Shahid Beheshti University

Faculty of Mathematical Science

Department of Computer Science

Concurrent Programming – Spring 2022 – Assignment 5 – Simple Matrix Multiplication implementation with CUDA

By:

Arman Davoodi

Introduction

In this assignment, Matrix multiplication Algorithm is written in C++ language in both parallel and sequential format. The purpose of this assignment is to parallelize this algorithm on GPU using CUDA.

The results given in this report are based on several runs on a system with GeForce GTX 1050 and Windows 10 operating system and the CUDA version is 11.6.0.

Simple Matrix Multiplication

The brute force method to compute the dot product of two matrices is to implement the formula illustrated in *Equation 1* for all elements of C. *Algorithm 1* demonstrates the pseudo-code for this implementation.

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \,.\, B_{kj}$$

Equation 1

Algorithm 1: Naïve Matrix Multiplication Algorithm(A[N][K], B[K][M]):

Define Matrix C of size NM.

```
1_  BEGIN
2_     for i from 0 to N do
3_        for j from 0 to M do
4_           C[i][j] = 0
5_           for k from 0 to K do
6_              C[i][j] += A[i][k] * B[k][j]
7_           endfor
```

8_ **endfor**

9_ **endfor**

10_ **return** C

11_ **END**

It is clear that in *Algorithm 1* the instruction at line 6 is repeated N*M*K times. Therefore using this algorithm on two n by n matrices has a time complexity of $O(n^3)$.

Parallel Implementations

For the parallel implementation of the algorithm, two dimensional blocks are used where in each block threads are also organized in a two dimensional space. Also, for this assignment two different kernels were used. The first kernel which is called the simple kernel in this report, computes a single element of the result matrix where the element's index is determined by the global index of the thread running the kernel. The second kernel uses tiling as an approach to increase the efficiency of the algorithm. In this implementation, threads of each block will first copy a part of the input matrices into their shared memory and then they do a part of computation by using the data loaded into their shared memory. Just like the simple kernel, in this approach each thread computes a single element of the resulting matrix. A more detailed explanation of the tiling approach is explained in the link provided in the resources.

Comparing Sequential and Parallel Implementations

In the host code of both parallel implementations, each block had a 16 by 16 dimension. All of the tested matrices were square matrices. The time in seconds for all runs and also the speedup for parallel implementations are presented in *Table 1* below.

Table 1

	Size				
Algorithm	128	256	512	1024	2048
Sequential	0.007960(s)	0.079373(s)	0.762961(s)	6.749335(s)	229.432862(s)
Parallel simple kernel	0.000995(s) 8(speedup)	0.007684(s) 10.330(speedup)	0.049508(s) 15.414(speedup)	0.374292(s) 18.032(speedup)	2.969001(s) 77.276(speedup)
Parallel tiling	0.000915(s) 8.699(speedup)	0.006404(s) 12.394(speedup)	0.042410(s) 17.990(speedup)	0.313866(s) 21.504(speedup)	2.492375(s) 92.054(speedup)

Same Algorithm on another System

Since both speedups and computation times of parallel implementations were low considering the specified system, the same algorithms were run on matrices of size 4096 by 4096 on the same system and another system which we referee to as system1 and system2 respectively. The second system had GeForce mx250 and a 5.15.32-1-MANJARO Linux operating system. The CUDA version of system2 was the same as system1. The results are shown in *Tabel 2* below.

Table 2

	System1	System2
Sequential	1149.159986(s)	-
Parallel simple kernel	24.023849(s)	2.919419(s)
Parallel tiling	20.051132(s)	1.073667(s)

Even though the GPU on system1 is stronger compared to system2, we can see a huge difference in performance in these systems as the runtime of the parallel implementation with simple kernel for a matrix of size 4096 by 4096 in system2 is lower than the runtime of the same algorithm for system1 for matrices of size 2048 by 2048. The reason for this phenomenon is unknown. However, we can conclude that these algorithms can give us much higher speedups than the ones provided in *Tabel 1*.

Conclusion

In the end, we can see that by parallelizing matrix multiplication on GPU, we can get a huge boost to our speed and by using tiling approach we can parallelize the algorithm on GPU more efficiently.

Refrence

https://penny-xu.github.io/blog/tiled-matrix-multiplication