Algorithm Analysis: A Comparative Study of BAM, SAM and SAMk

Heather DeVal
Department of Computer Science
and Electrical Engineering
University of Maryland
Baltimore County

Hafsa Chaudry
Department of Computer Science
and Electrical Engineering
University of Maryland
Baltimore County

Charles Place
Department of Computer Science
and Electrical Engineering
University of Maryland
Baltimore County

***Abstract*—When solving a complex yet essential mathematical operation such as matrix multiplication, it is typically desirable to implement an algorithm that uses time and space efficiently. In order to research the time and space complexity of various matrix multiplication algorithms, three algorithms were chosen to be compared: the Basic Matrix Multiplication algorithm (BAM), Strassen's algorithm (SAM), and a modified version of Strassen's algorithm with a small problem cutoff (SAMk). For each of the three algorithms, an original implementation was designed utilizing the programming language Python. Testing and recording of the data for all of the algorithms was performed on the UMBC GL server.**

***Keywords*— Algorithm, Matrix Multiplication, Strassen Multiplication, Timing Analysis, Resource Analysis**

I. INTRODUCTION

FROM encrypting communications to controlling robots to representing survey data, matrices and matrix operations are integrated throughout almost every career field. The binary operation of matrix multiplication is a popular linear algebra technique due to its condensed nature of solving mathematical problems. Although the usage of matrices is so high, complex algorithms including basic matrix procedures tend to require a high usage of time and space. In the case of matrix multiplication, the increased amounts of time and space typically come from multiple loops within the algorithm iterating over all the elements within the matrices. Many practical applications of matrix multiplication involve the combination of large matrices and require a rapid outcome. Therefore, the main focus of the experiment was to find the matrix multiplication algorithm that uses time and space most efficiently. In order to do this, the goal was to answer the hypothesis: what can be shown about the time and space usage of three different

implementations of matrix multiplication, specifically the following implementations: Basic Matrix Multiplication (BAM), Strassen's Algorithm (SAM), and a modified version of Strassen's Algorithm referred to as Strassen's Algorithm with a small problem cutoff k (SAMk)? The original implementation of Strassen's Algorithm with a Small Problem Cutoff k (SAMk) brings a new perspective to analyzing the algorithmic approaches of matrix multiplication.

II. Different Matrix Multiplication Algorithms

Three matrix multiplication algorithms were developed, executed, and recorded throughout this experiment in order to provide insight on the differences in usage of time and space between them.

A. Basic Matrix Multiplication Algorithm (BAM)

The Basic Matrix Multiplication Algorithm (BAM), uses the approach of a simple summation in order to generate the product of two matrices. The running time of the Basic Matrix Multiplication Algorithm is $O(n^3)$. Below is the general pseudocode for the Basic Matrix Multiplication algorithm:

$BAM(A[\,][n], B\,[n][\,])$
  $C[\,][\,] = C(n, n)$
  $for\ x\ =\ 0\ to\ n\ -\ 1$
    $for\ y\ =\ 0\ to\ n - 1$
      $total\ =\ 0$
      for w = 0 to z – 1
        total = total + $A_{wi} \times B_{wi}$
      $C_{ij} = sum$
  $return\ C$

B. Strassen's Matrix Multiplication Algorithm (SAM)

In 1969, Volker Strassen created a more efficient algorithm to solve matrix multiplication known Strassen's Algorithm (Strassen Algorithm). Strassen's implementation reduced the running time from the running time of the basic matrix multiplication algorithm, $O(n^3)$, to the running time of his algorithm, $O(n^{\log 7})$. Below is the general pseudocode for Strassen's algorithm:

SAM(a1, a2):

```
n = len(a1)
if n <= 2:
    return BAM(a1, a2)
else:
    a11,a12,a21,a22 = break_into_4_parts(a1)
    b11,b12,b21,b22 = break_into_4_parts (a2)
    //strassens equations avoiding as much copying as possible.
    p1 = SAM(a11, (b12 - b22))
    p2 = SAM((a11+ a12),b22)
    p3 = SAM((a21+ a22), b11)
    p4 = SAM(a22,(b21 -b11))
    p5 = SAM((a11+ a22), (b11 + b22))
    p6 = SAM((a12 - a22), (b21,+b22))
    p7 = SAM((a11 - a21), (b11 + b12))
    //rejoin into the 4 component matrices
    r11 = (((p5+p4)-p2)+p6)
    r12 = (p1+p2)
    r21 = (p3+p4)
    r22 = (((p1+p5)-p3)-p7)
    //rejoin into one
    result = join(r11,r12,r21,r22)
    return result
```

C. *Strassen's Matrix Multiplication Algorithm with a Small Problem Cutoff k (SAMk)*

Strassen's Matrix Multiplication Algorithm with a small problem cutoff k (SAMk) is a modified version of Strassen's original algorithm for matrix multiplication. SAMk was developed as an improvement over SAM due to the notion that the algorithm solves all of its subproblems less than the small problem cutoff k using the BAM algorithm implementation. Below is the general pseudocode for Strassen's Matrix Multiplication Algorithm with a small problem cutoff k (SAMk):

```
global K
SAM(a1, a2):
  n = len(a1)
  if n <= K:
      return BAM(a1, a2)
  else:
```

```
a11,a12,a21,a22 = break_into_4_parts(a1)
b11,b12,b21,b22 = break_into_4_parts (a2)
//strassens equations avoiding as much copying as possible.
p1 = SAM(a11, (b12 - b22))
p2 = SAM((a11+ a12),b22)
p3 = SAM((a21+ a22), b11)
p4 = SAM(a22,(b21 -b11))
p5 = SAM((a11+ a22), (b11 + b22))
p6 = SAM((a12 - a22), (b21,+b22))
p7 = SAM((a11 - a21), (b11 + b12))
//rejoin into the 4 component matrices
r11 = (((p5+p4)-p2)+p6)
r12 = (p1+p2)
r21 = (p3+p4)
r22 = (((p1+p5)-p3)-p7)
//rejoin into one
result = join(r11,r12,r21,r22)
return result
```

III. Related Previous Work

IV. Methods

A. Testing Environment

Throughout the experiment, one computer was used to run and test the data in order to ensure the accuracy and consistency of the results. The UMBC GL server was used to execute the algorithms for reasons of convenience and usability. The computer utilized for testing has the following specifications shown in Table 1 below.

Table 1: Testing Computer Specifications

Processor	Intel Core i7 CPU 2.80 GHz
RAM	16 GB
Operating System	Windows 64 Bit

B. Algorithms

All algorithms were developed in Python and run using Python version 3.7.1. The analyzed matrices were square matrices and their input size, $n \times n$, was determined through a random number generator to generate n from the numpy library of Python. The observed matrix sizes are shown in Table 2 below.

Table 2: Observed Matrix Size

N	Matrix Size
2	2×2
4	4×4
16	16×16
32	32×32
64	64×64
128	128×128
256	256×256
512	512×512
1028	1028×1028

1) Basic Matrix Multiplication (BAM) Implementation

In order to multiply matrix A and matrix B together using the Basic Matrix Multiplication algorithm, the two matrices must be compatible. In this sense, compatibility between matrices is observed if both matrix A and matrix B are organized such that the number of columns in matrix A is equivalent to the number of rows in matrix B. Due to the input parameters being strictly square matrices, no logical checking of compatibility was necessary for BAM. The implementation of BAM can be reviewed in Appendix A-1.

2) Strassen's Algorithm (SAM) Implementation

Strassen's algorithm is conditional on the basis that both matrices A and B to be multiplied together are square matrices. In the case that either matrix does not have the dimensions $2^n \times 2^n$, then matrices A and B cannot be multiplied until the non-square matrix is converted to a square matrix by filling in necessary rows and columns with zero's (Strassen Algorithm). Due to the input parameters being strictly square matrices, there was no requirement to check

the matrices or pad them with zeros. Segmentation, a huge part of Strassen's Algorithm, was required and implemented in order to insure the correct running time of the program. The implementation of SAM can be found in Appendix A-2.

3) Strassen's Algorithm with a Small Problem Cutoff k (SAMk) Implementation

V. Results

A. Optimal Value of k

After data analysis, it was observed that the optimal value of k for the SAMk algorithm was 4. A graph showing the running times of SAMk against SAM using k at a value of 4 is shown below in figure 1.

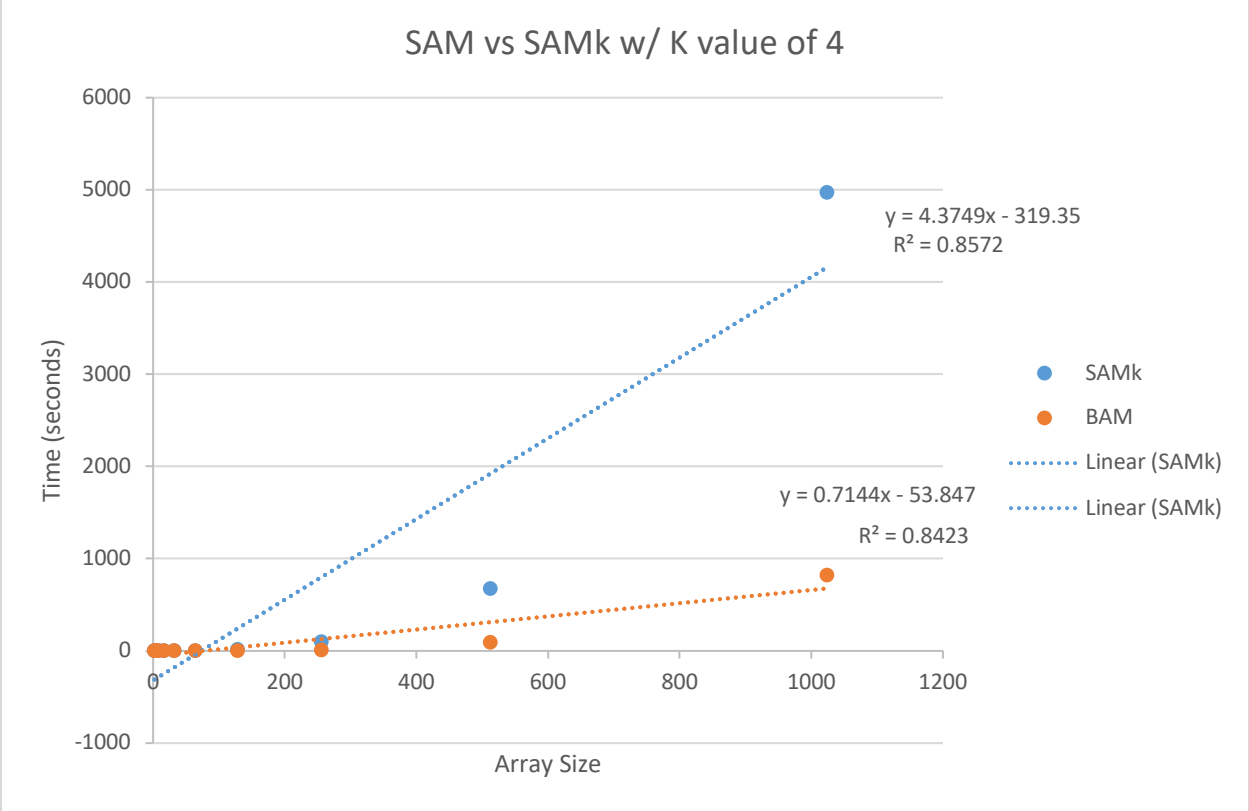

Figure 1: SAM vs SAMk Running Time

B. All Running Times with Log Scale

Due to the exponential increase in all running times as n increased, the algorithms could be observed on a logarithmic scale. A graph showing the running times of each of the algorithms on a logarithmic scale is shown below in figure 2.

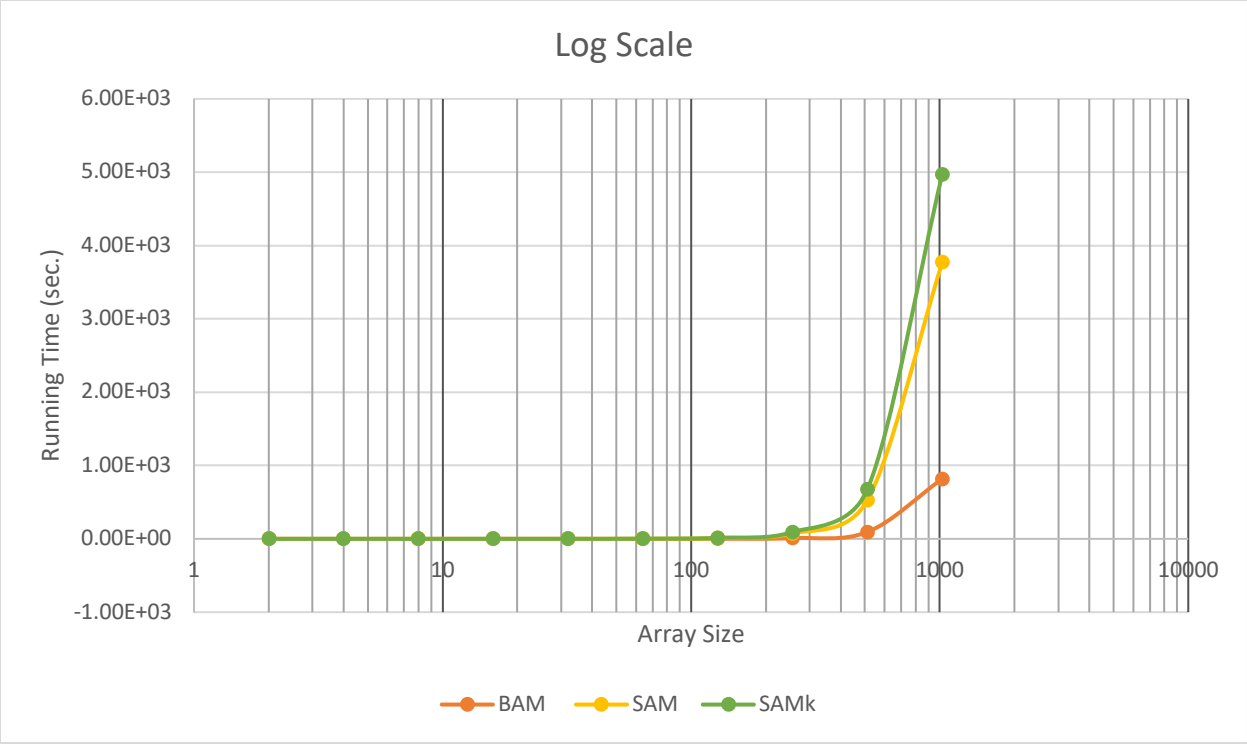

Figure 2: Algorithms on a Logorithmic Scale

VI. Discussion

A. Experiment

B. Memory Management

C. Results

D. Data Interpretation

VII. Conclusion

Add in

Appendix

A. Algorithm Implementations

1) Basic Matrix Multiplication Algorithm

```
def BAM(a1,a2):
    size = len(a1)
    result = [[0 for j in range(0, size)] for i in range(0,size)]
```

```
    for i in range(size):
        for j in range(size):
            for k in range(size):
                result[i][j]+=a1[i][k]*a2[k][j]
  return result
```

2) Strassen's Algorithm

```
# inspired by https://www.programiz.com/python-programming/examples/multiply-matrix
def SAM(a1, a2):
  n = len(a1)
  if n <= 2:
   return BAM(a1, a2)
  else:
     a11,a12,a21,a22 = segmentize(a1)
     b11,b12,b21,b22 = segmentize(a2)
     p1 = SAM(a11, np.subtract(b12, b22))
     p2 = SAM(np.add(a11, a12),b22)
     p3 = SAM(np.add(a21, a22), b11)
     p4 = SAM(a22, np.subtract(b21, b11))
     p5 = SAM(np.add(a11, a22), np.add(b11, b22))
     p6 = SAM(np.subtract(a12, a22), np.add(b21, b22))
     p7 = SAM(np.subtract(a11, a21), np.add(b11, b12))
     r11 = np.add(np.subtract(np.add(p5,p4),p2),p6).tolist()
     r12 = np.add(p1,p2).tolist()
     r21 = np.add(p3,p4).tolist()
     r22 = np.subtract(np.subtract(np.add(p1,p5),p3),p7).tolist()
     result = []
     for i in range(len(r12)):
       result.append(r11[i]+r12[i])
     for j in range(len(r22)):
       result.append(r21[j]+r22[j])
     return result
```

```
def segmentize(A):
  if (len(A[0]) % 2 != 0) or (len(A) % 2 != 0):
        return [[2,2]],[[2,2]],[[2,2]],[[2,2]]
  mlen = len(A)
  middle = (mlen//2)
  A11 = [[A[j][i] for i in range(0,middle)] for j in range(0,middle)]
  A12 = [[A[j][i] for i in  range(middle,mlen)] for j in range(0,middle)]
  A21 = [[A[j][i] for i in range(0,middle)] for j in range(middle,mlen)]
  A22 = [[A[j][i] for i in  range(middle,mlen)] for j in  range(middle,mlen)]
  return A11,A12,A21,A22
```

3) Strassen's Algorithm with a Small Problem Cutoff k

```
def SAMk(a1, a2):
  n = len(a1)
  if n <= 4:
    return BAM(a1, a2)
  else:
      a11,a12,a21,a22 = segmentize(a1)
      b11,b12,b21,b22 = segmentize(a2)

      p1 = SAM(a11, np.subtract(b12, b22))
      p2 = SAM(np.add(a11, a12),b22)
      p3 = SAM(np.add(a21, a22), b11)
      p4 = SAM(a22, np.subtract(b21, b11))
      p5 = SAM(np.add(a11, a22), np.add(b11, b22))
      p6 = SAM(np.subtract(a12, a22), np.add(b21, b22))
      p7 = SAM(np.subtract(a11, a21), np.add(b11, b12))
      r11 = np.add(np.subtract(np.add(p5,p4),p2),p6).tolist()
      r12 = np.add(p1,p2).tolist()
      r21 = np.add(p3,p4).tolist()
      r22 = np.subtract(np.subtract(np.add(p1,p5),p3),p7).tolist()
```

```
    result = []
    for i in range(len(r12)):
      result.append(r11[i]+r12[i])
    for j in range(len(r22)):
      result.append(r21[j]+r22[j])
    return result
```

4) Main Function Call

```
def main()
  runs = 2
  global k
  k = 2
  global scale
  scale = 1
  if(len(sys.argv) > 1):
    runs = int(sys.argv[1])
  df = pd.DataFrame(columns=['Function','Array Size', 'Time', 'Space', 'K'])
  for j in range(1,50):
    print("K      *****    " + str(k) + "\n")
    for i in range(1,11):
      size = 2**i

      print("SIZE      *****    " + str(size) + "\n")
      a1 = np.random.randint(low=0, high = 100, size = (size,size)).tolist()
      a2 = np.random.randint(low=0, high = 100, size = (size,size)).tolist()

      result1 = np.empty([size, size])
      result2 = np.empty([size, size])
      result3 = np.empty([size, size])

      tracemalloc.start()
      snapshot1 = tracemalloc.get_traced_memory()#tracemalloc.take_snapshot()
```

```
    t1 = time.time()
    result1 = BAM(a1,a2)
    t2 = time.time()
    snapshot2 = tracemalloc.get_traced_memory()#tracemalloc.take_snapshot()
    tracemalloc.stop()
   df = df.append({'Function': 'BAM', 'Array Size': size, 'Time': (t2-t1), 'Space':(snapshot2[1]),
  'K':k},
       ignore_index=True)
    tracemalloc.start()
    snapshot3 = tracemalloc.get_traced_memory()
    t3 = time.time()
    result2 = SAM(a1,a2)
    t4 = time.time()
    snapshot4 = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    df = df.append({'Function': 'SAM', 'Array Size': size, 'Time': (t4-t3), 'Space':(snapshot4[1]),
'K':k},
       ignore_index=True)
    tracemalloc.start()
    snapshot3 = tracemalloc.get_traced_memory()
    t3 = time.time()
    result3 = SAMk(a1,a2)
    t4 = time.time()
    snapshot4 = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    df = df.append({'Function': 'SAMk', 'Array Size': size, 'Time': (t4-t3), 'Space':(snapshot4[1]),
'K':k},
       ignore_index=True)
    df.to_csv('regression_stats.csv',index = None, mode='a', header=True)
  scale+=1
  k = 2**scale
```

```
return 0
```

REFERENCES

Cormen, T. (2009). Introduction to algorithms. Cambridge, Mass.: MIT Press.

Kolen, John & Bruce, Phillip. (2001). Evolutionary Search for Matrix Multiplication Algorithms.. 161-165.

"Matrix Multiplication Algorithm." *Wikipedia*, Wikimedia Foundation, 8 Nov. 2019, en.wikipedia.org/wiki/Matrix_multiplication_algorithm.

Python Program to Multiply Two Matrices, www.programiz.com/python-programming/examples/multiply-matrix.

"Strassen Algorithm." *Wikipedia*, Wikimedia Foundation, 18 Sept. 2019, en.wikipedia.org/wiki/Strassen_algorithm.