HPCA Programming Assignment Part B
Checkered Matrix Multiplication in CUDA

Submitted by Mayank Sati

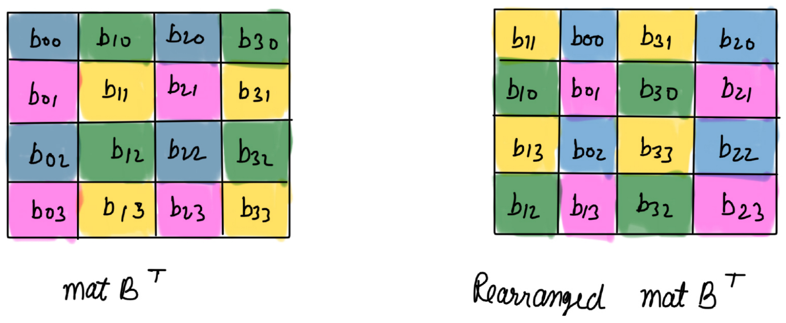
25 November 2021

1 Introduction

Hardware Specification :
CPU: Intel Core i7 7500U Processor 2 Core / 4 Threads, base frequency 2.70GHz andMax Turbo Frequency 3.50 GHz. Physical Memory: 8GB. L1 Cache Size 48 KB, 12-way Set Associative.L2 Cache size 512KB 8-way Set Associative. L3 Cache size 8MB, 16-way Set Associative.
GPU: Graphics Processor: NVIDIA GeForce 940MX, CUDA Cores: 384, Total Memory: 2048 MB, Bus Type: PCI Express x4 Gen3. Compute Capability: 5.0
The code is run in the GPU hardware and all the profiling results are obtained using nvprof. The structure of the report is as follows:
Section 2 briefs about how the single threaded CMM program was converted to CUDA programming environment. Section 3 and 4 shows various data reports and observations made regarding the GPU version of CMM and changes in the data on optimisations.

2 Implementation Details

The parts of the reference CMM Code that could run in parallel were identified in Part 1 of the Assignment, and were used as a base to design the GPU version of the same code. In order to efficiently calculate the matrix multiplication, I computed the transpose and did rearrangement of matrix matB (similar to Part 1 of the Assignment) in host(i.e CPU) so that the GPU handles multiplication in Row Major Access of matA and matB.



After rearrangement is done in CPU, the matrix multiplication can be done in kernel easily by computing the result for each cell of output matrix in a single pass of matA and rearranged matB rows in O(N) time. For this, I created device memory using CudaMalloc() and copied the values of matA and rearranged matB into device memory using cudaMemcpy() followed by kernel call to matrixMul().
Next, I tried to vary the number of threads running in each thread block to minimise the execution time and checked for further optimisations.

3 Reports

Execution time of GPU version of CMM with respect to varying number of threads per blocks (without any profiling tool) is reported as:(averaged over 5 trials)

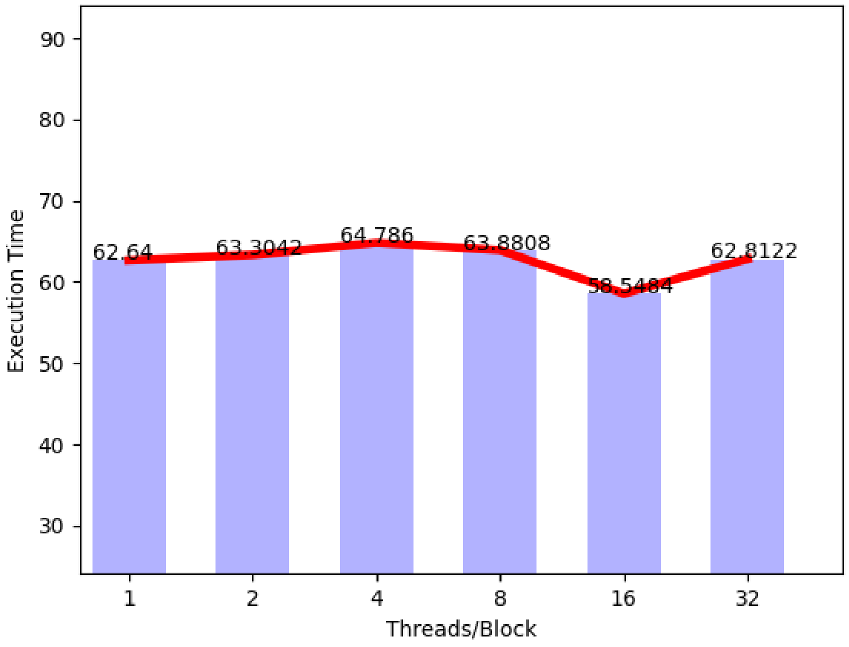


Figure 1: Execution Time(ms) v/s No. of Threads/Block

There is no significant change in execution time with respect to Threads per Blocks. Therefore, following data analysis is done taking Threads/Blocks = 16 (minimum reported).

Time taken by various GPU activities for Threads/Block = 16 using nvprof as a profiling tool are:

	MatrixMul (Kernel Call)	**Memcpy HosttoDevice**	**Memcpy DevicetoHost**
Time(Total)	365.45μs	66.657μs	15.520μs
Time(%)	81.64	14.89	3.47

Time taken by various API calls for Threads/Block = 16 are:

	Time(Total)	**Time(Average)=Time/No. of Calls**	**Time(%)**
cudaMalloc	108.87ms	36.289ms	99.19
cudaMemcpy	522.43μs	174.14μs	0.48
cudaFree	76.585μs	25.528μs	0.07

From above data, it is observed that most of the time is spent on allocating the device memory which is even larger than the kernel execution itself.

Throughputs reported (Threads/Blocks = 16):

Description	**Throughput**
Global Load Throughput	97.749GB/s
Global Store Throughput	10.946GB/s
Device Memory Read Throughput	174.45MB/s
Unified Cache Throughput	43.444GB/s
L2 Throughput (Reads)	97.767GB/s
L2 Throughput (Writes)	10.946GB/s

4 Optimisations in GPU CMM:

Since the code was optimised for single threaded and multi threaded CMM, not much optimisations were made. The only optimisation attempted was loop unrolling in the kernel function.
The two for loops in the kernel function were unrolled 8 times to see the changes. Compared to the unrolled version, the following comparison was made: Execution time (without using any profiler) of unrolled version compared to the original code:

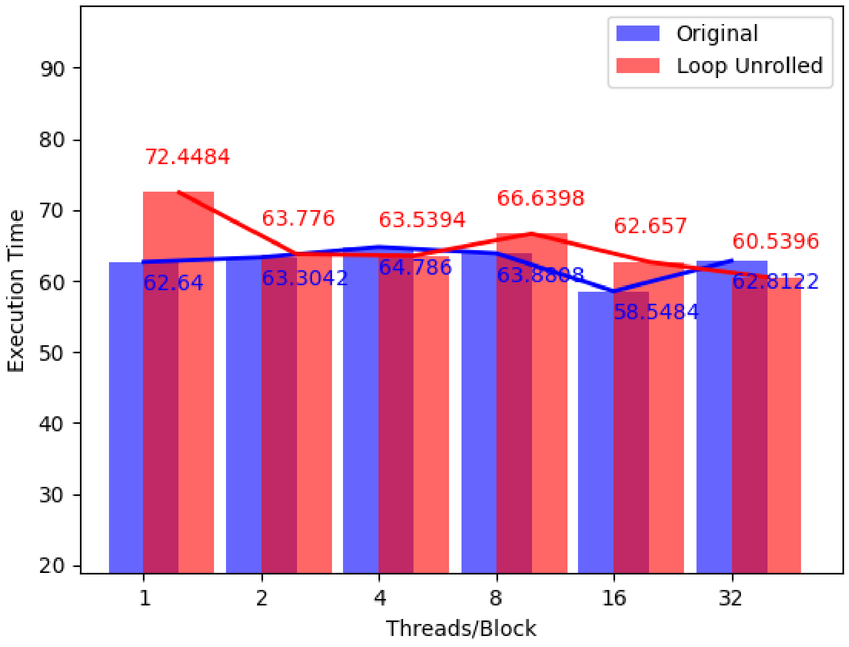


Figure 2: Execution Time(ms) compared to Original and its Loop Unrolled Counterpart

Here, we do not see any significant reduction in execution time compared to original code (Execution time

has infact increased on loop unrolling in some cases). This is due to the fact that loop unrolling causes more work to be done per thread while sacrificing a little in thread parallelism.
This was indeed beneficial in case of CPU bound program as CPU has very fast running cores and we saw speedups of 2 in case of loop unrolling in Assignment 1 for CPU based CMM program. But GPU exploits the property of parallelism to amortise the slower single-thread performance to achieve greater throughput. So loop unrolling does not help reduce the execution time and infact increases it.

5 Conclusion

GPU version of CMM allowed us to get familiar with CUDA programming and we learnt how a multi-threaded program can be made compatible to execute in a GPU. Architectural details of GPUs were studied that helped in analysing GPU CMM code. We saw how the data varies on varying number of threads and how certain optimisations which were helpful in CPU can have adverse affects on GPU environment.

6 References

[1] https://developer.nvidia.com/blog/even-easier-introduction-cuda/
[2] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[3] https://docs.nvidia.com/cuda/profiler-users-guide/index.html
[4] https://developer.nvidia.com/blog/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler
[5] https://matplotlib.org/stable/users/index.html