
    - N_1000_cuda_v1.txt
    - N_5000_cuda_v1.txt
    - N_10000_cuda_v1.txt
    - N_100_prof_v1.txt
    - N_1000_prof_v1.txt
    - N_5000_prof_v1.txt
    - N_10000_prof_v1.txt
  - v2
    - cuda_mult_v2.cu
    - cuda_mult_v2.sh
    - N_100_cuda_v2.txt
    - N_1000_cuda_v2.txt
    - N_5000_cuda_v2.txt
    - N_10000_cuda_v2.txt
    - N_100_prof_v2.txt
    - N_1000_prof_v2.txt
    - N_5000_prof_v2.txt
    - N_10000_prof_v2.txt
  - device_prop.txt
  - host_prop.txt
  - nvcc_version.txt
  - ME766_HW3_REPORT.pdf

The **script(.sh)** file can **compile and run the cuda code**. Value of **N(matrix size) has to be modified/set manually** in **.cu** file before running the script. **"*_cuda_*.txt" files contain the timing data/values** obtained by running the script file on the platform mentioned in the first section. **"*_prof_*.txt"** files contain more detailed profiling information on resource utilization and event percentage.