Matrix Multiplication and Ways to Multiply Faster

Ammar Rasyad Chaeroel - 13521136[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]*13521136@std.stei.itb.ac.id*

***Abstract*—Matrix multiplication algorithms that we generally know has a computational worst-case complexity of O(N^3). This is generally a non-issue for smaller matrices, but as the complexity grows cubically, it starts to show how unsuited it is for larger matrices, which in turn slows work down in practical uses, e.g., linear transformations in graphical applications. This paper analyzes common matrix multiplication algorithms and find ways to optimize such algorithms.**

***Keywords*—matrix, multiplication, algorithm, efficiency**

I. Introduction

Matrices are rectangular multi-dimensional arrays that can hold numbers, symbols, or expressions. Matrices play a huge role in graphics, such as linear transformations of images, even the images itself are represented in matrices. Machine learning uses matrices to determine the weighted sums of their inputs.

$$\begin{bmatrix} 2 & 3 & 5 \\ 1 & 4 & 1 \end{bmatrix}$$

Figure 1 - A 2x3 matrix holding 6 numbers

Matrix multiplication is a binary operation that produces a single matrix from two matrices. The number of columns of the first matrix must be equal to the number of rows of the second matrix. The product of the matrix has the number of rows of the first matrix and the number of columns of the second matrix. For example, a matrix A with dimensions *I* x *J* multiplied by a matrix B with dimensions *N* x *M* produces a matrix denoted as AB with dimensions *I* x *M*. Matrix multiplication can be chained to do a matrix chain multiplication to multiply several matrices.

Matrix multiplication has several properties. It is generally non-commutative (AB ≠ BA), distributive (A(B+C) = AB + AC), associative (A(BC) = (AB)C). Matrices can be multiplied by a scalar, in which the product is a matrix with its entries multiplied by the scalar. The transpose of a product is the multiplication in the reverse order of the transposes of the factors ($(AB)^T = B^TA^T$).

The naïve way to do matrix multiplication is as follows:

$$\begin{bmatrix} 2 & 3 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 3 & 7 \\ 3 & 1 \end{bmatrix} = \begin{bmatrix} 15 & 17 \\ 15 & 11 \end{bmatrix}$$

Figure 2 - Example of a matrix multiplication

Elements of a matrix is denoted by a_{ij}, *i* and *j* being the row and column respectively. a_{ij} of the product matrix is the result of multiplying term-by-term the entries of the *i*th row of the first matrix and the *j*th column of the second matrix and summing these *n* products. In this example for element a_{11}, $2 * 3 + 3 * 3 = 15$. The same operation is applied for other elements of the matrix.

II. Theoretical Basis

This approach in matrix multiplication (for square matrices) results in a computational complexity of $O(N^3)$, which means that the complexity grows cubically. As such, when matrices get larger, the time required to calculate the product matrix grows cubically. This is not ideal for larger matrices, as practical applications of matrix multiplications do so with large matrices in large quantities.

For reference, the naïve approach to matrix multiplication is written as such in Python and C++:

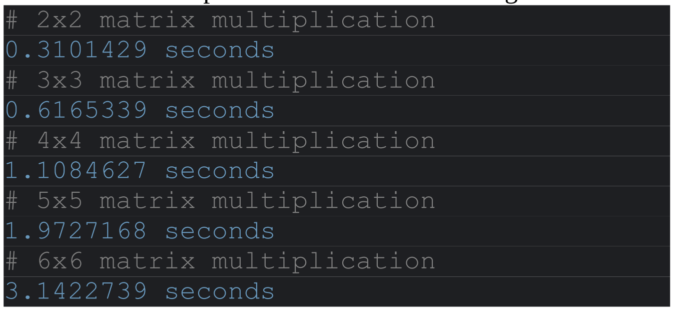
```
for i in range(len(A[0])):
# Columns of A
    for j in range(len(B)):
    # Rows of B
        C[i][j] = 0
        for k in range(len(B)):
        # Either the columns of A or rows
of B
            C[i][j] += A[i][k] * B[k][j]
# Or use Python's built-in matrix multi-
plication operator (Only for NumPy matri-
ces)
C = A @ B
```

Figure 3 - Python code

```
for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
        c[i][j] = 0;
        for (k = 0; k < 3; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

Figure 4 - C++ code

This is an example of matrix multiplication operations over 100.000 iterations to put into scale on how fast it grows:



```
# 2x2 matrix multiplication
0.3101429 seconds
# 3x3 matrix multiplication
0.6165339 seconds
# 4x4 matrix multiplication
1.1084627 seconds
# 5x5 matrix multiplication
1.9727168 seconds
# 6x6 matrix multiplication
3.1422739 seconds
```

Figure 5 - Elapsed time for each operation in Python

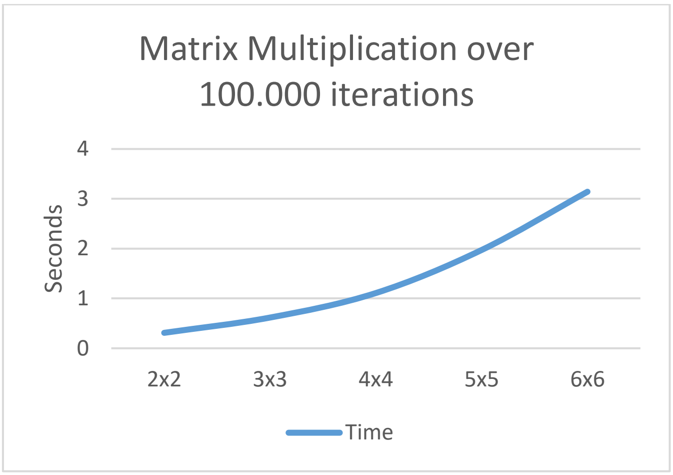
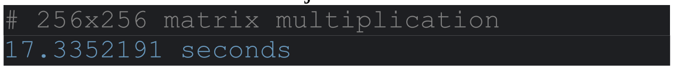


Figure 6 - Chart for elapsed time

Having 3 nested loops is a recipe for disaster when dealing with large matrices. It may not look like an issue at first, especially since we are dealing with such small matrices that it only starts to creep into the seconds territory after we deliberately repeat the same operation 100.000 times. How about a 256 x 256 matrix with just 10 iterations?



```
# 256x256 matrix multiplication
17.3352191 seconds
```

Figure 7 - Time required for a naive 256 x 256 matrix multiplication in Python

With just 10 iterations, a 256x256 matrix multiplication takes around 17.33 seconds, taking 1.73 seconds every iteration, making it unsuitable for real world use. The slowdown is even more apparent with larger matrices, both in size and in quantity, which is to be expected in work such as image processing.

Another aspect worthy of noting is computational or asymptotic complexity. Big O notation is a mathematical notation, but in computer science it is used to classify algorithms according to how their run time or even space requirements grow as the input size grows. Big O notation is also called worst-case complexity, as it describes the upper bound on the algorithm's growth rate. Examples of big-O notation include $\mathcal{O}(N^3)$ which grows cubically, $\mathcal{O}(N^2)$ which grows quadratically. Generally, the smaller the growth, the faster and better, although it is not the whole picture.

III. Approaches to Matrix Multiplication

A. Tensor Decomposition

A matrix multiplication operation is bilinear, so it can be represented by a 3-dimensional tensor. A tensor is an object that describes a multilinear relationship between sets of objects related to a vector space, just like scalars and vectors. In fact, scalars are essentially tensors of rank 0 (or 0-dimensional tensor), while vectors are tensors of rank 1 (or 1-dimensional tensor). Tensors are used extensively in physics because of its importance in solving physics problems in areas such as mechanics, electrodynamics, and others. See Fig. 8a for a representation of a 2 x 2 matrix multiplication operation as a 3-dimensional tensor with the size of 4 x 4 x 4.

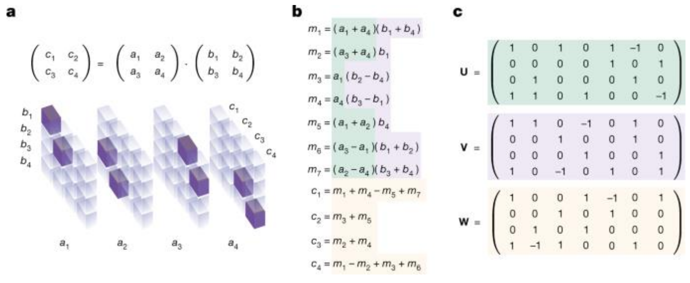

a
$\begin{pmatrix} c_1 & c_2 \\ c_3 & c_4 \end{pmatrix} = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \cdot \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}$
b_1 b_2 b_3 b_4
c_1 c_2 c_3 c_4
a_1 a_2 a_3 a_4
b
$m_1 = (a_1 + a_4)(b_1 + b_4)$
$m_2 = (a_3 + a_4)b_1$
$m_3 = a_1(b_2 - b_4)$
$m_4 = a_4(b_3 - b_1)$
$m_5 = (a_1 + a_2)b_4$
$m_6 = (a_3 - a_1)(b_1 + b_2)$
$m_7 = (a_2 - a_4)(b_3 + b_4)$
$c_1 = m_1 + m_4 - m_5 + m_7$
$c_2 = m_3 + m_5$
$c_3 = m_2 + m_4$
$c_4 = m_1 - m_2 + m_3 + m_6$
c
$\mathbf{U} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & -1 \end{pmatrix}$
$\mathbf{V} = \begin{pmatrix} 1 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 1 & 0 & 1 \end{pmatrix}$
$\mathbf{W} = \begin{pmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$

Figure 8 - Tensor decomposition of a 2x2 matrix multiplication operation

Source: https://www.nature.com/articles/s41586-022-05172-4

The matrices are decomposed into tensors to represent a matrix multiplication operation, which is denoted by $\mathcal{T}_n$ with n being the size of the matrix. In general, and this also applies to non-square matrices, it is represented by $\mathcal{T}_{n,m,p}$ as an operation between an n x m matrix and an m x p matrix. By tensor decomposition, $\mathcal{T}_n$ is decomposed to:

$$\mathcal{T}_n = \sum_{r=1}^{R} u^{(r)} \otimes v^{(r)} \otimes w^{(r)},$$

where u, v, and w are vectors.

Fig. 8a shows a tensor $\mathcal{T}_2$ that represents the multiplication of two 2 x 2 matrices. The opaque blocks represent tensor entries equal to 1, while the semi-transparent blocks represent entries equal to 0. As an example, tensor entries in (a_1, b_2, c_1) and (a_2, b_3, c_1) are set to 1 as per the following equation for matrix multiplication:

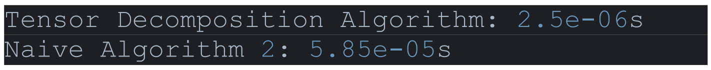
$$c_1 = a_1 b_1 + a_2 b_3$$

Fig. 8b is Strassen's algorithm, which will be explained in the Section III B. Strassen's algorithm is used for multiplying 2 x 2 matrices with 7 multiplications. Fig. 8c is the tensor factor representation of Strassen's algorithm, with u, v, and w highlighted in green, purple, and yellow respectively. It is a rank-7 decomposition of $\mathcal{T}_2$. The correspondence between 8b and 8c is shown by the colors.

This way, the computational worst-case complexity is reduced to $\mathcal{O}\left(N^{\log_n R}\right)$, with N being the size of a square matrix, and the tensor rank R. However, there is a catch.

Matrix rank decomposition is relatively easy to compute using techniques such as SVD (Singular Value Decomposition) and RRQR factorization (rank-revealing QR factorization). Unlike matrix rank decomposition, tensor rank decomposition is NP-complete. NP-complete is short for "non-deterministic polynomial-time complete," meaning that the problem is not solvable in realistic, polynomial time, but the solution can be verified in polynomial time. In other words, it is not algorithmically and thus programmatically possible to determine tensor rank without a non-deterministic Turing machine or an AI/neural network.

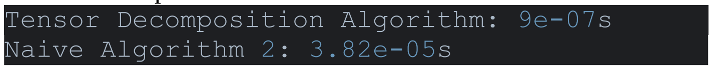
For this example, a 2 x 2 matrix with predetermined tensor rank and vectors are used.



```
Tensor Decomposition Algorithm: 2.5e-06s
Naive Algorithm 2: 5.85e-05s
```

Figure 9 - Tensor decomposition compared to the naive algorithm

In this example, tensor decomposition is over 20x faster than the naïve algorithm, although the results may vary wildly depending on the compiler, the computer configuration, CPU usage, among others. For example, setting the compiler flag -O3 can increase the performance boost from over 20x to 43x.



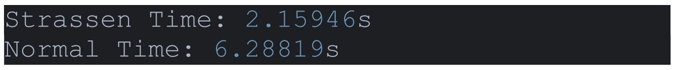
```
Tensor Decomposition Algorithm: 9e-07s
Naive Algorithm 2: 3.82e-05s
```

Figure 10 - By setting the -O3 flag, tensor decomposition is 43x faster

Tensor decomposition is more restrictive than other algorithms due to how it is programmed. Since tensor rank decomposition is inherently NP-complete, it is not simple to implement. With other algorithms being just as fast or only ever so slightly slower, tensor decomposition is not feasible for the time being, as the origin of this algorithm comes from an AI called AlphaTensor, specifically designed to discover new, efficient, and provably correct algorithms for matrix multiplication.

B. Strassen Algorithm

The Strassen algorithm, named after Volker Strassen, is a fast algorithm for matrix multiplication with better asymptotic complexity than the naïve algorithm for larger matrices. Below a certain point, the naïve algorithm is preferable. The Strassen algorithm uses a divide-and-conquer approach to reduce the number of multiplications needed. For example, in a 2 x 2 matrix, there are 8 multiplications needed when using the naïve approach, while only 7 multiplications are needed for the Strassen algorithm. The benefits of Strassen's algorithm are much more apparent in larger matrices, such as in this example where a 1024 x 1024 is used:



```
Strassen Time: 2.15946s
Normal Time: 6.28819s
```

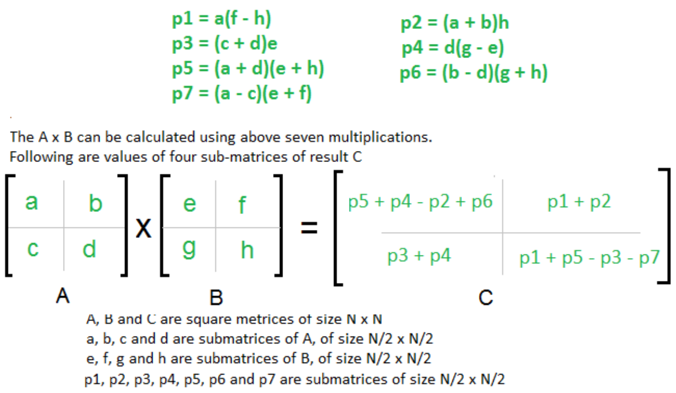
Figure 11 - 1024 x 1024 matrix multiplication operation in C++

In this example, Strassen's algorithm is 2.912x faster than the naïve algorithm. C++ is used as opposed to Python because Python has underlying optimizations for certain operations, where it calls operations written in C to speed up certain operations in which will result in inconsistent and essentially incomparable elapsed time.

Strassen's algorithm works wonders on large N x N matrices with N being the power of 2. On non-square matrices and square matrices where N is not a power of 2, padding with zeroes is needed for the algorithm to work. The matrices are padded with zeroes up to P with P being the smallest power of 2 larger than N, e.g., if N is 27 then the matrix will be padded to 32. For non-square matrices, it will be padded to a square matrix.

$$\begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} B & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & 0 \end{bmatrix}$$

Figure 12 - An example of a matrix multiplication with padded zeroes



p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Figure 13 - Strassen's algorithm

Source: https://www.geeksforgeeks.org/strassens-matrix-multiplication/

The time complexity of Strassen's algorithm is approximately:

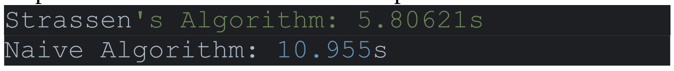
$$O\left(N^{\log_2 7}\right) = O\left(N^{2.8074}\right)$$

While Strassen's algorithm is fast and accurate, it is generally not preferred for practical applications for various reasons, such as the algorithm itself inherently using recursion, which takes extra space in the stack. The source code used for Strassen's algorithm can be found in the Appendix section.

Strassen's algorithm utilizes the naïve algorithm to a certain degree. Depending on the leaf size, or the size of the matrix where it defaults to the naïve algorithm, it can be faster or slower than the naïve algorithm. The leaf size is dependent on the architecture of the computer, there is no way to determine without trial-and-error. In this instance, the best leaf size has been determined to be 64 x 64. Strassen's algorithm is relatively easy to parallelize using available instructions in the CPU, which will be discussed in the next section.

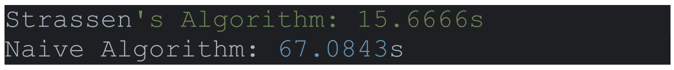
C. Multithreaded Approach to Multiplication, SIMD, and Compiler Optimizations

In current times, it's common to have more than two cores in a computer. Even the cheapest of computers have at least two cores with HyperThreading/SMT (Simultaneous Multithreading), in which a core has two execution threads to increase multithreaded performance. Most algorithms can be parallelized to take advantage of a multicore processor, and matrix multiplication algorithms are no exception. Here is an example of a 2048 x 2048 matrix multiplication.



```
Strassen's Algorithm: 5.80621s
Naive Algorithm: 10.955s
```

Figure 14 - Multithreaded 2048 x 2048 matrix multiplication



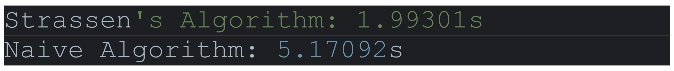
```
Strassen's Algorithm: 15.6666s
Naive Algorithm: 67.0843s
```

Figure 15 - Single-threaded 2048 x 2048 matrix multiplication

The calculation is done on an Intel Core i7-10750H CPU with 6 cores and 12 threads in total. Multithreaded processing has improved the speed of each algorithm by 2.70x and 6.124x respectively. Non-linear increase in the performance boost is caused by the nature of the code and how parallelization works. Not every part of the algorithm can be parallelized, which in turn means that not every algorithm benefits the same from multithreading.

By adding a simple compiler flag for auto-parallelization allows the compiler to automatically parallelize the code whenever possible and increase the speed of each calculation, with the only drawback being increased power consumption. Multithreaded processing is not always applicable to every situation, as there is an overhead in starting each thread corresponding to the number of logical threads available on the computer, processing data, and joining the result from each thread into one. One example of such condition where it may be counter-intuitive to utilize multiple threads is with small matrices.

Multithreading is not the only trick that can speed up matrix multiplications. Modern CPUs support SIMD instructions, short for Single Instruction Multiple Data, which can be used to further improve the performance of matrix multiplications by operating pieces of multiple data with only a single instruction (hence *Single Instruction Multiple Data*) instead of every single element in the matrices, greatly improving efficiency. SIMD instructions are an extension of the x86 ISA (Instruction Set Architecture) named SSE (Supplementary SIMD Extensions). SSE operates using 16 128-bit XMM registers.

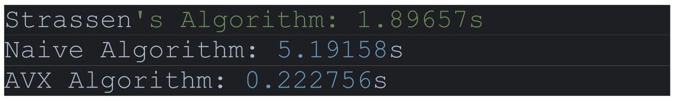


```
Strassen's Algorithm: 1.99301s
Naive Algorithm: 5.17092s
```

Figure 16 - Compiler optimization with SSE SIMD instructions to boost performance

But it does not end there. Intel and AMD CPUs from 2011 onwards support an extension to the x86 ISA named AVX (Advanced Vector Extensions). AVX uses 16 256-bit YMM registers comprised of 2 128-bit registers (YMM is to XMM as x86_64 is to x86 registers), with XMM registers as the lower half of their respective YMM registers, to perform SIMD operations. Both SSE and AVX are specifically made for vector operations, including but not limited to matrices. This approach speeds up matrix multiplication significantly, even by some orders of magnitude compared to the naïve algorithm, although it requires tinkering with the code and having low-level knowledge on how processor cache and memory alignment works. There is the more advanced and newer AVX2, but most of it is irrelevant in this example, as it is mainly focused on floating point arithmetic. Regardless, this example utilizes perks from both AVX and AVX2. Utilizing AVX works on non-power-of-two, non-square matrices.

Since AVX operates on 256-bit registers, the asymptotic computational complexity of the algorithm does not change, but rather the constants that do change, as now the computer processes 8 integers at once, cutting down a considerable amount of operations from N^3 to 0.125N^3. Cache misses are also reduced, as the algorithm does not load columns to the registers (C++ is a row-major order language, so contiguous blocks in memory are the elements per row).



```
Strassen's Algorithm: 1.89657s
Naive Algorithm: 5.19158s
AVX Algorithm: 0.222756s
```

Figure 17 - Compiler optimizations + AVX operations + low-level knowledge on how cache works greatly improve efficiency

The only downsides are that AVX operations are slightly heavier on the CPU—it hammers the CPU more than any other operation, and compatibility issues may arise, but only if we are dealing with decades-old computers.

The compiler used for this paper is G++ and the compiler flags that were used are as follows:

"-O3 -fopenmp -march=native -mtune=native -mavx2 -mfma."

-O3 enables compiler optimization level 3, which allows the compiler to optimize the underlying assembly/machine code to run things faster. It also allows the compiler to take use of AVX registers.

-fopenmp allows multithreading by using OpenMP, a library for multiprocessing, and it allows the compiler to automatically parallelize pieces of code that the compiler thinks are parallelizable without rewriting the code.

-march=native and -mtune=native tunes the compiler to automatically configure to the computer's configuration. Combined with -mavx2 and -mfma, it allows the compiler to use AVX instructions.

D. Coppersmith-Winograd Algorithm

Don Coppersmith and Shmuel Winograd developed an algorithm that further lowers the asymptotic complexity of the previous Strassen's algorithm to $\mathcal{O}(N^{2,3755})$, a substantial decrease in growth rate. It is considered the asymptotically fastest known algorithm to date, with techniques such as asymmetric hashing that further improves the asymptotic complexity to $\mathcal{O}(N^{2,3719})$, which is not exactly a meaningful difference, but it is there.

The thing about the Coppersmith-Winograd algorithm is that it is not so much an algorithm as an existence proof. The algorithm is not used for practical purposes. In fact, the Coppersmith-Winograd algorithm is classified as a "galactic algorithm," which is an algorithm that outperforms any other algorithm for problems that are sufficiently large, so large that it is never used in practice. Another example of a galactic algorithm is the fastest known way to multiple two integers, which uses a 1729-dimensional Fourier transform.

While having a lower asymptotic complexity is generally better on a surface level, the constants hidden by the big O notation is worth noting. Due to sufficiently large constants, it is impractical for galactic algorithms to operate with small inputs. It might only become practical with sufficiently large inputs.

Due to how it was designed, Coppersmith-Winograd algorithm cannot be easily written in a programming language. Nevertheless, it is worth delving into the inner workings of the algorithm. The Coppersmith-Winograd algorithm utilizes tensors and partitions.

One first constructs an algorithm A which given vectors x and y with length N, computes N values of the form $z_k = \sum_{i,j} t_{ijk} x_i y_j$, with $t_{ijk} \in \{0, 1\}$. The values z_k do not have to correspond to entries from a matrix product. *A* is considered a tensor, and A^n is called the n^{th} tensor power of *A*. It is obtained by applying *A* to vectors *x* and *y* of length N^n recursively *n* times. Split *x* and *y* into *N* subvectors of length N^{n-1}, then run A on *x* and *y* as vectors of length N with entries being vectors of length N^{n-1}. Its running time is $\mathcal{O}(r^n)$ with *r* being the number of multiplications performed by *A*.

If A^n can be used to multiply square matrices in $\mathcal{O}(r^n)$ time, then the asymptotic complexity is better bounded the larger the size of the matrix is. In other words, the larger the matrix, the faster the algorithm operates.

Don Coppersmith and Shmuel Winograd introduced techniques that allowed them to effectively choose which variables to set to 0 so that very large matrix products can be computed using A^n. They rely on partitioning the index triples *i*, *j*, $k \in [Q]^n$.

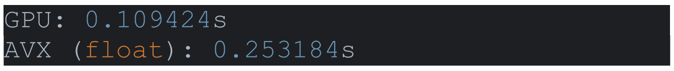
Depending on the underlying algorithm, the partitioning varies and affects the final bound on the complexity of the algorithm. Coppersmith and Winograd obtained a better complexity with $\mathcal{O}(N^{2{,}376})$ with A^2.

It is hard to simplify this algorithm to only a few words for this paper. Data structures that are used in this algorithm such as tensor powers are not practical to implement into a program. There is no readily available C++ implementation of the algorithm as of now.

E. GPU Matrix Multiplication

There is a different and more radical approach to matrix multiplication: do it outside the CPU. The GPU, or Graphics Processing Unit, can do things other than displaying images. GPGPU, or General-Purpose GPU computing, allows code to be executed using the GPU core. Performance will improve significantly with unparalleled parallelism compared to CPUs with a meager number of cores compared to GPUs with thousands of cores on the mid-high end.

GPU computing is harder to implement compared to normal CPU implementations. Additional libraries are needed to support uploading the program to the GPU for it to execute code. For this purpose, an NVIDIA GPU with CUDA (Compute Unified Device Architecture) capability is used. The AVX implementation is compared with the GPU as it is the fastest of the bunch for CPU compute. The algorithm is slightly modified to use single-precision floating point arithmetic to provide an even ground between the GPU and the CPU.

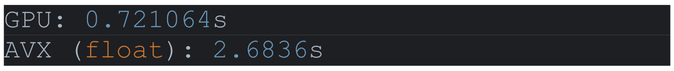


```
GPU: 0.109424s
AVX (float): 0.253184s
```

Figure 18 - CPU vs GPU in 2048x2048 matrix multiplication using a GTX 1660 Ti

Using C++ allows the program to manage the GPU's memory manually, allowing for faster matrix operations and utilizing several tricks to conform to what the GPU is better at computing. GPUs are specifically designed for matrix operations—that is the main data structure used for graphics processing, so it is no surprise that GPUs are faster.

What about a bigger matrix, say 4096 x 4096?

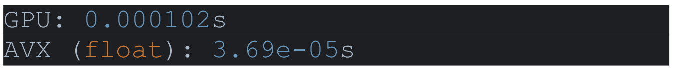


```
GPU: 0.721064s
AVX (float): 2.6836s
```

Figure 19 - CPU vs GPU in 4096 x 4096 matrix multiplication

The benefit of using GPUs is much more obvious now. The CPU's elapsed time is multiplied ten-fold, while the GPU only gets a 7x performance penalty with a much larger matrix. While both get a considerable hit on processing time, the amount it takes for the GPU to multiply the matrix is already much faster to begin with.

Using GPUs are not always better. On smaller matrices, the GPU can be slower than the CPU due to array copy between the system RAM and the GPU memory, object initialization, and other overheads. An example of a simple 8 x 8 matrix multiplication operation shows how GPUs can be slightly worse than CPUs with smaller matrices:



```
GPU: 0.000102s
AVX (float): 3.69e-05s
```

Figure 20 - A simple 8 x 8 matrix multiplication

The GPU is orders of magnitude slower than the CPU with such a small matrix. For smaller matrices, it is better to use the CPU to do the calculation.

IV. CONCLUSION

Computational complexities of algorithms do not paint the whole story. Even when the naïve algorithm has an asymptotic worst-case complexity of $\mathcal{O}(N^3)$, what matters is the constant of said complexity. An algorithm with a time complexity of for example $1.000.000N^3 + 3.000N^2$ and another algorithm with $0{,}0001N^3 + 2N^2$ are both classified as $\mathcal{O}(N^3)$, despite the latter being apparently much faster.

Not only that, but certain tricks can also be used to speed up matrix multiplication that do not change the complexity of the algorithm, such as utilizing multiple threads on the CPU, using more efficient ways to calculate by using instructions and registers specifically designed for vector operations, having knowledge on memory alignment and cache to reduce cache misses and faster memory access, and so on. These tricks do not influence the complexity of the algorithms used, but they speed up operations by doing it much more efficiently.

With that said, certain tricks should not be applied to every situation. Such as with small matrices, the overhead in multithreading outweighs the time saved by parallelizing due to thread spawning overhead. The overhead in copying to the AVX registers for smaller matrices is enough to make it slower than normal algorithms.

Doing matrix operations on a device specifically tailored for matrix operations, like a GPU, is a valid option, provided the appropriate situation. GPUs in general are more power hungry than CPUs, which can be an issue on laptops where battery life matters.

In conclusion, due diligence is needed to determine which approach is best for the job, and while the naïve algorithm is relatively the slowest of the bunch, what matters more is the implementation, not the asymptotic complexity. Don't be discouraged to use the naïve algorithm, unless large datasets are used in which other algorithms and implementations are much better suited for the job.

V. Appendix

Source code for the program used for this paper can be found in this link, which contains all the algorithms used in this paper. The program is entirely written in C++. Python source code is not provided, as it is only for example purposes.

VI. Acknowledgment

This paper would not have been brought to fruition without resources provided by the IF2120 Discrete Mathematics class with the guidance of Fariska Zakhralativa Ruskanda, S.T., M.T. as the lecturer of the author in class.

The author would like to thank other brilliant programmers in the world for sharing their knowledge and code, as without it this paper would not have been complete. Without readily available documentation, the author would not have been able to learn and create a program for this purpose.

References

[1] Ambainis, Adris, Yuval Filmus, François Le Gall, (2014), *Fast Matrix Multiplication: Limitations of the Laser Method*, http://www.cs.toronto.edu/~yuvalf/AmbFilLeG14.pdf [Accessed: Dec. 11, 2022].
[2] Duan, Ran, Hongxun Wu, Renfei Zhou, (2022), *Faster Matrix Multiplication via Asymmetric Hashing*, https://doi.org/10.48550/arXiv.2210.10173 [Accessed: Dec. 11, 2022].
[3] Fawzi, A., Balog, M., Huang, A. *et al. Discovering faster matrix multiplication algorithms with reinforcement learning.* Nature 610, 47–53 (2022). https://doi.org/10.1038/s41586-022-05172-4 [Accessed: Dec. 10, 2022]
[4] Huang, Jianyu & Smith, Tyler & Henry, Greg & van de Geijn, Robert. (2016). *Strassen's Algorithm Reloaded*. 690-701. 10.1109/SC.2016.58 [Accessed: Dec. 11, 2022].
[5] Higham, Nicholas J., (1990), *Exploiting Fast Matrix Multiplication Within the Level 3 BLAS*, https://www.maths.manchester.ac.uk/~higham/papers/high90s.pdf [Accessed: Dec. 11, 2022]
[6] Intel, (2022), *Intel Intrinsics Guide*, https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX,AVX2 [Accessed: Dec. 11, 2022]
[7] Intel, (2015), *Intel Architecture Instruction Set Extensions Programming Reference*, https://www.cs.utexas.edu/~hunt/class/2016-spring/cs350c/documents/Intel-x86-Docs/64-ia-32-architectures-instruction-set-extensions-reference-manual.pdf [Accessed: Dec. 11, 2022]
[8] Landsberg, J. M. *Geometry and Complexity Theory* 169 (Cambridge Univ. Press, 2017) [Accessed: Dec. 10, 2022].
[9] Mahato, Saahil, (2020), *Strassen's Matrix Multiplication Algorithm*, https://medium.com/swlh/strassens-matrix-multiplication-algorithm-936f42c2b344 [Accessed: Dec. 10, 2022].
[10] OpenMP, (2018), *SIMD Directives*, https://www.openmp.org/spec-html/5.0/openmpsu42.html [Accessed: Dec. 10, 2022].
[11] Ståhlberg, Henrik, (2017), *"AVX SIMD in Matrix Multiplication"*, https://codereview.stackexchange.com/questions/177616/avx-simd-in-matrix-multiplication [Accessed: Dec. 10, 2022].
[12] Williams, Virginia Vassilevska, (2014), *Multiplying matrices in $O(n^{2,373})$ time*, http://theory.stanford.edu/~virgi/matrixmult-f.pdf [Accessed: Dec. 11, 2022].

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 11 Desember 2022

[illegible]

Ammar Rasyad Chaeroel
13521136