Optimization of Matrix Multiplication

A TBB vs Sequential Approach

Kristi Stefa
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: kstefa@oakland.edu

I. Introduction

The objective of the project is to do an analysis on matrix multiplication algorithms and see performance in different cases between a naïve sequential implementation and a TBB parallel implementation using whatever functions are appropriate. Typical sequential matrix multiplication and Strassen will be the algorithms of focus and the first step is to develop a naïve algorithm for square matrices of any size. The test cases will be grayscale images processed through Matlab to create binary files and the application will return the modified output. The results will be compared numerically against Matlab's native multiplication function and visually using a recognizable image filter as the "B" matrix in the operations. Once validated for the test cases of different sized square images, performance analysis can be done and recorded, and conclusions can be made.

II. Methodology

A. Design

The analysis will revolve around 4 square images that were selected to represent a range of element numbers. Their specifications are listed below:

small	128x128
medium	256x256
large	512x512
very large	1024x1024

Table 1: Image properties

Through the use of a Matlab script, the images can be converted into grayscale and formatted as a binary file for use by the application and then reformatted based on the application's output as another binary file. The reformatted output can be compared against an image created by the Matlab script and the image difference will be displayed.

The input will be used in four ways when it gets to the inversion application: a naïve sequential approach of the "normal" algorithm, a parallel approach of the normal, a sequential version of the Strassen, and a parallel version of the Strassen that leverages the parallel normal. This allows for efficiency comparisons between both parallel methods and a speedup comparison between different versions of the same algorithm. The model for the system can be seen in the figure below

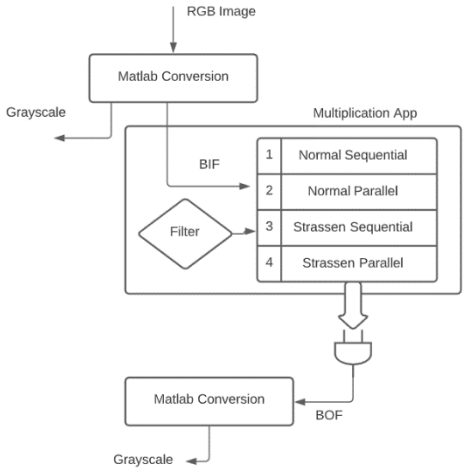

Figure 1: Project Model

B. Multiplication Algorithms

The first algorithm of focus is a normal naïve matrix multiplication, which utilizes a sum counter and goes row by column to compute the new value for each matrix point. Square matrixes with dimensions N x N will have a computation complexity $O(N^3)$ when performing this method, implying that doubling the matrix size will require 8x the computations to achieve and so on.

An alternative method is the Strassen algorithm [1], which begins with splitting all matrices into submatrices. An inherent requirement is that the cases must be of size 2^K.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

Figure 2: Matrix subdivision

The idea of this subdivision is that the matrices can be partitioned to break up a large multiplication into 8 smaller ones, an action that does nothing for the computation complexity. The main bulk of the Strassen algorithm is how it transforms the 8 input submatrices into 7 co-factors, as seen below

$$\mathbf{M}_1 := (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2})$$
$$\mathbf{M}_2 := (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1}$$
$$\mathbf{M}_3 := \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2})$$
$$\mathbf{M}_4 := \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1})$$
$$\mathbf{M}_5 := (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2}$$
$$\mathbf{M}_6 := (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2})$$
$$\mathbf{M}_7 := (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})$$

Figure 3(a): Co-factor Calculation

$$\mathbf{C}_{1,1} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$$
$$\mathbf{C}_{1,2} = \mathbf{M}_3 + \mathbf{M}_5$$
$$\mathbf{C}_{2,1} = \mathbf{M}_2 + \mathbf{M}_4$$
$$\mathbf{C}_{2,2} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$$

Figure 3(b): Output Computation

Looking at the equations listed, the multiplication operation is considerably less straightforward and the potential for improvement over a typical multiplication is not immediately visible. The improvement comes in the computational reduction from 8 sub-multiplications to 7 sub-multiplications and various additions. This reduces the order of complexity from 3 to $\log_2 7$ or $O(N^3)$ vs $O(N^{2.80})$.

The theory behind Strassen is that the operations are done recursively when computing the co-factors, reducing the needed multiplications to be 1 x 1. In practice, this full recursion is never done for anything above size 16 because the amount of co-factors and size needed would increase exponentially [2]. The practical implementation, and the one implemented here, is to divide only once and perform the needed 7 sub-multiplications with a different algorithm that is optimized for N/2 sized submatrices. The "normal" algorithm is leveraged for the subs in this experiment.

C. Implementation/Parallelism

The Strassen algorithm required the development of a custom class that could implement the submatrix functionality required. On construction, the class would assign pointers to the 4 "quadrants" of the linear matrix array and allow access to functions that required it. The class also had a method to merge 4 quadrant vectors together to create an output linear matrix when the Strassen operations were finished. The Strassen function utilizes two temp arrays and seven co-factor arrays for use in addition and for passing to the simple matrix multiplication function. The simple multiplication function takes in two input matrices as linear vectors and does a 3-dimensional for loop based on the size of the vectors that are passed.

The main parallelism strategy was the use of *TBB::parallel_for* over both 3D and 1D blocked ranges[3]. The normal multiplication exists as a lambda function with a 3D blocked range from 0 to passed size. For vector wide addition and pointer assignments, a 1D range was leveraged with *parallel_for* in the Strassen matrix construction and sub-calculations. No 2D ranges were used because they would require a pairing with *parallel_reduce*, a design pattern where the code "density" might outweigh possible improvement over a 3D range.

III. Experimental Setup

The actual application and all of its methods will be ran on the same board while utilizing the Intel TBB library and a Makefile. Performance time will be analyzed by the systime header and the functionality it offers. The four time captures will be performed in the same *main()* for consistency and 10 trials will be taken for each size, barring absurd wait times. Matlab will be utilized on a personal computer to initialize the binary file, generate its own matrix product off the input matrix, and construct a difference matrix between the experimental and simulated values.

IV. Results

The timing results for all cases can be seen in the table:

Size	Seq Norm	Seq Strass	Para Norm	Para Strass
128	37009.1	**12521.7**	26648.1	31785.4
256	279550	219712	212504	**147237**
512	15593915	4379200	2215604	**1477464**
1024	141.2 M	114.2 M	60116914	**12895696**

Table 2: Time results (us)

For a visual look at the data, here is a graph of performance:

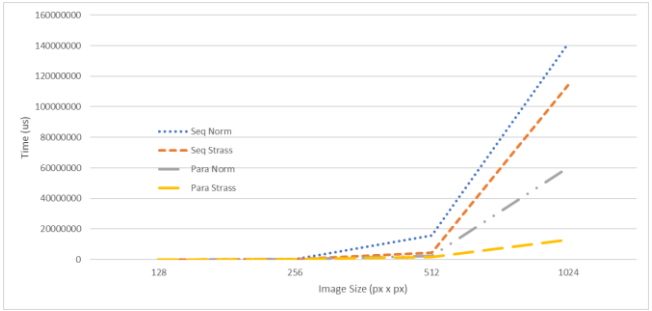

Figure 4: Graph of performance

Looking at a measure of the sequential normal vs the results from parallel Strassen, the following speedup can be observed:

128	1.16x
256	1.89x
512	10.55x
1024	10.94x

Table 3: Parallel Strassen Speedup vs Seq Normal

From these results, it is plain to see that parallel Strassen is the method to use for most cases and it will continue for larger matrix sizes. An interesting datapoint is the 128 x 128 case, where sequential Strassen held the speed advantage by ~2x over anything else. Analysis would conclude that Strassen is faster than normal in both sequential and parallel and it can be shown that parallel loses speed over sequential when sizes are small, namely 64, which is the size of the submatrices in that case.

The results also follow the theoretical computation complexity, in that doubling the matrix size should require 8x the computations. Further analysis could be done with double values for the B and C matrices instead of integers to assess performance, with the caveat that results would look like white noise if the B filter isn't created algorithmically for large sizes.

V. Conclusion

To give an overview of the objective and results, there is merit in trying to optimize matrix multiplication using different algorithms and methods of parallelism. The speedup from Strassen and TBB outweighs the extra development needed to implement them around a naïve approach. There was a clear divide from cases that were too small for parallelism to be effective and cases where parallelism shined, information that's to the benefit of future applications and developers. A rudimentary test case was used for the purposes of easy visualization but the application can be expanded to more involved and engaging filters.

VI. References

[1] Strassen, V. Gaussian elimination is not optimal. *Numer. Math.* **13,** 354–356 (1969). https://doi.org/10.1007/BF02165411

[2] Strassen Algorithm https://en.wikipedia.org/wiki/Strassen_algorithm

[3] TBB blocked range3D, Intel, https://www.threadingbuildingblocks.org/docs/doxygen/a00024.html