Application with parallel programming on Strassen Algorithm

Final Project Report - Team 02

Author,

陳正宗 0756823, Computer Science Department, Chiao Tung University, Hsinchu City, a9311072@gmail.com

謝祥志 0756825, Computer Science Department, Chiao Tung University, Hsinchu City, souleye1983@gmail.com

Abstract

In mathematics, matrix multiplication or matrix product is a binary operation that produces a matrix from two matrices. Matrix multiplication is thus a basic tool of linear algebra, and as such has numerous applications in many areas of mathematics, as well as in applied mathematics, statistics, physics, economics, and engineering. When two linear maps are represented by matrices, then the matrix product represents the composition of the two maps.

However, it may take a lot of time to process if the size of the matrix is large. In this project, we analyze several parallel programming methods for matrix multiplication, trying to compare parallel tools for matrix multiplication and getting better performance.

Introduction

Applications of matrices are found in most scientific fields. In every branch of physics, including classical mechanics, optics, electromagnetism, quantum mechanics, and quantum electrodynamics, they are used to study physical phenomena, such as the motion of rigid bodies. In computer graphics, they are used to manipulate 3D models and project them onto a 2-dimensional screen. In probability theory and statistics, stochastic matrices are used to describe sets of probabilities; for instance, they are used within the PageRank algorithm that ranks the pages in a Google search. Matrix calculus generalizes classical analytical notions such as derivatives and exponentials to higher dimensions. Matrices are used in economics to describe systems of economic relationships.

Volker Strassen first published this algorithm in 1969 and proved that the O (n^3) general matrix multiplication algorithm wasn't optimal. The Strassen algorithm[3][5][15] O ($n^{2.807}$) is only slightly better than that, but its publication resulted in much more research about matrix multiplication that led to faster approaches.

In an application, the matrix multiplication algorithm uses a lot of looping and multiplication operations, which costs lots of time. The computational efficiency of matrix multiplication[1][2][14][16] greatly affects the speed of the entire program, so the matrix multiplication algorithm. Some improvements in parallelization are necessary.

Proposed Solution

Because matrix multiplication[14] is such a central operation in many numerical algorithms, much work has been invested in making matrix multiplication algorithms efficient. Applications of matrix multiplication in computational problems are found in many fields including scientific computing and pattern recognition and in seemingly unrelated problems such as counting the paths through a graph. Many different algorithms have been designed for multiplying matrices on different types of hardware, including parallel and distributed systems, where the computational work is spread over multiple processors.

Directly applying the mathematical definition of matrix multiplication gives an algorithm that takes time on the order of n^3 to multiply two $n \times n$ matrices $O(n^3)$. Better asymptotic bounds on the time required to multiply matrices have been known since the work of Strassen in the 1960s.

Algorithms exist that provide better running times than the straightforward ones. The first to be discovered was Strassen's algorithm, devised by Volker Strassen in 1969 and often referred to as "fast matrix multiplication". It is based on a way of multiplying two 2 × 2-matrices which requires only 7 multiplications, at the expense of several additional addition and subtraction operations.

Strassen's algorithm is more complex than classical matrix multiplication, and the numerical stability is reduced compared to the traditional algorithm, but it is faster in cases where n > 100 or so and appears in several libraries. It is very useful for large matrices over exact domains such as finite fields, where numerical stability is not an issue.

Practical implementations of Strassen's algorithm switch to standard methods of matrix multiplication for small enough submatrices, for which those algorithms are more efficient. Strassen's algorithm is more efficient depends on the specific implementation and hardware. We had estimated that Strassen's algorithm is faster for matrices with widths from 32 to 128 for optimized implementations. However, it has been observed that this crossover point has been increasing in recent years, and a study found that even a single step of Strassen's algorithm is often not beneficial on current architectures, compared to a highly optimized traditional multiplication, until matrix sizes exceed 1000 or more, and even for matrix sizes of several thousand the benefit is typically marginal at best (around 10% or less). A more recent study observed benefits for matrices as small as 512 and a benefit of around 20%.

In practice, Strassen's algorithm can be implemented to attain better performance than conventional multiplication even for small matrices, for matrices that are not at all square, and without requiring workspace beyond buffers that are already needed for a high-performance conventional multiplication.

In the first step, we compare classical matrix multiplication and Strassen algorithm. The Strassen algorithm defines instead new matrices, only using 7 multiplications (one for each Tk) instead of 8. We may now express the M[i][j] in terms of Tk:

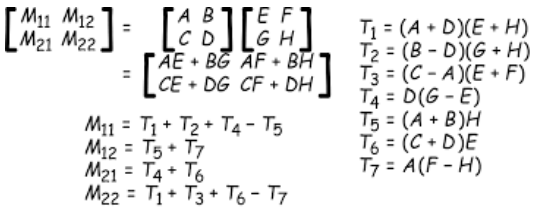


Fig1. Strassen Algorithm in 2*2 matrix

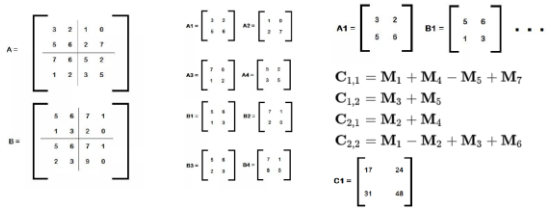


Fig2. Strassen Algorithm in 4*4 matrix

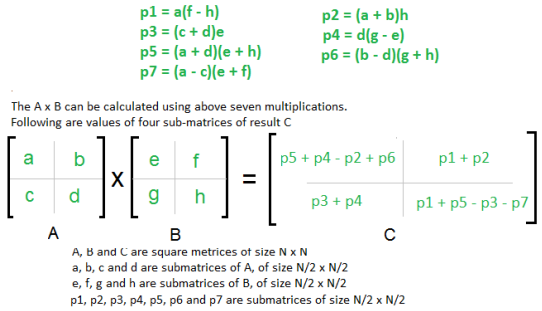


Fig3. Strassen Algorithm in 2*2 matrix

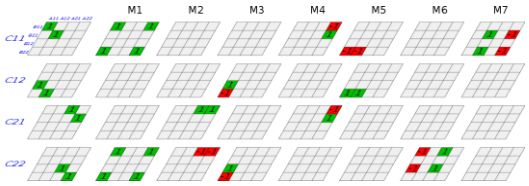


Fig4. Strassen Algorithm

In second step, we decide where to parallel. We found that the addition has little effect on time complexity of the matrix calculation. So, we decided to do parallelization multiplication part in the matrix calculation. It shows in Fig5, Fig6 and Fig7:

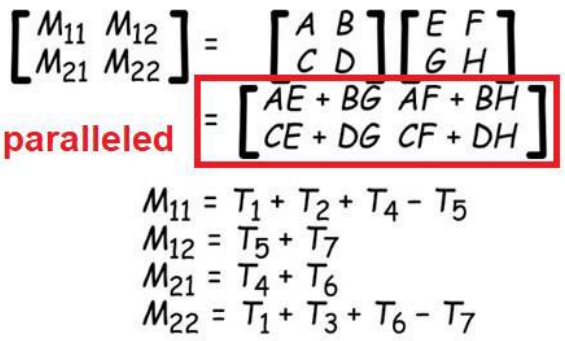


Fig5. Paralleled in Strassen Algorithm

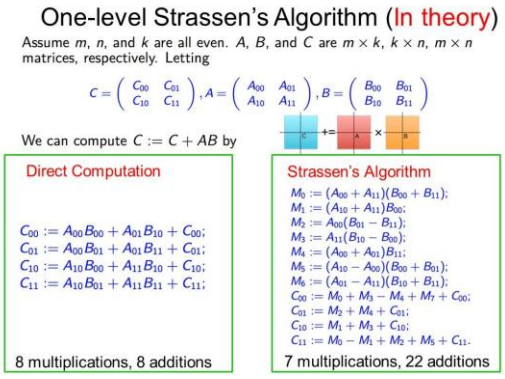


Fig8. The 2*2 matrix multiplication of traditional and Strassen algorithm

MMult(A, B, n)

1. If $n = 1$ Output $A \times B$
2. Else
3. Compute $A^{11}, B^{11}, \ldots, A^{22}, B^{22}$ % by computing $m = n/2$
4. $X_1 \leftarrow MMult(A^{11}, B^{11}, n/2)$
5. $X_2 \leftarrow MMult(A^{12}, B^{21}, n/2)$
6. $X_3 \leftarrow MMult(A^{11}, B^{12}, n/2)$
7. $X_4 \leftarrow MMult(A^{12}, B^{22}, n/2)$
8. $X_5 \leftarrow MMult(A^{21}, B^{11}, n/2)$
9. $X_6 \leftarrow MMult(A^{22}, B^{21}, n/2)$
10. $X_7 \leftarrow MMult(A^{21}, B^{12}, n/2)$
11. $X_8 \leftarrow MMult(A^{22}, B^{22}, n/2)$
12. $C^{11} \leftarrow X_1 + X_2$
13. $C^{12} \leftarrow X_3 + X_4$
14. $C^{21} \leftarrow X_5 + X_6$
15. $C^{22} \leftarrow X_7 + X_8$
16. Output C
17. End If

parallelled

Fig6. Paralleled in Strassen Algorithm

In Fig7 and Fig8, we can find that the classical matrix calculation algorithm has 8 multiplications and 8 additions in the 2*2 matrix, but the Strassen algorithm only needs 7 multiplications and 22 additions.

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time.

Many studies have shown that parallel programming is quite beneficial for matrix multiplication, so we choose different parallelization tools to accomplish and compare efficiency differences.

1. If $n = 1$ Output $A \times B$
2. Else
3. Compute $A^{11}, B^{11}, \ldots, A^{22}, B^{22}$ % by computing $m = n/2$
4. $P_1 \leftarrow Strassen(A^{11}, B^{12} - B^{22})$
5. $P_2 \leftarrow Strassen(A^{11} + A^{12}, B^{22})$
6. $P_3 \leftarrow Strassen(A^{21} + A^{22}, B^{11})$
7. $P_4 \leftarrow Strassen(A^{22}, B^{21} - B^{11})$
8. $P_5 \leftarrow Strassen(A^{11} + A^{22}, B^{11} + B^{22})$
9. $P_6 \leftarrow Strassen(A^{12} - A^{22}, B^{21} + B^{22})$
10. $P_7 \leftarrow Strassen(A^{11} - A^{21}, B^{11} + B^{12})$
11. $C^{11} \leftarrow P_5 + P_4 - P_2 + P_6$
12. $C^{12} \leftarrow P_1 + P_2$
13. $C^{21} \leftarrow P_3 + P_4$
14. $C^{22} \leftarrow P_1 + P_5 - P_3 - P_7$
15. Output C
16. End If

parallelled

Fig7. Paralleled in Strassen Algorithm

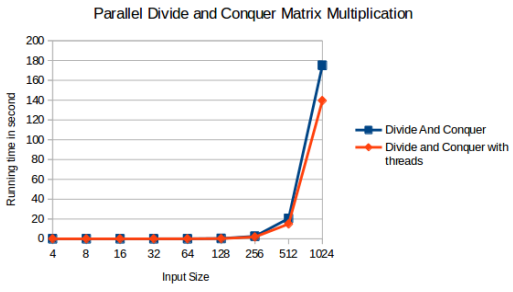


Fig9. Parallel in matrix multiplication

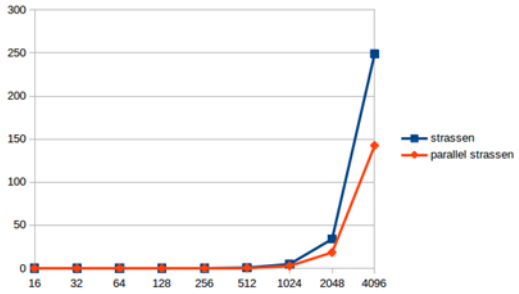


Fig10. Parallel in Strassen algorithm

Pthread[9][10] is an execution model that exists independently from a language, as well as a parallel execution model. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a thread, and creation and control over these flows are achieved by making calls to the POSIX Threads API.

OpenMP[17][18] is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, instruction set architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and MPI, such that OpenMP is used for parallelism within a (multi-core) node while MPI is used for parallelism between nodes. There have also been efforts to run OpenMP on software distributed shared memory systems, to translate OpenMP into MPI and to extend OpenMP for non-shared memory systems.

CUDA[12][13][19] is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled GPU [7][8] for general purpose processing — an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

As the above mentioned, in order to compare which tool is more suitable for matrix calculation, we choose Pthread, OpenMP and CUDA as our language. In order to make a detailed comparison of two algorithms, we need Pthread and OpenMP as our tool language. Furthermore, the multiplication of matrix computing can't be neglected, thus we choose CUDA to exploit the GPU power.

Experimental Methodology

To further evaluate the computation effect on performance, we use the same parallelization tools in two algorithms that matrix size has a great range. More specifically, we defined the matrix size as 256, 512, 1024, 2048, 4096 and 8192. We repeat each experiment fifty times and take the sum to keep the result easy to analyze.

The testing environment is showed below,

OS: Windows 10

CPU: Intel i7 8550U 1.8G

GPU: Nvidia GTX1060

Timeout: 30000 seconds

Loop: 50 times

Experimental Results

First, we use a random constant for matrix size is 256, 512, 1024, 2048, 4096 and 8192 to do matrix multiplication. we record each element and computation results to check correctness. The results were shown in Table1. It represents the Pthread we used to do parallel programming. The total time represents the average of execution time by fifty times. The speedup and efficiency are both based on the serial program.

As shown[Table2][Table3], the speedup and efficiency of classical matrix and Strassen algorithm by using Pthread, OpenMP and CUDA.

Under the premise that the calculation result is correct. When the matrix size is

8192*8192[Table1], both Pthread and OpenMP are timeout instead of CUDA. It is not difficult to find that CUDA has an advantage in matrix computing due to the help of the GPU.

Further analyzed of the traditional matrix multiplication. When the matrix size is larger than 1024*1024, the parallelization efficiency of Pthread is better than that of OpenMP. The Strassen algorithm is more suitable for using OpenMP because of the more complicated calculation method.

Array Size	256	512	1024	2048	4096	8192
classical	5.242	45.301	348.293	2854.328	21562.769	timeout
strassen	4.593	35.521	296.049	2085.327	17064.325	timeout
classical pthread	3.154	17.124	130.723	971.415	7723.912	timeout
strassen pthread	1.581	16.121	112.433	851.415	6912.319	timeout
classical omp	2.158	16.298	130.796	1035.809	8779.625	timeout
strassen omp	2.025	13.719	103.005	810.606	6690.526	timeout
classical cuda	0.364	3.246	52.153	450.244	2329.946	5596.781
strassen cuda	0.574	2.529	44.171	398.112	1798.022	4760.105

Table1. Execution time of parallel tools in tradition and Strassen algorithm (unit: seconds), we set the timeout limitation to 30000 seconds.

		256	512	1024	2048	4096
Speed Up - Pthread	classical	166%	265%	266%	294%	279%
	strassen	291%	220%	263%	245%	247%
Speed Up - OpenMP	classical	243%	278%	266%	276%	246%
	strassen	227%	259%	287%	257%	255%
Speed Up - CUDA	classical	1440%	1396%	668%	634%	925%
	strassen	800%	1405%	670%	524%	949%

Table2. Speed up of parallel tools in traditional and Strassen algorithm, Speed up = (Sum of execution time of matrix multiplication) / (Sum execution time of matrix multiplication in parallel tools).

		256	512	1024	2048	4096
Efficiency - Pthread	classical	0.602	0.378	0.375	0.340	0.358
	strassen	0.344	0.454	0.380	0.408	0.405
Efficiency - OpenMP	classical	0.412	0.360	0.376	0.363	0.407
	strassen	0.441	0.386	0.348	0.389	0.392
Efficiency - CUDA	classical	0.069	0.072	0.150	0.158	0.108
	strassen	0.110	0.056	0.127	0.139	0.083

Table3. Efficiency of parallel tools in traditional and Strassen algorithm, Efficiency = (Sum execution time of matrix multiplication in parallel tools) / (Sum of execution time of matrix multiplication).

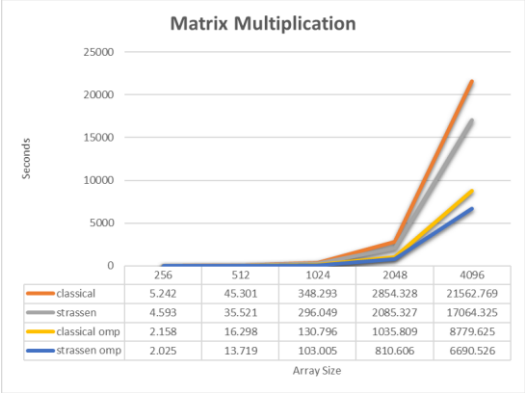


Fig11. matrix multiplication in Pthread

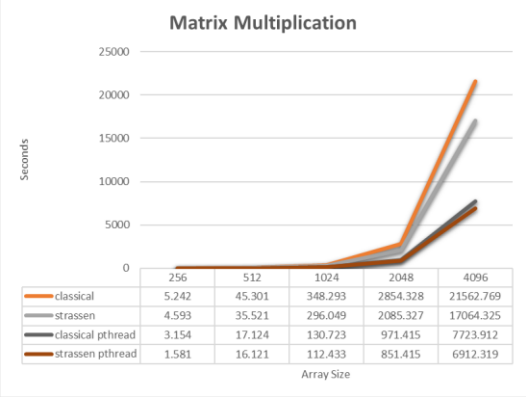


Fig12. matrix multiplication in OpenMP

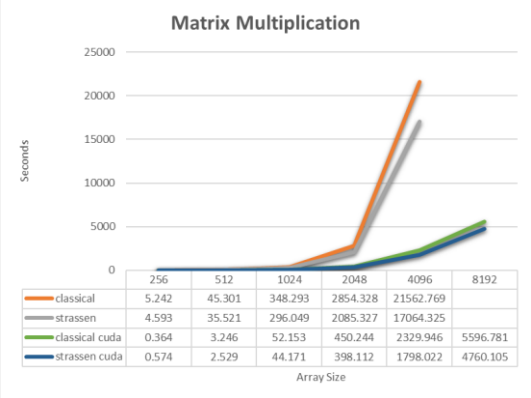
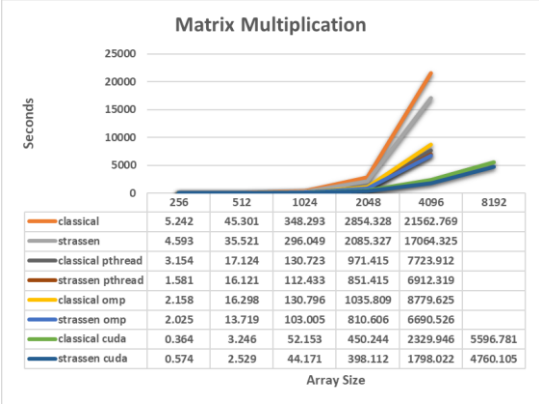


Fig13. matrix multiplication in CUDA



	256	512	1024	2048	4096	8192
classical	5.242	45.301	348.293	2854.328	21562.769	
strassen	4.593	35.521	296.049	2085.327	17064.325	
classical pthread	3.154	17.124	130.723	971.415	7723.912	
strassen pthread	1.581	16.121	112.433	851.415	6912.319	
classical omp	2.158	16.298	130.796	1035.809	8779.625	
strassen omp	2.025	13.719	103.005	810.606	6690.526	
classical cuda	0.364	3.246	52.153	450.244	2329.946	5596.781
strassen cuda	0.574	2.529	44.171	398.112	1798.022	4760.105

Fig14. matrix multiplication in parallel tools

Related work

Parallel Strassen algorithm[15] was implemented to optimize matrix multiplication. This algorithm reduces the problem of high computational costs and the low speed of traditional sequential algorithms.

An efficient multi-algorithms for OpenMP[11]. In this paper, the matrix multiplication algorithms were used to solve the matrix computation problems which is implemented by CPU.

An efficient multi-algorithms for GPUs[4]. In this paper, the matrix multiplication is used to solve the matrix computation problems which were implemented by CPU and GPU. The author found that GPU performance is faster than the CPU.

An efficient paralleled Strassen Algorithm[6]. In this paper, a parallel algorithm of calculation using traditional sequential algorithms and Strassen Algorithm is realized. The results of this paper show that the execution time of the parallel algorithm has been considerably reduced the execution time instead of the non-paralleled algorithm. The author proposes an algorithm which is implemented with CUDA by GPU to solve the eigenvalues of symmetric matrices.

Conclusions

Although, the Strassen algorithm reduces one-time multiplication and increases the twelve times addition operation. But it reduces the time complexity. Also, matrix multiplication is a key factor in time complexity. Because of this, there are different speed up by different parallelization tools.

The time complexity of traditional matrix multiplication is $O(n^3)$. The Strassen algorithm has only seven matrix multiplication operations for each Divide and conquer algorithm, so it can be obtained according to the main theorem. The time complexity is $O(n^{\log_2 7}) = O(n^{2.807})$. However, the numerical stability of Strassen's algorithm is poor.

Since matrix multiplication is the bottleneck of large matrix computing systems, we use different parallel tools Pthread, OpenMP and CUDA[20] for parallelization. Trying to compare and analyze to find the most suitable tool.

1 Pthread: Classical matrix multiplication algorithm is simpler than Strassen algorithm, the speed up is better than OpenMP.

2 OpenMP: The performance of OpenMP depends on the quality of the compiler used, when we use high-speed CPU and excellent complier, choose to use OpenMP is better.

3 CUDA: GPU has powerful computing capability. It is good parallelization speed up for both algorithms. Because of the large amount of stream multiprocessor and high memory bandwidth, GPU is suitable for parallel computation on a very large data set. The large scale matrix computation has plenty of intrinsic parallelisms and can be accelerated by GPU efficiently.

Our experimental results show the benefits of applying parallelism to Matrix multiplication, experimental results can be accelerated very quickly.

In the future, will focus on more complicated linear algebra computations such as matrix multiplication and library test. Besides these basic numeric computations, some real applications would be accelerated.

References

[1] Ting Wan, Zhao Neng Jiang, Yi Jun Sheng. “Hierarchical Matrix Techniques Based on Matrix Decomposition Algorithm for the Fast Analysis of Planar Layered Structures”, IEEE Transactions on Antennas and Propagation.

[2] Hongbin Li, P. Stoica, Jian Li. “Computationally efficient maximum likelihood estimation of structured covariance matrices”, IEEE Transactions on Signal Processing.

[3] Ahmad Jibir Kawu, Aishatu Yahaya Umar, Saminu I. Bala. “Performance of one-level recursion parallel Strassen's algorithm on dual core processor”, 2017 IEEE 3rd International Conference on Electro-Technology for National Development (NIGERCON).

[4] S. Huss-Lederman, E.M. Jacobson, J.R. Johnson, A. Tsao, T. Turnbull. “Implementation of Strassen's Algorithm for Matrix Multiplication”, Supercomputing '96:Proceedings of the 1996 ACM/IEEE Conference on Supercomputing

[5] Himeshi De Silva, John L. Gustafson, Weng-Fai Wong. “Making Strassen Matrix Multiplication Safe”, 2018 IEEE 25th International Conference on High Performance Computing (HiPC).

[6] Ayaz ul Hasan Khan, Mayez Al-Mouhamed, Allam Fatayer. “Optimizing strassen matrix multiply on GPUs”. 2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD).

[7] Pujianto Yugopuspito, Sutrisno, Robertus Hudi. “Breaking through memory limitation in GPU parallel processing using Strassen Algorithm”, 2013 International Conference on Computer, Control, Informatics and Its Applications (IC3INA)

[8] Junjie Li, Sanjay Ranka, Sartaj Sahni. “Strassen's Matrix Multiplication on GPUs”, 2011 IEEE 17th International Conference on Parallel and Distributed Systems.

[9] B. Dreier, M. Zahn, T. Ungerer. “Pthreads for Dynamic and Irregular Parallelism”, SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing.

[10] Juan F.R. Herrera, Leocadio G. Casado, Remigijus Paulavicius, Julius ilinskas, Eligius M.T. Hendrix. “On a Hybrid MPI-Pthread Approach for Simplicial Branch-and-Bound”, 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum.

[11] Neha Singh. “Automatic parallelization using OpenMP API”, 2016 International Conference on Signal Processing, Communication, Power and Embedded System (SCOPES).

[12] N. V. Sunitha, K. Raju, Niranjan N. Chiplunkar. “Performance improvement of CUDA applications by reducing CPU-GPU data transfer overhead”, 2017 International Conference on Inventive Communication and Computational Technologies (ICICCT).

[13] Massimiliano Fatica. “CUDA toolkit and libraries”, 2008 IEEE Hot Chips 20 Symposium (HCS).

[14] Matrix multiplication, https://en.wikipedia.org/wiki/Matrix_multiplication

[15] Strassen algorithm, https://en.wikipedia.org/wiki/Strassen_algorithm

[16] Matrix, https://en.wikipedia.org/wiki/Matrix

[17] OpenMP, https://zh.wikipedia.org/wiki/OpenMP

[18] OpenMP, https://www.openmp.org/

[19] CUDA, https://en.wikipedia.org/wiki/CUDA

[20] Final Project Source Code, http://bit.ly/2QYcnNn