Performance Comparison of Matrix Multiplication Algorithms

Pradyumna S
Department of Computer Science
PES University
Bengaluru
pradyumnasridhar@gmail.com

***Abstract*— Software has gained its importance in all aspects of every walk of life. Algorithms play a vital role in ensuring the successful operation of any software. According to the applications, various algorithms have been designed in order to accomplish the desired goals. This paper therefore aims to provide an insight on significance of performance and further, throw light on various matrix multiplication algorithms. The paper also puts forth a comparative analysis on the efficiency of performance of matrix multiplication algorithms by considering suitable factors. This work thus enables one to comprehend the suitability of the algorithm based on the chosen factors.**

Keywords— Algorithms, Performance, Matrix Multiplication, Software Quality, Analytics

INTRODUCTION

Ever since the introduction of software in human life, it has played a most significant role in modulating the standard of living and communication. The essence of software lies beneath the competency level of the algorithms designed and developed. Hence, algorithms tune the quality of software developed which in turn decides the depth of customer satisfaction.

Algorithms acts as a travel light for software personnel to proceed in a systematic manner in order to arrive at the solution [1]. The main benefit of algorithms is to ensure that the task is completed within the stipulated time frame. Further, it also assures the developer to progress as per the decided flow of logic such that result is attained as per the predefined specifications. Further, algorithms act a tool which simplifies the thought process and also brings in clarity for the code to be implemented [2]. Additionally, algorithms are language independent and hence enable the designer and developer to translate the algorithm into language specific code. Furthermore, algorithms are generic in nature allowing any reader to comprehend the flow of the logic embedded in the algorithm [3].

There are various types of algorithms to suit the application domain. It includes algorithms for sorting, searching, combinatorics, string processing, graph, geometric problems, and numerical problems and so on [5]. However, based on the design strategies several algorithm paradigms are further designed. They include brute force, divide and conquer, decrease and conquer, transform and conquer, space time tradeoff, dynamic programming, greedy technique and so on [6]. However, it is worth to note that the algorithms mentioned above involves computations. Matrix multiplication is one of the most widely used computations in almost all the application domain related algorithms as well in the design strategies related algorithms.

Matrix multiplication algorithms have evolved over the time from the brute force inner product multiplication method to complex matrix multiplication algorithms such as Coppersmith-Winograd algorithm [7]. Matrix multiplication forms the basis for many other mathematical operations such as computing the inverse of the matrix, LUP decomposition, Gaussian elimination which in turn leads towards designing of various applications [8].

Matrix multiplication is used for scientific, commercial, business and for entertainment applications as well [4]. Thus, design and development of matrix multiplication algorithm has an influencing impact on the applications. This paper therefore aims to provide a comparative analysis on some of the widely-used matrix multiplication algorithms in terms of performance.

Section II gives a literature survey on various works that has happened in the area of algorithms and matrix multiplication algorithms. Section III provides information about the comparative analysis of various matrix multiplication algorithms and inferences. Conclusion section provides a summary of the work carried out.

LITRATURE SURVEY

Author of [8] has made improvements in upper bound of $O(n^{2.376})$ for matrix multiplication. This has been achieved using analysis on tensor powers, a method similar to that of Coppersmith and Winograd.

The author of [9] has introduced a novel method of multiplying rectangular matrices. This method has proven to be effective in terms of asymptotic time complexity, for $n \times n^k$ and $n^k \times n$. This has been achieved by using tensor powers as one of the basic constructions and analysis of this construction for rectangular matrix multiplication.

In [10], the authors have conducted a systematic comparison of the standard method and the Strassen's matrix multiplication algorithm, along with their applicability on GPU offloading of

matrix multiplication computations. They have conducted their experiments on three generations of GPUs, across various dimensions of matrices.

The author of [11] gives an insight into the comparison of the arithmetic complexity of matrix operations against the number of bit operations required for those operations. This gives a better understanding of the practical efficiency of the algorithm, rather than the theoretical efficiency. The author focuses on the study of APA algorithms for matrix multiplication, and demonstrates that they are more efficient in practice. A new class of APA algorithms is also introduced.

COMPARTIVE ANALYSIS OF MATRIX MULTIPLICATION ALGORITHMS

From the literature survey, it is evident that matrix multiplication has a vital role to play in software application. This section however limits to consider few of the matrix multiplication algorithms such as

A. Brute force inner product multiplication
B. Divide and conquer method
C. Strassen's method

A. Brute force inner product multiplication

This method of multiplying two matrices performs the multiplication of values in each row of the Matrix A against each column of the Matrix B. This multiplication is in the asymptotic complexity of $O(n^3)$.

B. Divide-and-Conquer with brute force method

Here, both the matrices are split up into four submatrices each. Each of these submatrices are further split up, recursively, till submatrices of the required dimension is received. After the base case of recursion is reached, the corresponding submatrices are multiplied using brute-force method, and are assembled back together, resulting in the product matrix.

C. Strassen's method

Both of the above-mentioned algorithms perform 8 individual multiplications. Strassen's method uses 7 multiplications, and therefore has a slightly lesser asymptotic complexity of the order $O(n^{2.807})$ as compared to the normal matrix multiplication method.

All the above-mentioned algorithms are compared against factors such as

A. Parallel vs sequential computing
B. Use of SIMD architecture
C. Precision of values
D. Matrix dimensions
E. Processors with different per-core performance

A. *Parallel vs Sequential method*

One of the factors for the performance comparison of the algorithms is based on their parallelizability. In this work, concurrency is achieved using threads. Pthreads module has been used for the same.

The number of threads was chosen as a multiple of the number of processors. For the divide and conquer with brute force method and Strassen's method, the two matrices are split into submatrices until the number of submatrix pairs is not less than the number of available threads. Each of the submatrices is assigned to a thread without compromising symmetric load distribution. All further recursive subdivisions of the matrices occur on the assigned thread.

B. *Use of SIMD Architecture*

Single Instruction Multiple Data allows us to take advantage of the fact that the same multiplication operation has to be performed over many pairs of values. Using this architecture, the data can be fetched in blocks. As an instance, one row of a matrix is used instead of one value at a time. This allows the process to execute much faster, as compared to the traditional CPU design. The single main overhead of using this architecture is that, SIMD requires all the data to be allocated contiguously in memory. The AVX SIMD instruction set was used.

C. Precision of Values

In this work, the values used to populate the matrix were floating-point numbers, precise up to 2 decimal places. These values were obtained using pseudo-random number generator functions. The range of values is between 100 and 1000.

D. Matrix Dimensions

The dimensions considered varied from 16×16 matrices to 4096×4096, in powers of two. Square matrices are used in this work.

EXPERIMENTAL RESULTS

In this section, the analysis of 6 results is presented, over the combinations of the factors discussed in Section III.

- Comparison of the **effectiveness of using SIMD**, among various algorithms **with sequential implementation**:

Figure1 provides a graphical comparison of the improvement in throughput on the x-axis against the dimension of the matrices on the y-axis, for various sequential algorithms, when SIMD is used. The improvement in throughput is calculated as follows -

$$\text{Improvement in throughput} = \frac{\text{Execution time with SIMD}}{\text{Execution time without SIMD}}$$

From Figure1, it is evident that with increasing matrix dimensions, brute force inner product multiplication method

scales drastically, performing 8.35 times faster than its non-SIMD counterpart (for 4096×4096 matrix). There is an increase in performance, in the Strassen as well as the divide and conquer with brute force implementations, although not as drastic as that of inner product multiplication method. Strassen's method with SIMD performs 1.50 times faster than that without SIMD, and the divide and conquer method performs 1.445 faster (both for 4096×4096 matrices).

For almost all dimensions, the use of SIMD is at least not non-beneficial, except for the case of 64×64 matrices where the SIMD implementation performs worse than the plain implementation (0.33 times faster).

- Comparison of the **effectiveness of using SIMD**, among various algorithms **with parallel implementation:**

Figure2 represents the improvement in throughput on the y-axis and the matrix dimension on x-axis.

$$\text{Improvement in throughput} = \frac{\text{Execution time with SIMD}}{\text{Execution time without SIMD}}$$

Although strassen's algorithm has a better asymtotic upper bound as compared to the divide and conquer with brute force method, when implemented using parallel computing, Strassen's algorithm does not seem to outperform the divide and conquer with brute force method, by a significant factor. With the use of SIMD, both the algorithms perform better than those without SIMD, with an average of 1.49 times faster in the case of divide and conquer with brute force method, and 1.38 times faster for the strassen's method, for larger matrix dimensions. An exceptional result occurred for the case of 128×128 matrices, where the non-SIMD implementation outperformed the SIMD implementation. All the above implementations were done using 8 parallel threads.

- Comparison of the **effectiveness of the using parallel computing** as compared to sequential computing, **along with SIMD**:

Figure3 compares the improvement in throughput against the matrix dimensions.

$$\text{Improvement in throughput} = \frac{\text{Execution time when parallelized}}{\text{Execution time when sequential}}$$

On an average, use of parallel computing increased the throughput for Strassen's method by 3.25 times and divide and conquer with brute force by 2.84 times, as compared to the sequential implementation. Both the algorithms showed significant improvements in throughput, with SIMD, when parallelized.

- Comparison of the **effectiveness of the using parallel computing** as compared to sequential computing, **without SIMD**.

$$\text{Improvement in throughput} = \frac{\text{Execution time when parallelized}}{\text{Execution time when sequential}}$$

The parallelized implementation performs better than the sequential implementation without the use of SIMD. The average improvements in throughput are 2.93 for divide and conquer method and 2.91 for Strassen's method. Figure4 shows that Strassen's method has a higher improvement when parallelized, without SIMD as compared to with SIMD. However, for the divide and conquer method, the use of SIMD has a slightly negative impact on the improvement in throughput when parallelized.

- **Overall growth** of various algorithms, **without SIMD architecture.**

Figure5 compares the growth of all the methods used, along with their parallel variations, in absolute terms. Brute force multiplication has a sharp increase in execution time, as it has the highest rate of growth, and is not parallelized. Strassen's parallelized version performs the best, with the least growth rate. In between the large range of execution (from 1416.81 seconds for brute force method to 63.65 seconds for Strassen's parallel, for 4096×4096 matrices), exists all the other methods used. Divide and conquer parallelized performs better than Strassen's sequential, despite the higher asymptotic growth rate. There is a considerable difference in the performance between the parallelized and the sequential forms of divide and conquer with brute force method, with a gap of 314.41 seconds.

- **Overall growth** of various algorithms, **with SIMD architecture.**

With the use of SIMD, its effect on each method varies, as shown in Figure 6. Brute force inner product multiplication has improved as compared to its performance without SIMD. Although divide and conquer with brute force method has the longest execution time, it still performs better than the sequential implementation. Strassen's parallel implementation is still the fastest, taking 45.82 seconds for 4096×4096 matrices.

CONCLUSION

Software has its significance since the time of its introduction. Several evolutions have occurred to improve software using algorithms as the main area of focus. Algorithms related to matrix multiplication have therefore attained considerable importance as complex computations can be rendered effectively through matrix multiplication.

This paper provides a comparative study of some of the prominent matrix multiplication algorithms, against various factors such as the algorithm's parallelizability, compatibility with SIMD architecture and the rate of growth, among others. Strassen's method with SIMD, when parallelized, was the best performing algorithm, amongst those considered. The brute force inner product multiplication method was found to scale

extremely well with SIMD. All of the selected algorithms were found to perform better when parallelized, as compared to their sequential counterparts. The divide and conquer with brute force method was found to perform less efficiently with SIMD as compared to without. The analysis can be extended to more recent algorithms for matrix multiplication such as [8] and [9].

REFERENCES

[1] Cormen, Leiserson, Rivest, Stein, **"**Introduction to Algorithms", The MIT Press, 2009.

[2] Steven S. Skiena, "The Algorithm Design Manual", Springer, 2008.

[3] Edgar Dijkstra, "Goto statement considered harmful", 1968

[4] Zelun Luo, Boya Peng, De-An Huang, Alexandre Alahi, Li Fei-Fei, "Unsupervised Learning of Long-Term Motion Dynamics for Videos", 2017

[5] Bürgisser, Clausen, Shokrollahi, "Algebraic Complexity Theory", Spinger, 1997

[6] Ellis Horowitz, Sahni, Rajasekaran, "Computer Algorithms", Computer Science Press

[7] Don Coppersmith, Shmuel Winograd, "Matrix multiplication via arithmetic progressions", Proceedings of the nineteenth annual ACM symposium on Theory of computing, 1987

[8] Virginia Vassilevska Williams, "Multiplying matrices in $O(n^{2.373})$ time", also visit http://theory.stanford.edu/~virgi/matrixmult-f.pdf, 2014.

[9] Francois Le Gall, "Faster Algorithms for Rectangular Matrix Multiplication", also visit https://arxiv.org/pdf/1204.1111v2.pdf, 2012.

[10] Peng Zhang, Yuxiang Gao, "Matrix Multiplication on High-Density Multi-GPU Architectures: Theoretical and Experimental Investigations", Springer, 2015.

[11] V.Y. Pan, "The bit complexity of matrix multiplication and of related computations in linear algebra.", Elsevier, 1985.

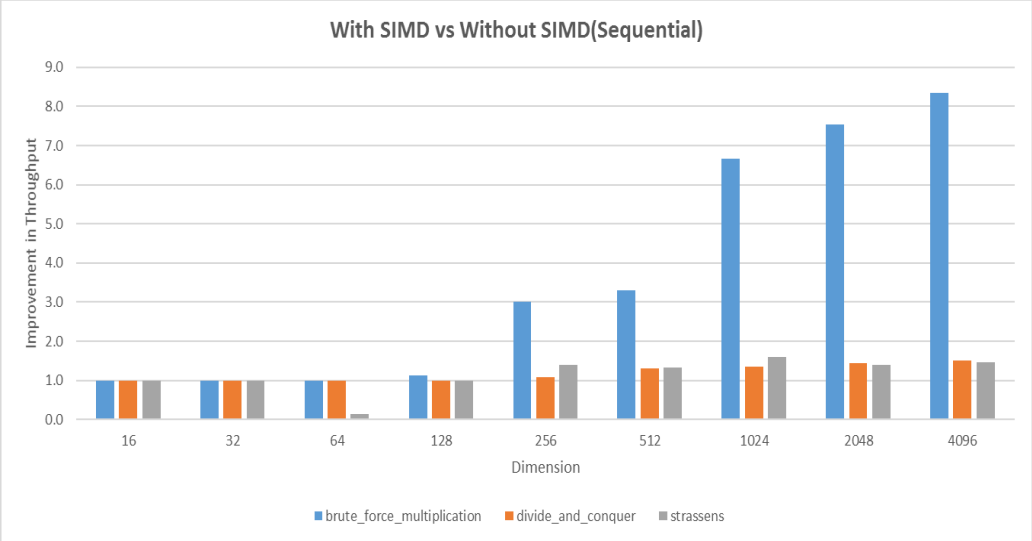

Figure 1. Comparison of the **effectiveness of SIMD** for **sequential implementation** of various algorithms.

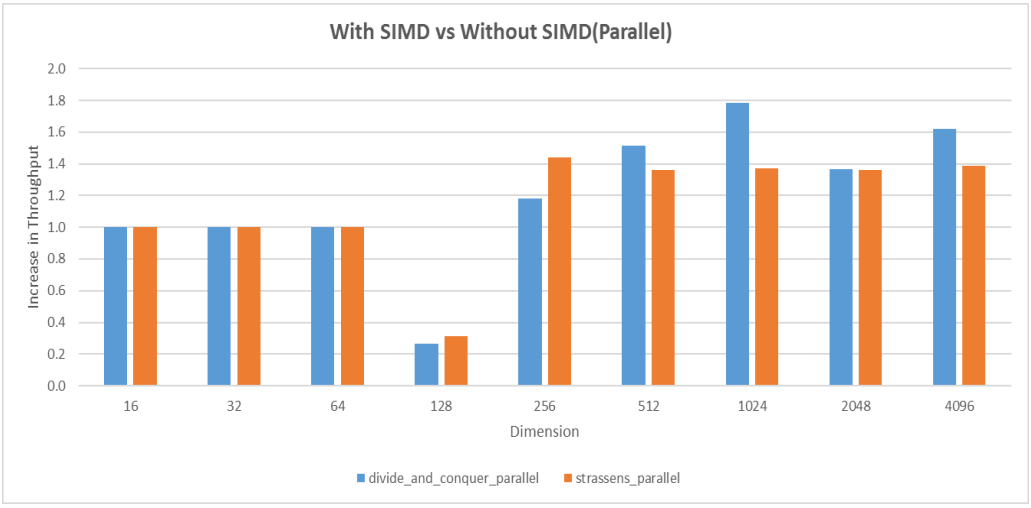

Figure 2. Comparison of the **effectiveness of SIMD** for **parallelized implementation** of various algorithms.

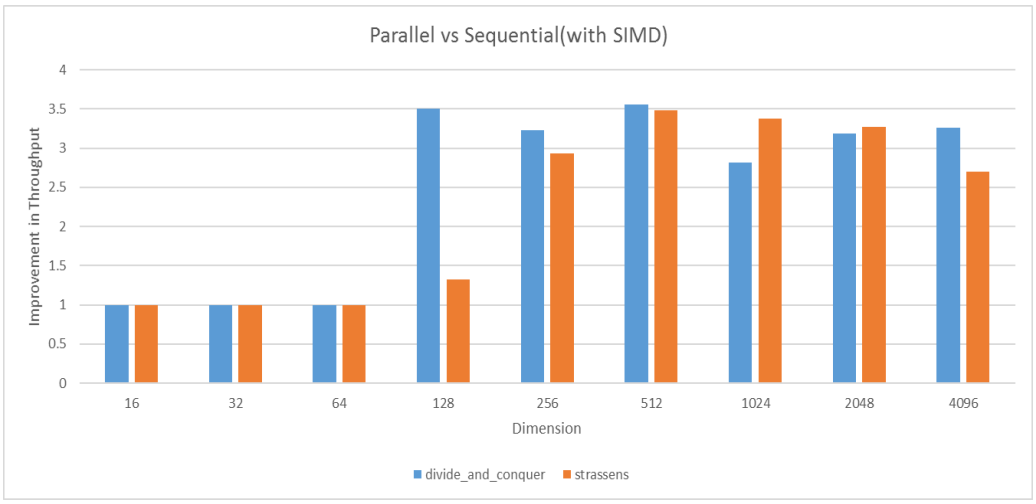

Figure 3. Comparison of the **effectiveness of parallelizing** various algorithms, along **with SIMD**.

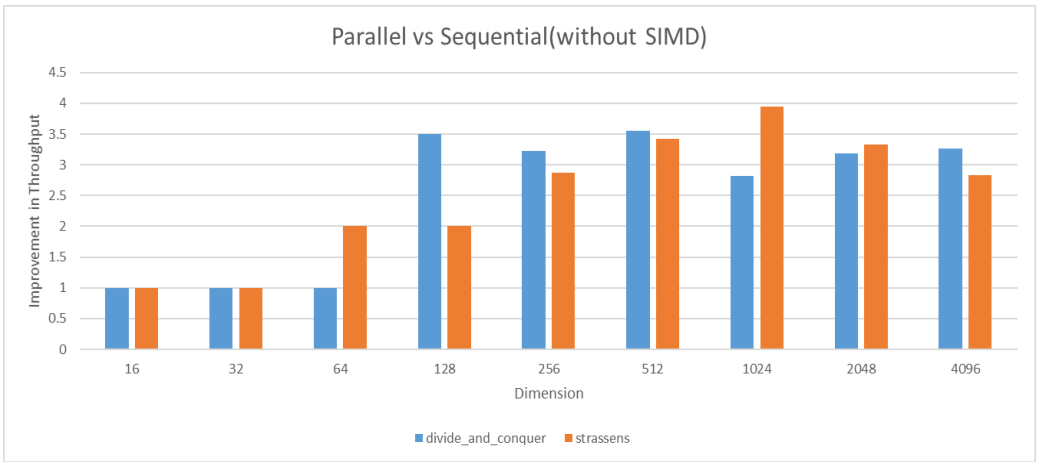

Figure 4. Comparison of the **effectiveness of parallelizing** various algorithms, **without SIMD**.

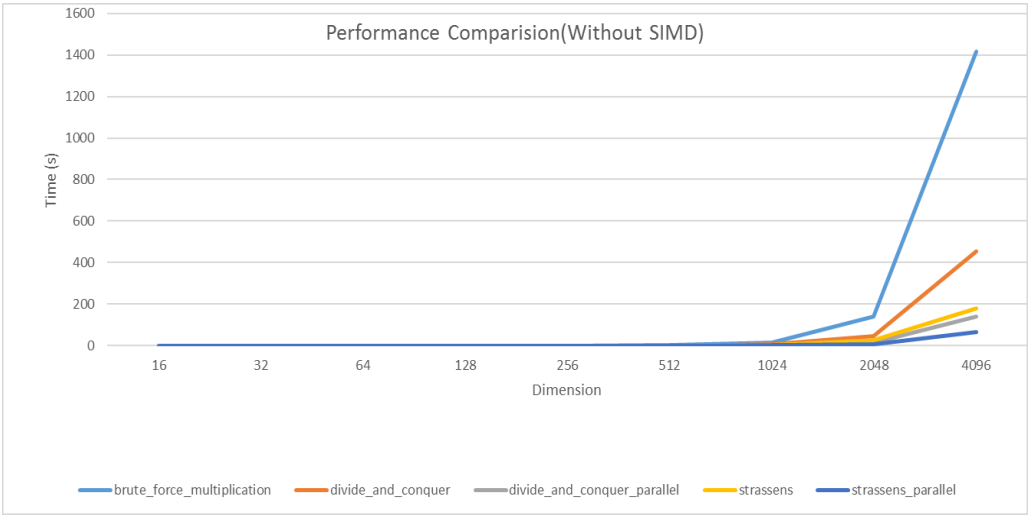

Figure 5. **Overall growth** of various algorithms, **without SIMD**

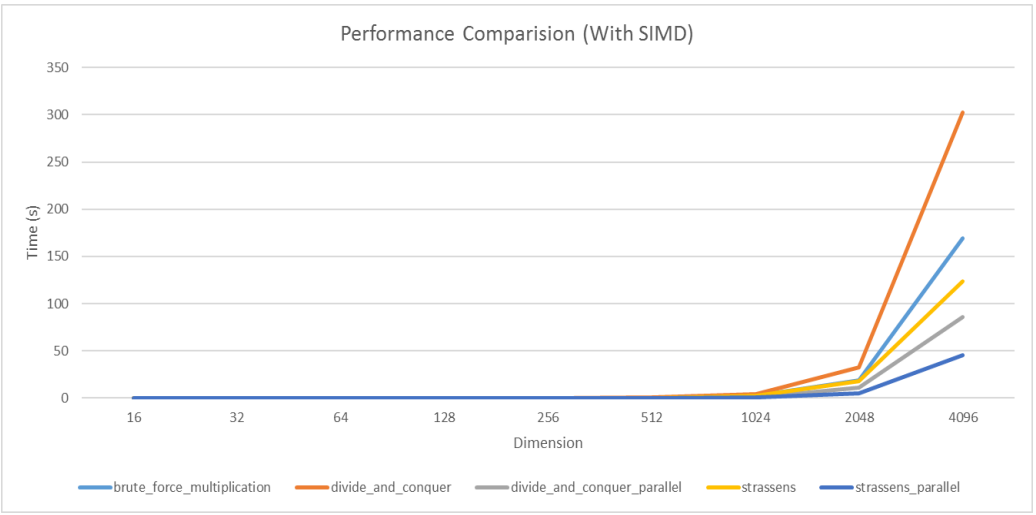

Figure 6. **Overall growth** of various algorithms, **with SIMD**