Fast Matrix Multiplication Algorithms with Symmetry Research Project Progress Report

Jacob Adams

December 4, 2023

Abstract

The project's goal is to adapt Kauer and Moosbauer's recent algorithm finding software [4], so that it can search for matrix multiplication algorithms that have certain symmetry groups that were observed in [2] and [1]. This may lead to faster practical matrix multiplication algorithms, and it can add to our fundamental understanding of efficient computational linear algebra.

1 Introduction to the Problem

Matrix multiplication is a binary operation on two matrices of size $m \times n$ and $n \times p$ to produce a 3rd matrix of size $m \times p$. It is a widely used operation in both mathematics and computer science. It's broad application spans many different areas of both fields such as AI, 3d graphics and many more. It makes sense then, that improving the running time of this algorithm has been a major area of research.

For square matrices of size $n \times n$ The result of this operation can easily be found with time complexity $O(n^3)$. Although it is possible to improve this.

The focus of this project is to design algorithms that aid in finding new ways to multiply matrices faster.

For this project we will mostly be looking at matrix multiplication over the field $\mathbb{F}_2$. Where $\mathbb{F}_2$ is the field over 2 elements. This means all entries in matrices will be either 0 or 1.

2 Background Research

2.1 How to Find Faster Multiplication Algorithms

The naive algorithm for the general $m \times n$ multiplied by $n \times p$ matrix is easy to implement in code and is widely used. However, for certain matrix sizes there exist certain ways of combining the matrix entries such that it requires less multiplications than the naive algorithm would.

Being able to find the result of a multiplication of just one size may not seem that useful. However, it turns out to be very useful. This is because matrix multiplication can also be defined recursively.

This is done by subdividing a large matrix into smaller matrices. Once this is done each sub matrix can be treated as a single entry of a smaller matrix and multiplied. Separately before being recombined by multiplying out the smaller matrices. This recursive step can be repeated for as many layers is necessary. Below is an example on a two 4×4 matrices with only 1 layer of recursion.

2.1.1 Example of Recursive Definition of Matrix Multiplication

We will use the tradition multiplication scheme for a matrix in this example. For two matrices A, B we want to compute the matrix C where $C = AB$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

We define 8 separate, 2×2 matrices A_{ij} and B_{ij} $\forall i, j \in \{0, 1\}$

$$A_{ij} = \begin{bmatrix} a_{1+2i,1+2j} & a_{1+2i,2+2j} \\ a_{2+2i,1+2j} & a_{2+2i,2+2j} \end{bmatrix}$$

$$B_{ij} = \begin{bmatrix} b_{1+2i,1+2j} & b_{1+2i,2+2j} \\ b_{2+2i,1+2j} & b_{2+2i,2+2j} \end{bmatrix}$$

Define two new matrices A' and B'

$$A' = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

$$B' = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Compute C' where $C' = A'B'$

$$C' = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C' = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

This gives us 4 new 2×2 matrices C_{ij} $\forall i, j \in \{1, 2\}$

$$C_{i,j} = A_{i1}B_{1j} + A_{i2}B_{2j}$$

We can then get the result of the original multiplication $C = AB$ from

$$C = \begin{bmatrix} (C_{1,1})_{11} & (C_{1,1})_{12} & (C_{1,2})_{12} & (C_{1,1})_{12} \\ (C_{1,1})_{21} & (C_{1,1})_{22} & (C_{1,2})_{21} & (C_{1,2})_{22} \\ (C_{2,1})_{11} & (C_{2,1})_{12} & (C_{2,2})_{12} & (C_{2,1})_{12} \\ (C_{2,1})_{21} & (C_{2,1})_{22} & (C_{2,2})_{21} & (C_{2,2})_{22} \end{bmatrix}$$

2.1.2 Strassen's Algorithm

In 1969 Volker Strassen discovered a new method for 2×2 matrix multiplication requiring only 7 multiplications instead of the original 8. This new method was faster than the traditional, naive method. Such an algorithm was, until that point, widely assumed impossible.

Since then, other more efficient schemes have been discovered for larger matrices using a variety of methods such as those mentioned in [4] which found faster methods for (4,4,5) and (5,5,5) (m,n,p) matrix multiplications. More progress has also recently been made by using machine learning in AlphaTensor's recent discoveries as seen in [3].

2.2 Strategy for Representing Multiplication Schemes

We will call a set of additions and multiplications to produce a correct multiplication result a matrix multiplication scheme. Examples of multiplication schemes include the mentioned naive algorithm and Strassen's algorithm.

2.2.1 Using Tensors to Represent Matrix Multiplication

Matrix multiplication can be represented by a tensor. For example, as mentioned in [4], matrix multiplication can be represented as:

$$\sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{p} a_{i,j} \otimes b_{j,k} \otimes c_{k,i}$$

Where $a_{x,y}$, $b_{x,y}$ and $c_{x,y}$, refer to matrices of the correct dimensions having a 1 at position (x, y) and zeros everywhere else.

2.2.2 Rank-1 Decompositions for Representing Multiplication Schemes

This on it's own is not all that helpful for searching for different algorithm, but it turns out that a multiplication method can be expressed through a series of rank-1 terms that sum to the tensor mentioned above.

For example, the naive algorithm for a 2×2 multiplied by a 2×2 requires 8 of these summands and follows directly from our definition of the tensor above.

2.2.3 Example of $(2, 2, 2)$ Multiplication Scheme

$$\{a_{i,j} \otimes b_{j,k} \otimes c_{k,i} : \forall i, j, k \in \{1, 2\}\}$$

While the multiplication scheme/tensor decomposition above is pretty obvious from our definitions, some schemes are less obvious.

2.3 Looking for Better Schemes Through Tensor Decompositions

We want to look for schemes that reduce the number of multiplications required. As each rank-1 summand corresponds to exactly 1 multiplication we are looking for the scheme with least number of summands.

While there are many approaches you could take for looking for these rank-1 decompositions. It makes sense to try to start with a valid tensor decomposition representing a multiplication scheme and then reduce the number of these summands. One way to do this is to performing flips and reductions on the tensor representing the matrix multiplication scheme as seen in [4]. This can be thought of as exploring a graph with of multiplication schemes. It is then possible to look for better multiplication schemes by performing random walks on this graph.

3 Project Progress

In this section we will discuss the progress made so far on the project and evaluate it against the timeline set out in the specification.

3.1 Completion of Re-Implementing Original Algorithm

The algorithm described in the original paper [4] has been re-implemented successfully looking specifically over the field of $\mathbb{F}_2$. This was completed before the deadline set out in the project timeline.

3.2 Finding the Multiplication Scheme for Strassen's Algorithm

The code has been tested and was able to find a 7 summand multiplication scheme for $(2, 2, 2)$ (on $\mathbb{F}_2$). The code was also able to quickly find other better multiplication schemes such as a 23 summand multiplication scheme for matrix multiplication for 3×3 matrices.

3.3 Modifying the Original Algorithm

After this the next step in the project is to begin to understand how we can modify this algorithm to potentially improve what it can find in "reasonable time".

In the original algorithm all summands are stored and dealt with separately. Our goal is to improve the performance this could potentially be done by reducing the number of summands the algorithm considers at any one time. One way of doing this is by looking at a symmetry of the summands.

For this part of the project we will only be considering matrix multiplication of the form of an $n \times n$ multiplied by a $n \times n$.

The symmetry we will look at first is for matrices A, B, C of size $n \times n$ for some integer n where you have three summands in the form:

$$A \otimes B \otimes C + B \otimes C \otimes A + C \otimes A \otimes B$$

The restriction to (n, n, n) matrix multiplication is a necessary restriction as matrices A,B and C need to be of the same sizes to be swapped in this way.

We will now define some functions to explore and manipulate these symmetries.

3.3.1 Definition of an Exchange

Fix n as an integer. Let T be the set of 3D tensors of size $n^2 \times n^2 \times n^2$ on $\mathbb{F}_2$.

We want to define a function $\pi \colon T \to T$ such that for matrices A,B,C of size $n \times n$ the rank 1 summand $A \otimes B \otimes C$

$$\pi(A \otimes B \otimes C) = B \otimes C \otimes A$$

We will define this function on the set of basis vectors

$$\{a_{u,v} \otimes b_{w,x} \otimes c_{y,z} : u, v, w, x, y, z \in \{1...n\}\}$$

Where $a_{x,y}$, $b_{x,y}$ and $c_{x,y}$, refer to $n \times n$ matricies having a 1 at position (x, y) and zeros everywhere else.

We define the function as follows:

$$\pi(a_{u,v} \otimes b_{w,x} \otimes c_{y,z}) = a_{w,x} \otimes b_{y,z} \otimes c_{u,v}$$

This can be extended from the basis vectors by:

$$\pi(\sum_{u,v,w,x,y,z=0}^{n} k_{u,v,w,x,y,z} a_{u,v} \otimes b_{w,x} \otimes c_{y,z}) = \sum_{u,v,w,x,y,z=0}^{n} k_{u,v,w,x,y,z} \pi(a_{u,v} \otimes b_{w,x} \otimes c_{y,z})$$

$$\pi(\sum_{u,v,w,x,y,z=0}^{n} k_{u,v,w,x,y,z} a_{u,v} \otimes b_{w,x} \otimes c_{y,z}) = \sum_{u,v,w,x,y,z=0}^{n} k_{u,v,w,x,y,z} a_{w,x} \otimes b_{y,z} \otimes c_{u,v}$$

For a high level understanding, this function allows us to "rotate" the 3 matrices contained within a rank-1 summand. Note that π is also a linear map.

3.3.2 Proving That You Don't Need More Than 3 Exchanges

Proof. Need to show:

$$\pi^3(t) = t$$

Let $t = \sum_{u,v,w,x,y,z=0}^{n} k_{u,v,w,x,y,z} a_{u,v} \otimes b_{w,x} \otimes c_{y,z}$ Therefore:

$$\pi^3(t) = \pi^3(\sum_{u,v,w,x,y,z=0}^{n} k_{u,v,w,x,y,z} a_{u,v} \otimes b_{w,x} \otimes c_{y,z})$$

$$\pi^3(t) = \pi^2(\sum_{u,v,w,x,y,z=0}^{n} k_{u,v,w,x,y,z} a_{w,x} \otimes b_{y,z} \otimes c_{u,v})$$

$$\pi^3(t) = \pi(\sum_{u,v,w,x,y,z=0}^{n} k_{u,v,w,x,y,z} a_{y,z} \otimes b_{u,v} \otimes c_{w,x})$$

$$\pi^3(t) = \sum_{u,v,w,x,y,z=0}^{n} k_{u,v,w,x,y,z} a_{u,v} \otimes b_{w,x} \otimes c_{y,z}$$

$$\pi^3(t) = t$$

□

This proof makes sense intuitively as by swapping the matrices in rank-1 tensors 3 times, you get back to the starting tensor.

It follows that from repeatedly applying π to a tensor t will produce at most 3 unique tensors.

3.3.3 Using a Tensor to Represent a Set 3 With Symmetry

We want to be able to define a function that can use one summand to represent atmost three others. We want a function $\varphi(t)$ such that for a rank 1 summand $A \otimes B \otimes C$

$$\varphi(A \otimes B \otimes C) = A \otimes B \otimes C + B \otimes C \otimes A + C \otimes A \otimes B$$

$$\varphi(A \otimes B \otimes C) = A \otimes B \otimes C + \pi(A \otimes B \otimes C) + \pi^2(A \otimes B \otimes C)$$

For any tensor t

$$\varphi(t) = t + \pi(t) + \pi^2(t)$$

Note that over $\mathbb{F}_2$ a single tensor of the form $A \otimes A \otimes A$ can also be represented by $\varphi(A \otimes A \otimes A)$. This is because.

$$\varphi(A \otimes A \otimes A) = A \otimes A \otimes A + \pi(A \otimes A \otimes A) + \pi^2(A \otimes A \otimes A)$$

$$\varphi(A \otimes A \otimes A) = A \otimes A \otimes A + A \otimes A \otimes A + A \otimes A \otimes A$$

As $1 + 1 + 1 = 1$ in $\mathbb{F}_2$

$$\varphi(A \otimes A \otimes A) = A \otimes A \otimes A$$

We will call the tensor used to represent the 3 summands the representative tensor.

3.3.4 The Basic Multiplication Scheme for $(2,2,2)$ Can be Represented By a Symmetry

We can represent the naive multiplication scheme for $(2,2,2)$ using this symmetry with only 4 representative tensors. Leading to a total rank of 12, although 4 are redundant due to the case mentioned above.

3.3.5 Proof φ is a Linear Map

We will prove that φ is a linear map to help with later proofs.

Proof. Need to show:

$$\varphi(\alpha v + w) = \alpha\varphi(v) + \varphi(w)$$

Proof:

$$\varphi(\alpha v + w) = \alpha v + w + \pi(\alpha v + w) + \pi^2(\alpha v + w)$$
$$\varphi(\alpha v + w) = \alpha v + w + \pi(\alpha v) + \pi(w) + \pi^2(\alpha v) + \pi^2(w)$$
$$\varphi(\alpha v + w) = \alpha v + w + \alpha\pi(v) + \pi(w) + \alpha\pi^2(v) + \pi^2(w)$$
$$\varphi(\alpha v + w) = \alpha(v + \pi(v) + \pi^2(v)) + w + \pi(w) + \pi^2(w)$$
$$\varphi(\alpha v + w) = \alpha\varphi(v) + \varphi(w)$$

□

3.3.6 Using Symmetries With the Original Algorithm Operations

The original flip graph is explored through two operations on the summands. Those are flips and reductions. We need to prove that both of these operations still work on summands representing 3.

3.3.7 Proof of Correctness For Flips

This is almost identical to a proof found in the original paper [4]. We will use this later to help prove that flips still work on a symmetry.

Proof. Need to show:

$$A \otimes B \otimes C + A \otimes B' \otimes C' = A \otimes (B + B') \otimes C + A \otimes B' \otimes (C' - C)$$

Proof:

$$A \otimes (B + B') \otimes C + A \otimes B' \otimes (C' - C) = A \otimes B \otimes C + A \otimes B' \otimes C' + A \otimes B' \otimes C - A \otimes B' \otimes C$$
$$= A \otimes B \otimes C + A \otimes B' \otimes C'$$

This is true for all permutations of A B and C □

3.3.8 Proof Flips Still Work on Symmetry

We want to be able to perform a flip on tensor representatives and preserve the correctness of the multiplication scheme. We will perform flips on representatives in the same way we did for normal flips. we claim that flips preserve the correctness of the scheme. Below is a proof that this is true.

Proof. Need to show:

$$\varphi(A \otimes B \otimes C) + \varphi(A \otimes B' \otimes C') = \varphi(A \otimes (B + B') \otimes C) + \varphi(A \otimes B' \otimes (C' - C))$$

Proof:

$$\varphi(A \otimes B \otimes C) + \varphi(A \otimes B' \otimes C') = \varphi(A \otimes B \otimes C + A \otimes B' \otimes C')$$

Using previous result $A \otimes B \otimes C + A \otimes B' \otimes C' = A \otimes (B + B') \otimes C + A \otimes B' \otimes (C' - C)$

$$\varphi(A \otimes B \otimes C) + \varphi(A \otimes B' \otimes C') = \varphi(A \otimes (B + B') \otimes C + A \otimes B' \otimes (C' - C))$$
$$\varphi(A \otimes B \otimes C) + \varphi(A \otimes B' \otimes C') = \varphi(A \otimes (B + B') \otimes C) + \varphi(A \otimes B' \otimes (C' - C))$$

So flips still work even on the group □

3.3.9 Proof Reductions Work on Symmetry

We want to be able to look at a set of representative tensors and reduce them in the same way. Below is a proof of correctness.

Proof. As we know that reductions on normal summands work. We know:

$$\sum_{i \in I} A_i \otimes B_i \otimes C_i = \sum_{i \in I \setminus \{t\}} A_i \otimes B_i \otimes (C_i + \alpha_i \beta_i C_t)$$

So it directly follows that:

$$\varphi(\sum_{i \in I} A_i \otimes B_i \otimes C_i) = \varphi(\sum_{i \in I \setminus \{t\}} A_i \otimes B_i \otimes (C_i + \alpha_i \beta_i C_t))$$

$$\sum_{i \in I} \varphi(A_i \otimes B_i \otimes C_i) = \sum_{i \in I \setminus \{t\}} \varphi(A_i \otimes B_i \otimes (C_i + \alpha_i \beta_i C_t))$$

□

3.3.10 Evaluation of Current Progress with Proofs

So far, progress of understanding and proving features of the modified algorithm has been on track. This is good as it means there is more time for writing efficient implementations and actually looking for better multiplication schemes.

3.4 Development Of New Random Walk Program Using Symmetry

Because representative tensors can be flipped and reduced in the same way as their normal counterparts we can use the instead with minimal modifications to the old algorithm. The only real immediate modifications required are the initial input multiplication scheme and adding an extra step to expand the representatives into a valid multiplication scheme for testing and an output.

It's worth mentioning that the tensor ranks in this modified program are going to be lower than in the original algorithm due to only needing one out of 3 representative summands in a symmetry. However, this does not mean the actual rank or indeed the number of multiplications required is 3 times lower.

In fact for a $(2, 2, 2)$ matrix multiplication problem. The best we can hope to do using this symmetry is 3 representative tensors often leading to rank 9. Therefore this algorithm is not very useful on the $(2, 2, 2)$ problem. The question is whether it is useful on larger problems.

3.5 Defining one Extra Operation: Change of Representative

Despite having reduced the number of rank-1 summands being considered at any one time by the algorithm. It is sometimes helpful to allow the algorithm to at random change which tensors it considers.

We can define a new operation called the change of representative that switches the representative tensor. This is entirely equivalent to the exchange function π earlier.

We will see later that giving the algorithm this capability greatly increases what it can find and was a necessary addition.

3.5.1 Issues with the walk getting stuck

Sometimes while the algorithm tried to perform a flip. It could get stuck looking for a valid flip. This would happen because the conditions required for a flip were not met. To fix this, each time the algorithm looks for a flip, it has a chance to also perform a change of representative.

This lead to the following psuedocode for the random walk

```
START
    X = GET_BASIC_ALGORITHM(n,m,p)
    X' = REDUCE_TO_SYMMETRY(X)
    LOOP:
        //One step of the random walk
        RESULT = CAN_REDUCE(X)
        IF RESULT != FAIL THEN
            X = REDUCE(X, I)
        ELSE
            X = MAKE_FLIP_AND_CHANGE_OF_REPRESENTATIVES(X)
        END
    ENDLOOP
END

MAKE_FLIP_AND_CHANGE_OF_REPRESENTATIVES(X)
    LOOP:
        IF RANDOM() > 0.5 THEN
            PERFORM_CHANGE_OF_REPRESENTATIVE()
        ENDIF
        T1 = SELECT_RANDOM_SUMMAND_FROM(X)
        T2 = SELECT_RANDOM_SUMMAND_FROM(X/{T2})
        IF CAN_FLIP(T1, T2) THEN
            FLIP(T1, T2)
            RETURN
        ENDIF
    ENDLOOP
RETURN

CAN_REDUCE(X)
    FORALL SUBSETS I OF X
        //The conditions for reduction are
        //For all I. The set of one of the 3 matricies
        //used to construct a term must have dimension 1
        //
        //The set of one of the other matricies used
        //to construct a term must have a linear dependancy
        IF I MEETS THE CONDITIONS FOR REDUCTION THEN
            RETURN I
        ENDIF
    NEXT I
RETURN FAIL

REDUCE(X)
    X = X\{I} U (REDUCED SET OF I)
END
```

3.6 Development Of Exhaustive Search Version

The program can also run an exhaustive search trying to explore the whole graph. It does this by starting at the node representing the original matrix multiplication algorithm and then exploring all adjacent nodes adding them to a stack if they are not already on it. It then pops the first node from the stack and repeats.

```
START
S <- Stack of Nodes
```

```
A = GET_BASIC_ALGORITHM(n,m,p)
A' = REDUCE_TO_SYMMETRY(A)
S.add(A')
WHILE S.isEmpty() = FALSE DO
    n = S.pop()
    T = EMPTY
    T = (Adjecent Algorithms To n By Flips)
    T = T U (Adjecent Algorithms To n By Change of Representatives)
    T = T U (Adjecent Algorithms To n By Reductions)
    FOR t IN T DO
        IF S.contains(t) = FALSE THEN
            S.add(t)
        ENDIF
    NEXT t
ENDWHILE
END
```

3.7 Implementation in Java

While faster implementations of these algorithms will eventually be created in C/C++. These algorithms have currently been implemented in Java. Writing these in Java for now means we can take advantage of the improved error checking and easier memory management. It allows us to focus on the ideas of the algorithm more before porting it into C/C++.

3.8 Findings Without Change of Representative

Using the algorithm described above exhaustively exploring the flip graph with symmetry (without a change of representative operation) leads to discovering only 23 multiplication methods in total.

3.8.1 Screenshot of Algorithm Findings Without a Change In Representative Operation

```
jacobadams@Jacobs-MacBook-Air fastmatrixmultiplicationsymmetryGraph % javac *.java
jacobadams@Jacobs-MacBook-Air fastmatrixmultiplicationsymmetryGraph % java FastMatrixMultiplication
===== TEST CASE POST SYMMETRY =====
======= VALID =======
Explore Node
Progress: 0.0%
Explore Node
Progress: 12.5%
Explore Node
Progress: 20.0%
Explore Node
Progress: 21.428571428571427%
Explore Node
Progress: 28.571428571428573%
Explore Node
Progress: 31.25%
Explore Node
Progress: 35.294117647058826%
Explore Node
Progress: 41.1764705882353%
Explore Node
Progress: 47.05882352941177%
Explore Node
Progress: 52.94117647058823%
Explore Node
Progress: 52.63157894736842%
Explore Node
Progress: 55.0%
Explore Node
Progress: 60.0%
Explore Node
Progress: 65.0%
Explore Node
Progress: 66.66666666666667%
Explore Node
Progress: 68.18181818181819%
Explore Node
Progress: 72.72727272727273%
Explore Node
Progress: 77.27272727272727%
Explore Node
Progress: 81.81818181818181%
Explore Node
Progress: 86.36363636363636%
Explore Node
Progress: 86.95652173913044%
Explore Node
Progress: 91.30434782608695%
Explore Node
Progress: 95.65217391304348%
RESULT SIZE: 23
jacobadams@Jacobs-MacBook-Air fastmatrixmultiplicationsymmetryGraph %
```

3.9 Findings On the Graph With A Change of Representative

Using the same algorithm but giving it the capability to randomly change a set of summands representative drastically increases the size of the graph from the original multiplication scheme for $(2, 2, 2)$ matrix multiplication. There has not been enough time yet to fully explore the size of this graph but it is planned to do so.

3.9.1 Screenshot of Algorithm Findings With a Change In Representative Operation

```
===== TEST CASE POST REDUCTION =====
======= VALID =======
Did add: true
===== TEST CASE POST REDUCTION =====
======= VALID =======
Did add: false
==== FOUND REDUCTION! ====
==== FROM ====
<(+a_0,0  +a_0,1  ) ⊗ (+b_1,1  ) ⊗ (+c_1,0  +c_1,1  )> Z_3
<(+a_0,1  +a_1,1  ) ⊗ (+b_0,1  +b_1,0  +b_1,1  ) ⊗ (+c_0,0  )> Z_3
<(+a_0,0  +a_0,1  ) ⊗ (+b_1,1  ) ⊗ (+c_0,0  +c_0,1  +c_1,1  )> Z_3
<(+a_0,0  +a_0,1  +a_1,1  ) ⊗ (+b_0,0  +b_0,1  +b_1,1  ) ⊗ (+c_0,0  +c_1,1  )> Z_3

==== TO ====
<(+a_0,1  +a_1,1  ) ⊗ (+b_0,1  +b_1,0  +b_1,1  ) ⊗ (+c_0,0  )> Z_3
<(+a_0,0  +a_0,1  ) ⊗ (+b_1,1  ) ⊗ (+c_0,0  +c_0,1  +c_1,0  )> Z_3
<(+a_0,0  +a_0,1  +a_1,1  ) ⊗ (+b_0,0  +b_0,1  +b_1,1  ) ⊗ (+c_0,0  +c_1,1  )> Z_3
```

3.10 Finding A Reduction With the Random Walk Algorithm

After finding a reduced multiplication scheme exhaustively. The random walk algorithm was tested. Initially there were issues with getting the algorithm to find anything. This turned out to be because of the algorithm getting stuck while trying to perform a flip (the problem mentioned in 3.5.1). Originally the algorithm randomly decided to choose between either a flip or a change of representative but this would lead to it sometimes getting stuck when flipping. The code was changed to always flip but in between it trying to find a flip it has the chance to change representative. Below is a screenshot of the modified flip code.

3.10.1 Screenshot of the Modified Flip Code

```
     */
    public void makeFlip()
    {

        ArrayList<Integer> potentialFlip = new ArrayList<>();

        RankTensor xTensor = null;
        RankTensor yTensor = null;

        int xIndex = -1;
        int yIndex = -1;

        while (potentialFlip.isEmpty())
        {

            if (Math.random() < 0.5) //Random chance to change representative
            {
                changeRepresentitive();
            }



            //Generate 2 unique and nearly evenly distributed numbers
            xIndex = randomInt(tensors.size());
            yIndex = randomInt(tensors.size()-1);
            if (yIndex >= xIndex)
            {
                yIndex++;
            }

            xTensor = tensors.get(xIndex);
            yTensor = tensors.get(yIndex);

            //Verify flip can take place
            if (RankTensor.areMatrixEqual(xTensor.a, yTensor.a))
            {
                potentialFlip.add(1);
            }
            if (RankTensor.areMatrixEqual(xTensor.b, yTensor.b))
            {
                potentialFlip.add(2);
            }
            if (RankTensor.areMatrixEqual(xTensor.c, yTensor.c))
            {
                potentialFlip.add(3);
            }
        }
```

After these modifications the random walk was able to successfully find reductions. Below is a screenshot of a reduction the random walk found.

3.10.2 Screenshot of Random Walk Algorithm Output

```
jacobadams@Jacobs-MacBook-Air fastmatrixmultiplicationsymmetryGraph % javac *.java
jacobadams@Jacobs-MacBook-Air fastmatrixmultiplicationsymmetryGraph % java FastMatrixMultiplication
====== RUN RANDOM WALK ON (2, 2, 2) ======
(+a_0,0  ) ⊗ (+b_0,0  ) ⊗ (+c_0,0  )
(+a_0,0  ) ⊗ (+b_0,1  ) ⊗ (+c_1,0  )
(+a_0,1  ) ⊗ (+b_1,0  ) ⊗ (+c_0,0  )
(+a_0,1  ) ⊗ (+b_1,1  ) ⊗ (+c_1,0  )
(+a_1,0  ) ⊗ (+b_0,0  ) ⊗ (+c_0,1  )
(+a_1,0  ) ⊗ (+b_0,1  ) ⊗ (+c_1,1  )
(+a_1,1  ) ⊗ (+b_1,0  ) ⊗ (+c_0,1  )
(+a_1,1  ) ⊗ (+b_1,1  ) ⊗ (+c_1,1  )

====== REDUCE TO SYMMETRY =======
<(+a_0,0  ) ⊗ (+b_0,0  ) ⊗ (+c_0,0  )> Z_3
<(+a_0,0  ) ⊗ (+b_0,1  ) ⊗ (+c_1,0  )> Z_3
<(+a_0,1  ) ⊗ (+b_1,1  ) ⊗ (+c_1,0  )> Z_3
<(+a_1,1  ) ⊗ (+b_1,1  ) ⊗ (+c_1,1  )> Z_3

======= TEST SYMMETRY REDUCTION ======
======= VALID =======

====== START WALK =======
Done
===== METHOD WITH RANK 3 (0 REDUCTIONS): =====
<(+a_0,1  +a_1,0  ) ⊗ (+b_0,1  +b_1,0  ) ⊗ (+c_0,0  +c_1,1  )> Z_3
<(+a_1,0  +a_1,1  ) ⊗ (+b_0,1  +b_1,0  +b_1,1  ) ⊗ (+c_0,1  +c_1,1  )> Z_3
<(+a_0,0  +a_0,1  ) ⊗ (+b_0,0  +b_0,1  +b_1,0  ) ⊗ (+c_0,0  +c_1,0  )> Z_3

===== TEST CASE POST REDUCTION =====
======= VALID =======
^C%
jacobadams@Jacobs-MacBook-Air fastmatrixmultiplicationsymmetryGraph %
```

3.11 Verifying The Results

To verify the results of the algorithm a function was implemented to test a multiplication scheme for correctness. It works by expanding all of the symmetry summands and then summing all the terms in a tensor. It then compares this sum to the sum achieved by the original algorithm. If they are the same it returns true, otherwise it returns false and prints out the incorrect summed tensor.

This was helpful for debugging as you could call this function after any step the algorithm takes. As we know reduction to symmetry, flips, reductions and change of representatives should all preserve this summed tensor this test could be used on almost all of the algorithms operations atleast at a high level.

3.11.1 Screenshot of Some of the Code Used To Test The Multiplication Schemes

```
public boolean testValidity()
{
    int[][][][][][] result = constructResult(this);
    int[][][][][][] testAgainst = constructResult(getBasicMethod(tensors.get(0).a[0].length, tensors.get(0).a.length, tensors.get(0).b[0].length));
    boolean success = true;
    StringBuilder sb = new StringBuilder();
    for (int ci = 0; ci < result[0][0][0][0].length; ci++)
    {
        for (int cj = 0; cj < result[0][0][0][0][0].length; cj++)
        {
            sb.append("c_");
            sb.append(ci);
            sb.append(",");
            sb.append(cj);
            sb.append(" = (");
            for (int ai = 0; ai < result.length; ai++)
            {
                for (int aj = 0; aj < result[0].length; aj++)
                {
                    for (int bi = 0; bi < result[0][0].length; bi++)
                    {
                        for (int bj = 0; bj < result[0][0][0].length; bj++)
                        {
                            if (result[ai][aj][bi][bj][ci][cj] == 1)
                            {
                                sb.append("a_");
                                sb.append(ai);
                                sb.append(",");
                                sb.append(aj);

                                sb.append(" * b_");
                                sb.append(bi);
                                sb.append(",");
                                sb.append(bj);

                                sb.append(") + (");
                            }
                            if (result[ai][aj][bi][bj][ci][cj] != testAgainst[ai][aj][bi][bj][ci][cj])
                            {
                                success = false;
                            }
                        }
                    }
                }
            }
            sb.append(")");
            sb.append("\n");
        }

    }
    if (success == false) { System.out.println(sb.toString());}
    return success;
```

4 Planning What's Next and Project Management

4.1 Progress Compared to Original Timeline

So far progress has been on track with the original timeline laid out in the specification. The only real exception to this is the delay in writing this progress report.

4.2 Updated Risks

The risks of this project have stayed mostly the same. With the only real change being that Risk 1 "Not discovering any practical use of the discoveries in [2] and [1]" (not finding ways to use the observed symmetry) is no longer a problem.

4.2.1 Identified Risks

Below is a list of the risks, originally identified and any modifications to that list as the project has progressed.

1. Not discovering any practical use of the discoveries in [2] and [1]. This is no longer a risk as practical uses have been found and implemented.

2. Practical issues with implementing the algorithm in efficient C++ code. Having never used C++ before I will need to learn this language to do this project.

3. Not gaining access to any large computer resources. This would cause issues as it would restrict how many multiplication schemes the project can explore.

4. Not finding any new multiplication methods. This would happen if the project successfully discovers new ways to explore multiplication schemes but this does not lead to any new discoveries.

4.2.2 Risk Solutions

Below is a list of plans to help mitigate and reduce the chances of the risks mentioned.

1. Risk 1 is no longer a risk.

2. To reduce the likelihood of this risk I will make sure to learn c++ well in advance of my actual implementation. This risk is reduced further by the fact I already know similar languages such as C and Java which gives me confidence. However if this risk still arises I could switch to another language such as C if necessary.

3. To minimise the chance of this risk I will make sure to request resources well in advance and keep multiple options open. If however, I still fail to gain access to any resources I could use my own personal computer running for much longer to try and achieve some level of useful results.

4. If this happens I may have time to modify the algorithm and try again or look at multiplication schemes for different sizes of the input matrices.

4.3 Whats Next

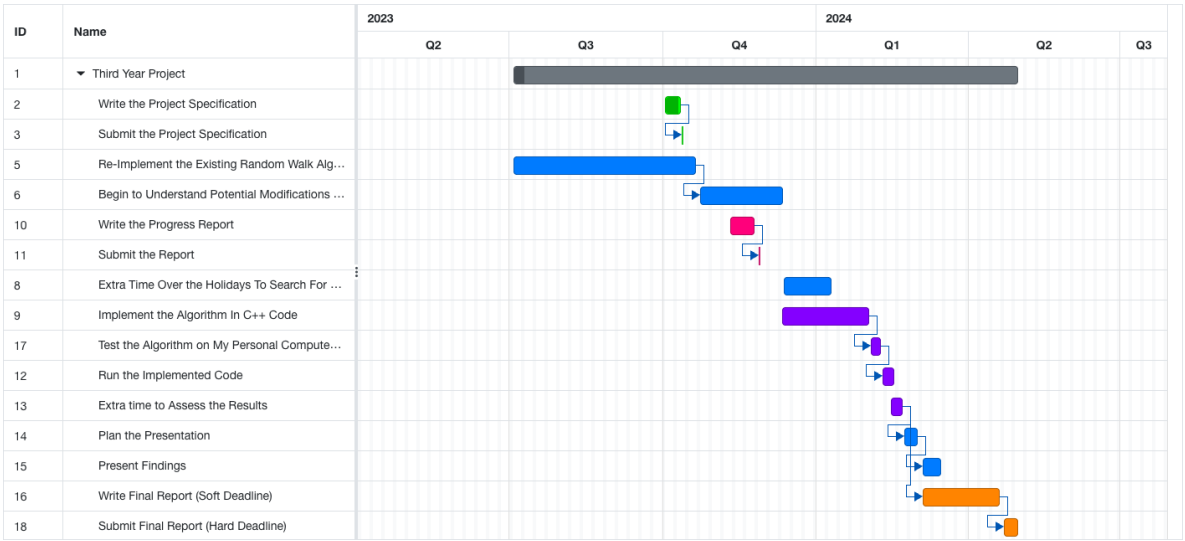
The plan now is to try this modified algorithm on larger matrix multiplication problems. To do this it will be required to look for potential speedups in the specific implementation and perhaps even slightly in the high level description. This will allow us to more efficiently search for these larger schemes and will lead on nicely to the implementation of this algorithm in C/C++.

After the algorithm has been implemented efficiently we can begin to search for multiplication schemes with this symmetry. There is a lot of risk as previously mentioned associated with this step so the specifics of what the algorithm looks for will depend largely on it's performance.

4.4 Updated Timeline

The timeline for the project has remained mostly the same as the project is on track.

4.4.1 Gantt Chart



4.5 Legal, Ethical and Social Issues

The project has not required any external human involvement and does not plan to. So nothing is required.

References

[1] Grey Ballard, Christian Ikenmeyer, Joseph M Landsberg, and Nick Ryder. The geometry of rank decompositions of matrix multiplication ii: $3\times$ 3 matrices. *Journal of Pure and Applied Algebra*, 223(8):3205–3224, 2019.

[2] Luca Chiantini, Christian Ikenmeyer, Joseph M Landsberg, and Giorgio Ottaviani. The geometry of rank decompositions of matrix multiplication i: $2\times$ 2 matrices. *Experimental Mathematics*, 28(3):322–327, 2019.

[3] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[4] Manuel Kauers and Jakob Moosbauer. Flip graphs for matrix multiplication. *arXiv preprint arXiv:2212.01175*, 2022.

Fast Matrix Multiplication Algorithms with Symmetry Research Project Specification

Jacob Adams

December 4, 2023

Abstract

The project's goal is to adapt Kauer and Moosbauer's recent algorithm finding software [4], so that it can search for matrix multiplication algorithms that have certain symmetry groups that were observed in [2] and [1]. This may lead to faster practical matrix multiplication algorithms, and it can add to our fundamental understanding of efficient computational linear algebra.

1 Introduction to Project Problem

1.1 Problem Overview

Matrix multiplication is a binary operation on two matrices to produce a 3rd matrix. It is a widely used operation in both mathematics and computer science. It's broad application spans many different areas of both fields such as AI, 3d graphics and many more. It makes sense then, that improving the running time of this algorithm has been a major area of research.

1.2 Current Progress on the Problem

The naive algorithm for the general nxm multiplied by mxp matrix is easy to implement in code and is widely used. However, in 1969 Volker Strassen discovered a new method for 2x2 matrix multiplication. This new method was faster than the traditional, naive method. Such an algorithm was, until that point, widely assumed impossible.

Since then, other more efficient schemes have been discovered for larger matrices using a variety of methods such as those mentioned in [4] which found faster methods for (4,4,5) and (5,5,5) (n,m,p) matrix multiplications. More progress has also recently been made by using machine learning in AlphaTensor's recent discoveries as seen in [3].

2 Project Objectives

My project is going to look for faster matrix multiplication algorithms by looking at how we can adapt Kauer and Moosbauer's random walk algorithm to potentially improve it's efficiency. This will hopefully allow us to explore more multiplication schemes with this, increasing the chance of a useful discovery.

In this section I will break down this requirement into a series of objectives. The objectives are broken down into essential and extension objectives of the project.

2.1 Essential Objectives

This is the list of objectives that must be completed for the project to be successful.

1. Try to define new types of reductions/flips based on discoveries from papers [2] and [1] and begin to understand what kind of reductions and flips could potentially be implemented in code efficiently. If this fails I will try to explore other potential adaptations of the algorithm (see the risk evaluation section for more information).
2. Adapt the Kauer and Moosbauer's algorithm to work with these new reductions/flips.
3. Implement this adapted algorithm in efficient C++ code. The code must be efficient to have any chance of finding something that has not already been found.
4. Run this algorithm on large computers and assess the results for different matrix multiplication sizes.

2.2 Extension Objectives

This is a list of some objectives which could potentially enhance the success of the project but are not necessary.

1. Look for other adaptations to expand what we can look for to potentially discover other better multiplication schemes.
2. Look for other ways to improve the speed of the algorithm on our hardware. Potentially using the GPU to perform some computations or looking more deeply into clever uses of the hardware to get some level of speedup.

3 Intended Methodology to Use

The traditional waterfall method will be used to develop this software. This is because the size of the actual code-base will be quite small but each section will require careful planning to maximise efficiency and ensure correctness. Projects like this tend to be better suited to the more traditional plan based methods such as waterfall.

This project will also involve a lot of research and discovery. So having time to develop a clear algorithm is a necessity. The planned timeline of development of the project is explained more under "Project Timeline".

4 Project Timeline

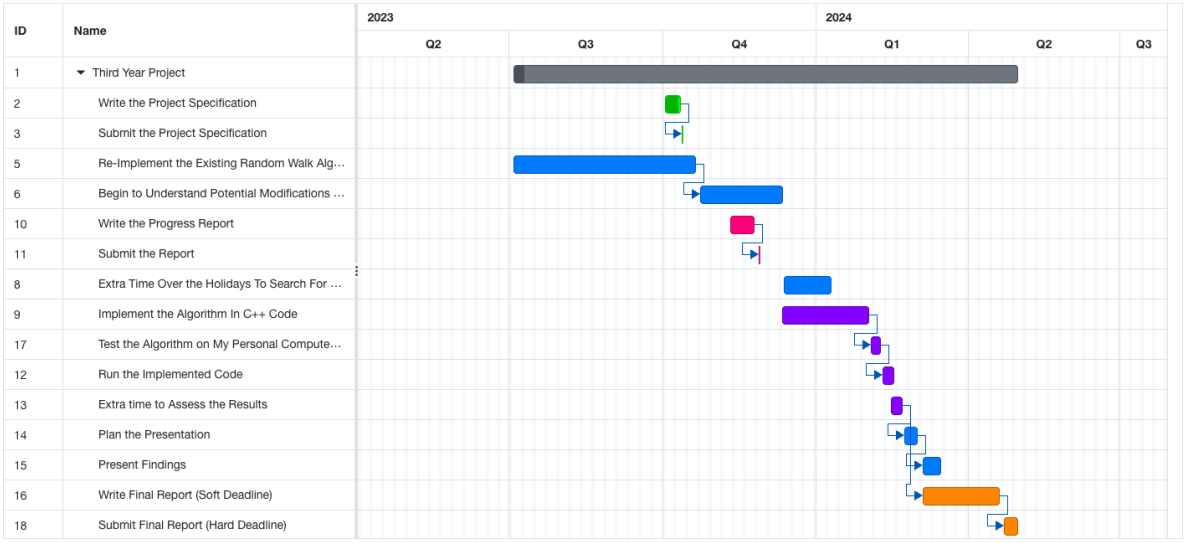
Below is an outline of the project timeline. It takes into consideration other commitments and coursework deadlines that I will have this year and makes sure that I have enough time to complete each part of the project. It also allocates extra time should it be necessary over holidays to ensure the completion of this project for parts of the project which the timescale is harder to predict. I've also included the 4 major submission milestones (specification, progress report, presentation and final report).

4.1 Justification of Plan

As seen in the Gantt chart below, I will first re-implement the algorithm described in [4] to ensure I understand it correctly. This will be implemented in a language I am comfortable with (Java). While this is going on I will make sure to leave time for writing the progress report which will be started early and kept up to date to make the best use of time.

After and during this I will begin to look for potential ways to adapt the code to improve it's running time. Once this has been achieved I can begin to design, implement the modified algorithm in C++ code. I will then go onto testing the algorithm thoroughly before running the code. I will then begin planning both my presentation and final report. I have assumed the "worst case" for the date of my presentation making sure I have plenty of time to plan it well.

4.2 Gantt Chart



5 Testing Plan

Testing plays a crucial role in verifying the correctness of a project. This is all the more important with this project as I will most likely have allocated computer resources for only a limited amount of time. Therefore, I must be certain that the algorithm and implementation is correct and stable before

using these resources. Because of this, testing must will be performed beforehand on my personal computer.

In the waterfall methodology, testing plays a crucial role in the development of a project. After the development of the code the entire project must be tested. I will also adopt testing each module of code separately after it has been written.

The software should be thoroughly tested before being run on large computers as to not waste computer resources running incorrect code.

Each function should be tested individually to ensure correctness. I will need to construct a wide range of test data and expected outputs in a test plan to do this. Likewise each sub-section of code should also be tested individually to ensure correctness.

On completion of the codes development, the program should initially be tested on small matrices to ensure that it correctly finds a reduction into Strassen's algorithm which is known to be optimal for 2x2 matrix multiplication. This will provide a good test as if the program fails to "re-discover" this or "discovers" some false 6 rank-1 decomposition then I know that the program is not working as intended.

6 Risk Evaluation

This section will outline various risks of this project and discuss how these risks can be avoided and mitigated.

6.1 Identified Risks

Below is a list of the risks identified.

1. Not discovering any practical use of the discoveries in [2] and [1].
2. Practical issues with implementing the algorithm in efficient C++ code. Having never used C++ before I will need to learn this language to do this project.
3. Not gaining access to any large computer resources. This would cause issues as it would restrict how many multiplication schemes the project can explore.
4. Not finding any new multiplication methods. This would happen if the project successfully discovers new ways to explore multiplication schemes but this does not lead to any new discoveries.

6.2 Risk Solutions

Below is a list of plans to help mitigate and reduce the chances of the risks mentioned.

1. If this is the case I can explore other potential avenues to look for adaptations of the algorithm. Some of these adjacent problems may be easier to practically implement.
2. To reduce the likelihood of this risk I will make sure to learn c++ well in advance of my actual implementation. This risk is reduced further by the fact I already know similar languages such as C and Java which gives me confidence. However if this risk still arises I could switch to another language such as C if necessary.
3. To minimise the chance of this risk I will make sure to request resources well in advance and keep multiple options open. If however, I still fail to gain access to any resources I could use my own personal computer running for much longer to try and achieve some level of useful results.
4. If this happens I may have time to modify the algorithm and try again or look at multiplication schemes for different sizes of the input matrices.

7 Resources Required

This project will require a number of different resources to be able to complete the objectives.

1. Computer and/or laptop to develop the software on. I already own both so this will make the development of the project much easier.
2. Code editor and compiler (Visual Studio Code)
3. I will need to use some software for organising and keeping tracking of the different development stages and iterations of my project (version control). I will choose to use a private GitHub repository for this as I'm already comfortable with the system and it meets my needs perfectly.
4. Large computers to run the code looking for the better multiplication schemes. I will explore a couple of different avenues for this to increase the chance of finding something I can use.

8 Legal, Social and Ethical Issues

In this project I will be looking for better multiplication scheme finding algorithms than ones that already exist. I will ensure that all code developed is my own but I will make sure to give the correct credit when adapting pseudo-code from other papers.

This research project does not require working with other people. Therefore, there are no immediate ethical or social issues for this project.

References

[1] Grey Ballard, Christian Ikenmeyer, Joseph M Landsberg, and Nick Ryder. The geometry of rank decompositions of matrix multiplication ii: $3\times$ 3 matrices. *Journal of Pure and Applied Algebra*, 223(8):3205–3224, 2019.

[2] Luca Chiantini, Christian Ikenmeyer, Joseph M Landsberg, and Giorgio Ottaviani. The geometry of rank decompositions of matrix multiplication i: $2\times$ 2 matrices. *Experimental Mathematics*, 28(3):322–327, 2019.

[3] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[4] Manuel Kauers and Jakob Moosbauer. Flip graphs for matrix multiplication. *arXiv preprint arXiv:2212.01175*, 2022.