Shahid Beheshti University

Faculty of Mathematical Science

Department of Computer Science

Concurrent Programming – Spring 2022 – Assignment 6 – Strassen Matrix Multiplication implementation with CUDA

By:

Arman Davoodi

Introduction

In this assignment, Strassen Matrix multiplication Algorithm is written in C++ language in both data-parallel and sequential format; and the performance of both implementations are compared with each other to get the amount of speedup gained by parallelizing this algorithm. The CUDA library is used to parallelize the algorithm.

The results given in this report are based on several runs on a system with Intel Core I7-8550U CPU, GeForce GTX 1050, CUDA version 10.1 and Linux mint version 20 operating system.

Strassen Matrix Multiplication

Another approach is to use the divide and conquer paradigm to present an algorithm capable of computing dot products of two matrices with a time complexity lower than $O(n^3)$.

To simplify the problem, we assume that both matrices are n by n square matrices with n being a power of 2.

With the above assumption we can rewrite our matrices like *Equation 2* in a way that for any i and j from 1 to 2, A_{ij}, B_{ij}, and C_{ij} will be of size n/2 by n/2.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \qquad A.B = C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$C_{11} = A_{11}.B_{11} + A_{12}.B_{21}, \qquad C_{12} = A_{11}.B_{12} + A_{12}.B_{22}$$

$$C_{21} = A_{21}.B_{11} + A_{22}.B_{21}, \qquad C_{22} = A_{21}.B_{12} + A_{22}.B_{22}$$

Equation 1

If matrices A and B are 2 by 2 matrices, matrix C can be computed easily, However if the size of our matrices is 4 or higher we need to use this algorithm.

The first step after dividing the matrices is to construct ten matrices shown in *Equation 3*, then by using these matrices, 7 other matrices are constructed as shown in *Equation 4*. The last step shown in

Equation 5 uses matrices constructed in previous steps to compute matrices C_{11}, C_{12}, C_{21}, and C_{22}.

$$S_1 = B_{12} - B_{22}, \quad S_2 = A_{11} + A_{12}, \quad S_3 = A_{21} + A_{22}, \quad S_4 = B_{21} - B_{11}$$

$$S_5 = A_{11} + A_{22}, \quad S_6 = B_{11} + B_{22}, \quad S_7 = A_{12} - A_{22}, \quad S_8 = B_{21} + B_{22}$$

$$S_9 = A_{11} - A_{21}, \quad S_{10} = B_{11} + B_{12}$$

Equation 2

Since matrix addition and subtraction can be done in $O(n^2)$, the construction of matrices S_1 to S_{10} have a time complexity of $O(n^2)$.

$$P_1 = A_{11}.S_1 = A_{11}.B_{12} - A_{11}.B_{22}, \quad P_2 = S_2.B_{22} = A_{11}.B_{22} + A_{12}.B_{22}$$

$$P_3 = S_3.B_{11} = A_{21}.B_{11} + A_{22}.B_{11}, \quad P_4 = A_{22}.S_4 = A_{22}.B_{21} - A_{22}.B_{11}$$

$$P_5 = S_5.S_6 = A_{11}.B_{11} + A_{11}.B_{22} + A_{22}.B_{11} + A_{22}.B_{22}$$

$$P_6 = S_7.S_8 = A_{12}.B_{21} + A_{12}.B_{22} - A_{22}.B_{21} - A_{22}.B_{22}$$

$$P_7 = S_9.S_{10} = A_{11}.B_{11} + A_{11}.B_{12} - A_{21}.B_{11} - A_{21}.B_{12}$$

Equation 3

Matrices P_1 to P_7 can be computed by 7 recursive calls of this algorithm.

$$C_{11} = P_5 + P_4 - P_2 + P_6, \quad C_{12} = P_1 + P_2, \quad C_{21} = P_3 + P_4, \quad C_{22} = P_5 + P_1 - P_3 - P_7$$

Equation 4

Construction of matrices C_{11}, C_{12}, C_{21}, and C_{22} by using *Equation 5* has a time complexity of $O(n^2)$ due to the usage of matrix addition and subtraction methods.

Therefore the time complexity of Strassen matrix multiplication as shown in *Equation 6* is $O(n^{lg7})$ which is approximately $O(n^{2.81})$ and is lower than $O(n^3)$. This can be computed by using the master method.

$$T(n) = \begin{cases} 1 & n = 1 \\ 7T\left(\frac{n}{2}\right) + \theta(n^2) & n > 1 \end{cases}$$

Equation 5

To compute dot products of non-square matrices or matrices with dimension sizes that are not a power of 2, padding method can be used in which some rows and columns of 0s are appended at the end of input matrices.

Sequential Implementation

To improve the performance of the algorithm, the implementation uses the naïve method to compute the dot product of matrices of size 64 by 64 or lower. This is due to the fact that the Strassen algorithm has a lot of overhead and therefore it is not efficient to use for small matrices. Moreover to reduce the amount of computations, instead of creating new matrices for A_{ij}, B_{ij}, and C_{ij}, starting indexes and matrix sizes are passed to the function.

Parallel Implementation

In parallel implementation, the naïve dot product algorithm, and matrix addition and subtraction methods are implemented as GPU kernels, whereas the rest of the algorithm is implemented sequentially.

Comparison

To compare time efficiency for these implementations, both algorithms were run on 4 matrices of different sizes. The result of each one is illustrated on *Table 1*. Duration is the average time of all of those runs in seconds.

Table 1

row	size		Duration(s)	Speedup
1	512×512	Sequential	0.42813	43.73
		Parallel	0.00979	
2	1024×1024	Sequential	0.29640	4.61
		Parallel	0.06431	
3	2048×2048	Sequential	2.13906	4.75
		Parallel	0.45360	
4	4096×4096	Sequential	152.84215	50.52
		Parallel	3.02546	

Conclusion

In the end, we can conclude that this parallel implementation of the Strassen algorithm can give us up to 50.52 times speedup on large enough matrices on the specified system.