On the Capabilities of Three Matrix Multiplication Algorithms

Joey Mulé and Chris Bispels

CMSC-441 Algorithms, Dr. Alan Sherman

University of Maryland, Baltimore County

April 19, 2024

Abstract

We examine three matrix multiplication algorithms, and experimentally test their asymptotic runtime and space complexities. This is achieved through completing 5 executions of the algorithms run on randomly generated square matrices for 12 different sizes. We use least regression to find the time constant, and predict which matrix multiplication algorithm is the most time efficient at each matrix size, and discuss the value of using each matrix multiplication algorithm at these sizes by considering their runtime and required space.

Keywords: Computations on Matrices, Coppersmith-Winograd Algorithm, Matrix Multiplication, Matrix Multiplication Algorithm, Strassen's Algorithm

1 Introduction

French mathematician Jaqcues Binet was the first to multiply matrices [4]. Since then, matrix multiplication has become integral across various algorithms and programs the average person uses daily, so a hunt for a more efficient way to multiply matrices began.

Note that for simplicity, letting $n \in \mathbb{N} \setminus \{0\}$, in this paper all matrices are square matrices of nonnegative integers of size $n \times n$, that is, matrices in the set $\mathbb{Z}_{\geq 0}^{n \times n}$, which we will refer to as the set of square natural matrices and denote as $\mathbb{N}^{\times n}$. Note that we will be using the RAM cost model for our asymptotic analysis throughout this paper.

We immediately find that the upper bound for the time a matrix multiplication algorithm takes is $O(n^3)$, as n multiplications are summed n^2 times, and the lower bound is $\Omega(n^2)$, as each entry in the matrices being input needs to be read. Strassen developed Strassen's Algorithm, the first matrix multiplication algorithm that runs faster then the Basic Matrix Multiplication Algorithm. After many years of improvements, we now have matrix multiplication algorithms that run in $O(n^{2.371552})$ with a recently discovered method that is expected to lead to even more improvements in the near future [3].

In this paper, we focus on comparing three algorithms for matrix multiplication, the Basic Matrix Multiplication Algorithm, Strassen's Algorithm, and the Coppersmith-Winograd Algorithm. We find and compare the constant factors of the runtimes of these algorithms from time as a function of matrix size. The matrix size where one matrix multiplication algorithm becomes faster then the other is found for all three combinations of the three matrix multiplication algorithms. Lastly, how well the space required to the algorithm matches the theoretical space required is analyzed.

We hypothesize that at small matrix sizes ($n < 25$), *SAM* will run faster then *BAM*. Additionally, we hypothesize that at large matrix sizes of at least $n = 4500$, *CWA* will run faster then *SAM*. Further, we hypothesize that *BAM* will always be the most space efficient matrix multiplication, and that *SAM* and *CWA* will not have significant space differences until the matrix size where *CWA* becomes faster then *SAM*, at which point *CWA* will be more space efficient.

1.1 Basic Matrix Multiplication Algorithm

The Basic Matrix Multiplication Algorithm, or *BAM*, is the first way anyone learns to multiply matrices. As this is also the definition of matrix multiplication, we omit the history of this algorithm and it's discovery, which is more suited for a history of mathematics textbook or linear algebra class. The method is fairly simple, as for matrices $A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$, $B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{bmatrix}$ it simply computes matrix

$$A \times B = \begin{bmatrix} \sum_{i=1}^{n} a_{1i}b_{i1} & \sum_{i=1}^{n} a_{1i}i2 & \dots & \sum_{i=1}^{n} a_{1i}b_{in} \\ \sum_{i=1}^{n} a_{2i}b_{i1} & \sum_{i=1}^{n} a_{2i}i2 & \dots & \sum_{i=1}^{n} a_{2i}b_{in} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{n} a_{ni}b_{i1} & \sum_{i=1}^{n} a_{ni}i2 & \dots & \sum_{i=1}^{n} a_{ni}b_{in} \end{bmatrix} \tag{1.1}$$

with only the operations seen here, no recursion or anything else. From this we see there are exactly n multiplications and n additions to find the value in ever cell, which happens n^2 times to find the values for all the cells of the matrix, hence we simply conclude this algorithm runs in $\Theta(n^3)$. Note that the space required for this algorithm is simply the output matrix, which is $n \times n$ so it has a space complexity of $O(n^2)$.

1.2 Strassen's Algorithm

Strassen's Algorithm, or *SAM*, is an algorithm discovered by Strassen and published in 1969 in an article titled "Gaussian Elimination is not Optimal", [5]. This algorithm works only on square matrices over rings, as it uses the commutativity of the elements in a matrix to reduce the total number of calculations needed to find each matrix entry. By partitioning the matrices being multiplied into four equal sized parts, that is for $A, B \in \mathbb{N}^{\times n}$ we get eight partitions

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

where each partition is in $\mathbb{N}^{\frac{n}{2} \times \frac{n}{2}}$. Then, basic matrix operations are used to get seven matrices, each with a recursively calling Strassen's algorithm:

$$M_1 = (A_{11}+A_{22})\times(B_{11}+B_{22}),\ M_2 = (A_{21}+A_{22})\times B_{11},\ M_3 = A_{11}\times(B_{12}-B_{22}),$$

$$M_4 = A_{22}\times(B_{21}-B_{11}),\ M_5 = (A_{11}+A_{12})\times B_{22},\ M_6 = (A_{21}-A_{11})\times(B_{11}+B_{12}),$$

$$M_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22}).$$

Lastly, the product of the two matrices is calculated,

$$A \times B = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}. \tag{1.2}$$

Assuming $n = 2^k$, letting $T(n)$ be the recurrence for Strassen's algorithm we find

$$T(n) = \begin{cases} c_1 & n = 2 \\ 7T\left(\frac{n}{2}\right) + c_2 n^2 & n > 2 \end{cases} \tag{1.3}$$

for constants c_1, c_2. By Master Theorem $T(n) \in \Theta\left(n^{\lg 7}\right)$, as $f(n) = c_2 n^2 < n^{\lg 7} = h(n)$ for sufficiently large n. Note that a restriction to *SAM* is that, for smaller n, the constant factor from the extra calculations required leads to *BAM* being faster. This is why good implementations if *SAM* often use *BAM* to solve subproblems when n is sufficiently small. Additionally, note the space requirement according to [2] is $O(n^2)$ for efficient implementations.

1.3 Coppersmith-Winograd Algorithm

The Coppersmith-Winograd Algorithm, or *CWA*, is an algorithm discovered by Coppersmith and Winograd at the Thomas J. Watson Research Center in 1987 and published in 1990 in an article titled "Matrix Multiplication via Arithmetic Progressions" [1]. The construction of this matrix multiplication algorithm is difficult to describe, so we present a simplified explanation based on what Williams presents in [6].

A matrix can be viewed as a vector of vectors, specifically a $n \times n$ matrix can be viewed as a vector of length n, where each entry is an vector of length n. From this, an algorithm "A" can be constructed to compute the n tensor products $z_k = \sum_i^n \sum_j^n t_{ijk} x_i y_j$ where $1 \leq k \leq n$ and $t_{ijk} \in \{0, 1\}$. We can selectively choose values of i, j, k for which $t_{ijk} = 0$ to reduce the number of products that are taken. Note this concepts is not all that different from Strassen's algorithm so far, which partitions the matrix and, instead of multiplying each of the partitions as normal, only requires 7 matrix multiplications (picking one of the t_{ijk}s to be 0). We can abstract the concept, and realize that z_k does not have to be an entry of a matrix product. With this, we can define an algorithm A^q which works on vectors x, y of size n^q as follows:

1. Let x'_1 be the first n^{q-1} entries of x, x'_2 second n^{q-1} entries, and so on with x'_n being the last n^{q-1} entries and we do the same for y.
2. We then find our z_ks, and for each multiplication we use the algorithm A^{q-1} until we reach A^2, at which point the multiplications are completed.

Note that A^n is just the nth tensor power of A.

By being the right x_is, y_js, z_ks to 0, one can compute the matrix product xy while skipping out on many matrix multiplications. With this we have given a high level idea of how CWA works while carefully avoiding going into too much detail so we do not detract from the main focus of the paper.

This results in multiplying square matrices with dimensions $n \times n$ with an asymptotic time complexity of $O(n^{2.376})$. Note that since, like in Strassen's algorithm, we are essentially multiplying subproblems, we obtain a space complexity of $O(n^2)$. Note that this is not the strictest space complexity, however it is the simplest and the one we choose to use for this paper.

2 Theoretical Analysis and Experimental Methods

We now introduce the constants to our asymptotic runtime in preparation for finding our results from the experimental data. We first look at how to find the constant time for a matrix multiplication algorithm based off the asymptotic runtime. Asymptotic runtime of a matrix multiplication algorithm can be represented as $f(n)$, where n is a natural number and $f : \mathbb{N} \to \mathbb{R}^+$ takes an integer n and returns the asymptotic runtime for the matrix multiplication algorithm running on a matrix in $\mathbb{N}^{\times n}$. From this,

$$t = Cf(n) \tag{2.1}$$

where $t \in \mathbb{R}^+$ is the runtime (not asymptotic) of the algorithm, and $C \in \mathbb{R}^+$ is the constant factor. We can then divide both sides by $f(n)$ to get our equation for the constant factor in terms of the runtime (not asymptotic) and the asymptotic runtime:

$$C = \frac{t}{f(n)}. \tag{2.2}$$

We then substitute in the asymptotic runtime for our matrix multiplication algorithms to find the equation for the constant factor in terms of time (not asymptotic) and asymptotic runtime. For BAM,

$$C_{BAM} = \frac{t}{n^3}. \tag{2.3}$$

For SAM,

$$C_{SAM} = \frac{t}{n^{\lg 7}}. \tag{2.4}$$

Lastly for CWA,

$$C_{CWA} = \frac{t}{n^{2.376}}. \tag{2.5}$$

Next, we use this to solve for the matrix size n where one algorithm becomes faster then the other. Taking equation (2.1) and setting it equal for two different matrix multiplication algorithms with constant time C_1, C_2 and asymptotic runtime $g(n), h(n)$ respectively, we find

$$\frac{g(n)}{h(n)} = \frac{C_2}{C_1}. \tag{2.6}$$

We first find the matrix size where SAM runs faster then BAM by taking the first matrix multiplication algorithm to be BAM and second to be SAM from equation (2.6). From this we get

$$\frac{C_{SAM}}{C_{BAM}} = \frac{n^3}{n^{\lg 7}} = n^{3-\lg 7}.$$

By raising both sides to the power of $\frac{1}{3-\lg 7}$ we find our equation for the matrix size where SAM becomes faster then BAM :

$$n = \left\lceil \left(\frac{C_{SAM}}{C_{BAM}}\right)^{\frac{1}{3-\lg 7}} \right\rceil. \tag{2.7}$$

Note that we round up the number on the right side as n needs to be an integer, since it is a matrix size, and since we are interested in finding the smallest matrix size where SAM runs faster then BAM, we need to round up or else we get an n where matrices in $\mathbb{N}^{\times n}$ are multiplied faster by BAM then SAM.

Continuing, we now complete a similar process, now having the first algorithm from equation (2.6) to be SAM and the second to be CWA. We find

$$\frac{C_{CWA}}{C_{SAM}} = \frac{n^{\lg 7}}{n^{2.376}} = n^{\lg 7 - 2.376}$$

By raising both sides to the power of $\frac{1}{\lg 7 - 2.376}$ we find our equation for the matrix size where *CWA* becomes faster then *SAM* :

$$n = \left\lceil \left(\frac{C_{CWA}}{C_{SAM}} \right)^{\frac{1}{\lg 7 - 2.376}} \right\rceil . \tag{2.8}$$

Lastly, we find complete the same process for a third time for *BAM* and *CWA*. We start with equation (2.6), and take the first algorithm to be *BAM* and the second as *CWA*, giving

$$\frac{C_{CWA}}{C_{BAM}} = \frac{n^3}{n^{2.376}} = n^{0.624}.$$

We then raise both sides to the power of $\frac{1}{.624}$ to get our equation for the matrix size where *CWA* becomes faster then *BAM* :

$$n = \left\lceil \left(\frac{C_{CWA}}{C_{BAM}} \right)^{\frac{1}{0.624}} \right\rceil . \tag{2.9}$$

For analysing the space, we find that it is straight forward for *BAM*, as the r^2 value of a linear line of best fit comparing the theoretical space model to the space used will give the necessary information. That is, we will find the coefficient of space for *BAM* in bytes, S_{BAM}, by finding the coefficient of the linear equation for Space, y, as a function of matrix size squared, n^2, according to this equation:

$$y = S_{BAM} n^2. \tag{2.10}$$

For *SAM* and *CWA*, the equation is similar, changing the coefficient to be S_{SAM} and S_{CWA}:

$$y = S_{SAM} n^2, \tag{2.11}$$

$$y = S_{CWA} n^2. \tag{2.12}$$

The method we utilize in order to capture the running time of our algorithms encompass the simple method known as “Timestamping.” Here, in essence, we start a timer right before the algorithm call, then stop the timer after the algorithm has returned. While this method is very efficient, the effectiveness could be hindered by background processes, or any other pieces of data running on memory, and CPU processing.

```
start = time.now();
run_algorithm();
end = time.now();
elapsedtime = end - start;
print("Time:", elapsedtime);
```

The method we employ in regards to analyzing the space of our algorithms consist of running our programs through the well known, ‘valgrind,’ command. Here we are able to not only check our code for memory leaks and errors, but to also determine what, in terms of bytes, our program heap size is.

```
valgrind ./algorithm_program
```

3 Data Analysis and Discussion of Results

We include our code for each of the algorithms in the separate, accompanying document.

For clarity, this section is divided into two subsections, one for the time complexity and one for the space complexity of the algorithms. In the Conclusion section, we provide a more holistic discussion of which algorithm works better for different purposes, considering both the time and space complexities.

3.1 Time Complexity

First, we find the C_{BAM}, C_{SAM}, and C_{CWA} according to equation (2.3), (2.4), and (2.5) respectively for each value of n, shown in Table 1.

n	C_{BAM}	C_{SAM}	C_{CWA}
200	31.11	76.51	675.6
400	35.34	65.61	794.1
600	38.85	91.81	1153
800	34.89	70.65	1111
1000	41.32	80.70	1486
1200	40.69	75.82	1522
1400	58.14	83.18	1863
1600	63.59	69.68	1555
1800	72.98	69.98	1749
2000	70.76	64.16	1680
3000	52.27	79.86	2496
5000	58.21	97.76	3681
AVG	49.84	77.14	1647
SD	14.67	10.22	807.3
SD of Mean	4.236	2.952	233.0

Table 1: Solving equations (2.3), (2.4), and (2.5) from experimental data, with table values in units of 10^{-10} seconds rounded to four significant figures. The standard deviation is computed using Google Sheets STDEV function, and the standard deviation of the mean is computed by dividing the standard deviation by the square root of the sample size $\left(\sqrt{12}\right)$.

Very interestingly, our implementation resulted in rather close values for the constant factors of *BAM* and *SAM*, with the average constant factor for *SAM* being about 1.5 times the the constant factor of *BAM*. Additionally, despite *BAM* running the exact same loops every time and *SAM* having to adjust what is run to make the matrix size divisible by two in the subproblems, *SAM* has standard deviation less then half that of *BAM* as a percent of their respective averages (29.43% for *BAM* and 13.25% for *SAM*). We also note that *CWA* has a very large standard deviation, almost half the the size of the average. As such, unlike the other two algorithms, this method of calculating the constant factor may not be very accurate. This is possibly also because *CWA* is more efficient for selecting different products depending on the matrix size, and how much a matrix size needs to be altered until it becomes a power of 2.

We use the average constant values from Table 1 to find the average n satisfying equations (2.7), (2.8), and (2.9):

$$\left\lceil \left(\frac{C_{SAM}}{C_{BAM}}\right)^{\frac{1}{3-\lg 7}} \right\rceil = 10, \quad \left\lceil \left(\frac{C_{CWA}}{C_{SAM}}\right)^{\frac{1}{\lg 7 - 2.376}} \right\rceil = 1209, \quad \left\lceil \left(\frac{C_{CWA}}{C_{BAM}}\right)^{\frac{1}{0.624}} \right\rceil = 273.$$

As expected, *SAM* quickly becomes faster then *BAM* at computing the product of two square matrices. Despite *CWA* having a better asymptotic runtime then *SAM*, and a significantly faster asymptotic runtime then *BAM*, *CWA* only becomes faster at large then the other algorithms at significantly large matrix sizes. To put it in perspective, $1209^{\lg 7}$, the asymptotic runtime for *SAM* with matrix size 1209, is just over 450 million while $1209^{2.376}$, the asymptotic runtime for *CWA* with matrix size 1209, is a bit over 21 million. This just further demonstrates how large the constant factor for *CWA* is, making it very impractical for most matrix multiplication needs. Through this method of analysis, we do see that in most cases using *SAM* is faster then *BAM*. There is no need to compare at which point *CWA* becomes better then *SAM*, as by this time *SAM* is already far more efficient then *BAM*, and *SAM* is more efficient then *CWA*. However, it is important to note that, as seen in Graph 1, the difference between *SAM* and *CWA* is fairly small, especially when compared to how much more efficient then *BAM* both these algorithms are.

Next, a linear line of best fit was created to estimate the value of the constants for each algorithm's version of equation (2.1) by graphing the linear relationship of t as a function of $f(n)$, the theoretical runtime of the algorithm algorithm, using the Google Sheets LINEST function with the "calculate_b" boolean option set to False. The data from this is included in Table 2.

	BAM	*SAM*	*CWA*
Slope	58.02	96.48	3535
Error	.04953	1.522	133.0
r^2	0.9992	0.9973	0.9847

Table 2: Values calculated to solve equation (2.1) for time as a function of $f(n)$ for each of the three algorithms using the Google Sheets LINEST function. The slope and error are the calculated value of the constant and its error in units 10^{-10} seconds.

We can see the line of best fit for both *BAM* and *SAM* have a very high r^2, with *CWA* having a slightly lower r^2 but still fairly high. From this, we know our data fits fairly accurately to a linear line of best fit, so the asymptotic runtime is a very good estimate for the runtime of the algorithms for square matrices in $\mathbb{N}^{\times n}$ where $200 \leq n \leq 5000$.

We use the constant values from Table 2 to find the solutions $n \in \mathbb{N}$ to the equations (2.7), (2.8), and (2.9):

$$\left\lceil \left(\frac{C_{SAM}}{C_{BAM}}\right)^{\frac{1}{3-\lg 7}} \right\rceil = 15, \quad \left\lceil \left(\frac{C_{CWA}}{C_{SAM}}\right)^{\frac{1}{\lg 7 - 2.376}} \right\rceil = 4224, \quad \left\lceil \left(\frac{C_{CWA}}{C_{BAM}}\right)^{\frac{1}{0.624}} \right\rceil = 725. \tag{3.1}$$

Again, as expected SAM becomes faster then BAM at a fairly small matrix size. However CWA becomes faster then both BAM and SAM at large matrix sizes. The matrix size that CWA becomes faster then SAM is very close to the largest matrix size we tested.

We now compare the constants found from the two different data analysis methods. Note that the values for the time constants of BAM and SAM are reasonable, though only the constant for BAM agrees across the different data analysis methods. This is because the constant found using the LINEST method is within one standard deviation of the constant found using the average method and the LINEST constant is within two standard deviations of the mean from the constant found using the average method. The constant found for SAM using the LINEST method is within two standard deviations of the constant found using the average method, but it is over five standard deviations of the mean greater then the constant found using the average method. From the other perspective, we see that the constants found using the average method are well outside the margin of error from the LINEST methods constant value. Due to how accurately the least squares line approximates the data, it is more reasonable to choose the constants from the LINEST calculating method.

For CWA, there is a stark difference in the constants found, with the LINEST method producing a constant more then twice that of the average method. However, because the standard deviation of from the average method is so large, the constants are still within three standard deviations of one another, so the LINEST found constant is a reasonable value according to the average method. However, the constants do not agree with each other, as they are over 5 standard deviations of the mean away from each other. Additionally, it is interesting to note that the error for the constant found using the LINEST method is half the standard deviation of the mean from the average method. It is clear that neither of these methods estimated the constant very precisely, though the LINEST method did produce a very accurate line of best fit, with an r^2 of 0.9847. From this, it is reasonable to conclude that the LINEST method computed a more accurate constant.

Going forward we will refer to the values found in Table 2 when talking about the constant time of one of the algorithms, and we will use the values from equation (3.1) when discussing at what matrix sizes one algorithm becomes more efficient then the other.

It is interesting to see how impactful the coefficient of the algorithm runtime is at smaller matrix sizes. SAM quickly becomes faster then BAM (15×15 matrices), however CWA only becomes faster then BAM at the significantly larger matrix size of 725×725. Despite having a significantly faster asymptotic runtime, CWA takes a while to actually become faster then BAM. This is due to the large size of the coefficient of CWA, which more then 60 times the size of BAM's coefficient.

3.2 Space Complexity

From equations (2.10), (2.11), and (2.12), we find the values of S_{BAM}, S_{CWA}, and S_{CWA} as the slope for a least squares regression line calculated in Google Sheets using the built in LINEST function. These values are presented in Table 3

	BAM	SAM	CWA
Slope	11.95	236.8	171.6
Error	.1487	12.01	9.109
r^2	0.9985	0.9749	.9726

Table 3: Values for S_{BAM}, S_{SAM}, and S_{CWA} calculated from equations (2.10), (2.11), and (2.12) using the LINEST function in Google Sheets.

Interestingly, has a very strong linear trend for its theoretical space as a function of matrix size squared, with an r^2 value greater then 0.998. This is surprising, as BAM's space depends solely on the individual entries of the matrix's values, so the more larger the integers the more space BAM requires. Due to this, some variance could reasonably be expected in the space requirnments for the different matrix sizes. The reason this is not present is likely that, due to the values in the matrix being large and the immense number of matrix entries, the average space taken by an entry in the matrix is very close to 50 (the median of the possible values), and this holds across all the matrices, so there is minimal variance. Perhaps smaller matrices could be analyzed, or matrices created with different average entry sizes (integers differing in value by a power of 2). However, SAM has a strong linear trend with a r^2 value of just under 0.975. CWA has a slightly weaker linear trend then SAM, though it is still very strong.

If space is the main concern, then without a question BAM is the best choice of matrix multiplication algorithm. The space required is absolutely less then either SAM or CWA for any size matrix, as the constant factor is smaller when all three have space modeled in by an equation of the same variable (the variable being n^2). If deciding between SAM and CWA, the space requirements will be fairly similar, even at large matrix sizes, as both algorithms have a quadratic relationship between matrix size and space required. However, the slope of this relationship is much smaller for CWA, and as such it is the better choice between the two where space is concerned.

4 Conclusion and Open Problems

Comparing the results, it seems somewhat unclear what the best matrix size is to switch from using one matrix multiplication algorithm to another. Equation

(3.1) provides a good general baseline for this. It is important to note that at matrix sizes where BAM is faster, it is clearly the best choice as is also more space efficient. Additionally, as how CWA runs varies depending on matrix size (as does SAM, however SAM varies to a much lesser degree), it is difficult to judge exactly at what size matrices CWA becomes faster then SAM. Further work to help answer this problem could involve more detailed analysis into how SAM and CWA work at specific different matrix sizes, and potentially finding which specific matrix sizes CWA is faster then SAM and vice versa in a range of 200 or so around 4224. A similar question to be answered is; what is the smallest matrix size such that is CWA faster then SAM ?

When wanting to multiply two large square matrices (matrices in $\mathbb{N}^{\times 2000}$ or larger), it is important to consider both your time and space constraints. The runtime of CWA and SAM are relatively close at this point while CWA uses significantly less space. For this reason, depending on the number of matrix multiplications of this size being completed, it may be very worthwhile to use CWA instead of SAM for space considerations, even if it takes slightly longer to complete the matrix multiplication.

In conclusion, we have found that our hypothesis for the matrix sizes that BAM is more efficient then SAM was to loose, with a more accurate maximum matrix size that BAM is more efficient then SAM being 15×15. Our hypothesis for the matrix sizes that CWA runs faster then SAM is supported by our analysis. We also found evidence to support our hypothesis that BAM is the most space efficient matrix multiplication algorithm out of the three. Lastly, we found evidence to reject the hypothesis that CWA and SAM have similar space efficiencies until the matrix size where CWA runs faster then SAM.

Appendix

A Data Tables and Graphs

	BAM		*SAM*		*CWA*	
n	time	space	time	space	time	space
200	0.0249	554,752	0.0221	5,437,252	0.0198	4,042,252
400	0.2262	1,994,752	0.1324	39,532,252	0.1209	28,807,252
600	0.8391	4,394,752	0.5783	48,337,252	0.4601	35,782,252
800	1.787	7,754,752	0.9979	134,137,252	0.8786	99,262,252
1000	4.132	12,074,752	2.133	262,837,252	1.995	194,482,252
1200	7.030	17,354,752	3.343	283,957,252	3.152	205,042,252
1400	15.95	23,594,752	5.653	355,192,252	5.567	258,667,252
1600	26.04	30,794,752	6.890	434,437,252	6.379	321,442,252
1800	42.56	38,954,752	9.631	986,512,252	9.494	718,387,252
2000	56.61	48,074,752	11.87	1,206,637,252	11.71	892,762,252
3000	141.1	108,074,752	46.11	2,017,972,252	45.60	1,450,207,252
5000	727.6		236.9		226.3	

Table 4: We have the average runtime in seconds up to 4 significant figures from 5 trials of three different matrix multiplication algorithms multiplying to matrices in $\mathbb{N}^{\times n}$ for listed values of n, and the space in bytes required by the algorithms to complete these matrix multiplications.

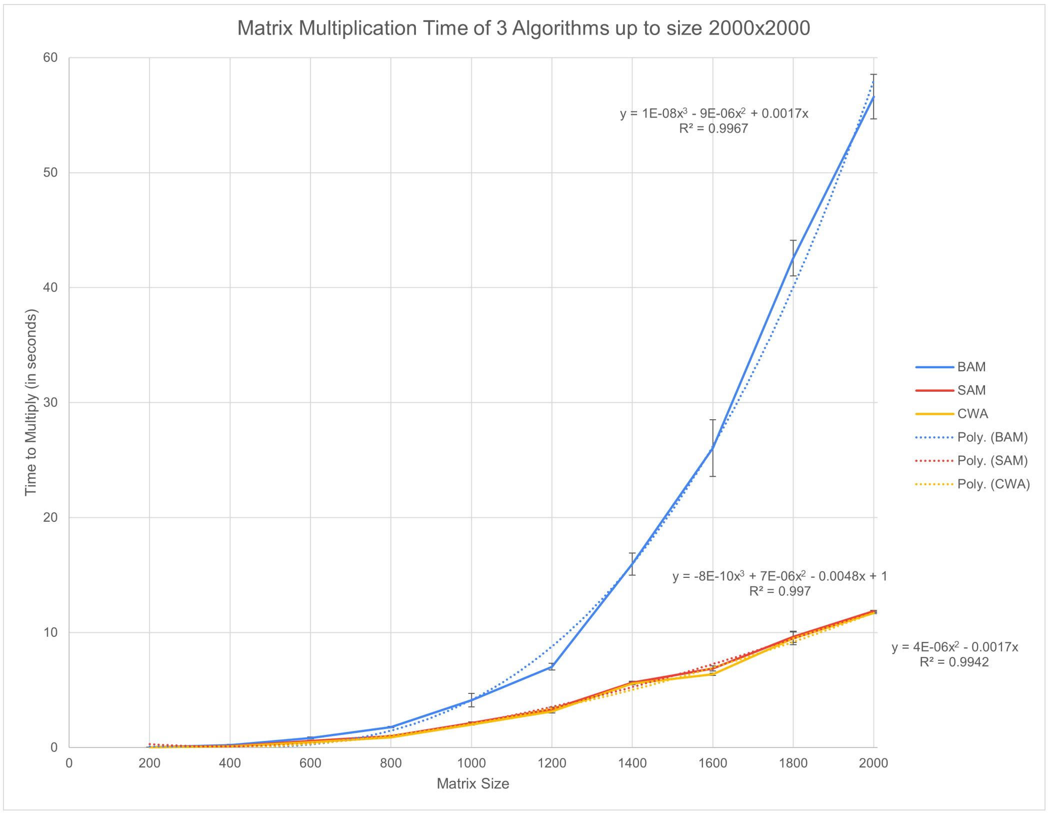

Figure 1: Average runtime in seconds of the three algorithms, *BAM* in red, *SAM* in blue, and *CWA* in yellow.

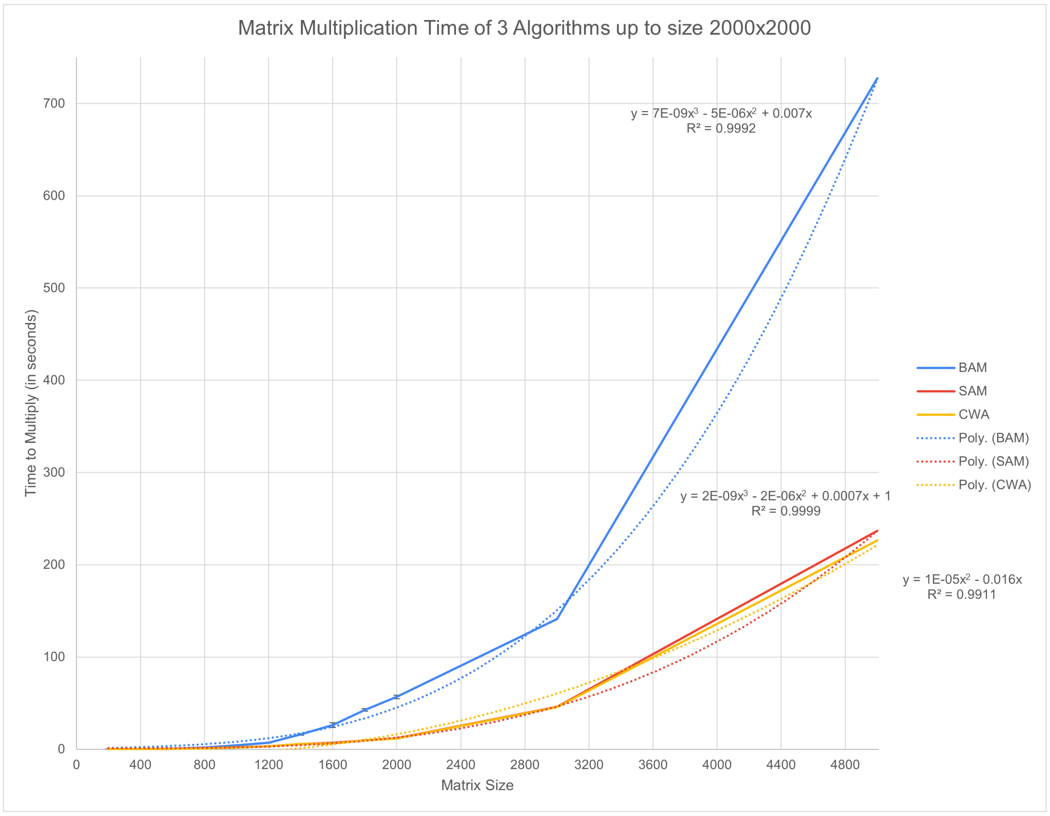

Figure 2: Graph of average runtime in seconds of the three algorithms, BAM in red, SAM in blue, and CWA in yellow, with smaller y axis to show smaller differences in the runtime at smaller matrix sizes.

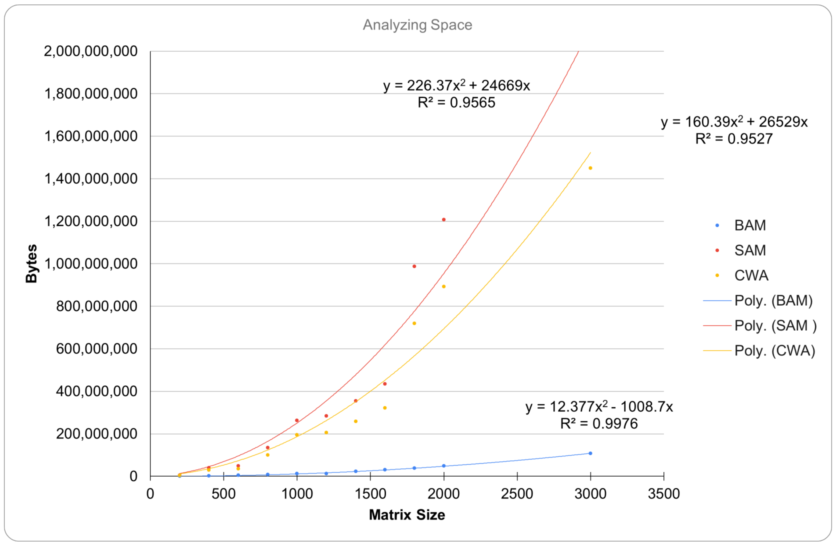

Figure 3: Space used by the algorithm plotted by the size of the square matrices that were being multiplied. Note the data only goes to 3000×3000 matrices, as space data for 5000×5000 sized matrices was not collected.

References

[1] Coppersmith, Winograd; "Matrix Multiplication via Arithmetic Progressions"; Journal of Symbolic Computation 9, 251-280 (1990). https://doi.org/10.1016/S0747-7171(08)80013-2

[2] Gary; "Lecture 1: Introduction and Strassen's Algorithm"; Graduate Algorithms Course, Carnegie Mellon (2018). https://www.cs.cmu.edu/ 15451-f20/LectureNotes/lec01-strassen.pdf

[3] Nadis; "New Breakthrough Brings Matrix Multiplication Closer to Ideal"; https://www.quantamagazine.org/new-breakthrough-brings-matrix-multiplication-closer-to-ideal-20240307/

[4] O'Connor, Robertson; "Jacques Philippe Marie Binet"; MacTutor History of Mathematics Archive, University of St Andrews. https://mathshistory.st-andrews.ac.uk/Biographies/Binet/

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

[6] Williams; "Multiplying matrices in $O(n^{2.373})$ time"; Stanford (2014). https://theory.stanford.edu/ virgi/matrixmult-f.pdf