Verify the Correctness of Matrix Multiplication Algorithm With Freivald's Algorithm

Adyaksa Wisanggeni 13517091
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
adyaksa.wisanggeni@gmail.com

***Abstract*—Matrix multiplication is one of the most widely discussed algorithm in the computer science community. Many proposed algorithm and theorem are already proposed to decrease its complexity. With a deep mathematical insight and various supporting paper for every proposed algorithm, it would need a long time to understand the paper, more so to approve or disprove the proposed algorithm. In this paper, we will try to check whether a multiplication algorithm is valid or not using the Freivald's algorithm and compare it with another verifier algorithm.**

***Keywords*—Matrix, Matrix Multiplication, Divide and Conquer, Freivald's Algorithm**

I. Introduction

Matrix is one of the most widely used mathematical concept in computer science. We can model a problem to its matrix representation and manipulate it using mathematic tools. For example, we can model a graph with an adjacency matrix and find its complement easily by using the matrix. Another example is the uses of linear algebra in data science.

One of the most commonly matrix manipulation is to multiply it with another matrix. Matrix multiplication algorithm is one of the most common topics to be researched in computer science. Many algorithms and optimization are proposed to decrease its complexity. To check the proposed algorithm, we need to check whether the proposed algorithm find the correct solution in better time than current algorithm. In this paper, we will use Freivald's algorithm to determine the correctness of matrix multiplication and compare it with various matrix multiplication correctness verifier algorithm.

II. Basic Theory

A. Matrix

Matrix is defined as a collection of number that arranged in rows and columns. Basic operations in matrix that will be used in this paper are addition, scalar multiplication, transposition, and matrix multiplication.

- Addition operation in matrix is defined as follows

$$(\mathrm{A}+\mathrm{B})_{\mathrm{i,j}} = A_{i,j} + B_{i,j}$$

Where A and B is m-by-n matrices and $1 \le \mathrm{i} \le \mathrm{m}, 1 \le \mathrm{j} \le \mathrm{n}$

- Scalar multiplication c in matrix $\mathrm{A}_{\mathrm{i,j}}$ is defined as multiplying every entry in matrix with c, or more formally

$$(\mathrm{cA}_{\mathrm{i,j}}) = c.A_{i,j}$$

- Transposition in matrix $\mathrm{A}_{\mathrm{i,j}}$ is defined as turning it into matrix $\mathrm{A}_{\mathrm{j,i}}$, or more formally

$$(\mathrm{A}_{\mathrm{i,j}})^T = \mathrm{A}_{\mathrm{j,i}}$$

- Matrix multiplication m-by-n matrix A with n-by-p matrix B is defined as follows

$$\mathrm{AB} = \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{n} a_{i,r}b_{r,j}$$

B. Brute Force

Brute force technique is defined as enumerating every possible sequence and check for the most optimum solution in the enumeration. Brute force technique is usually easy to implement but have a bigger complexity than other possible solution. This technique is good for verifying the optimality of other proposed solution.

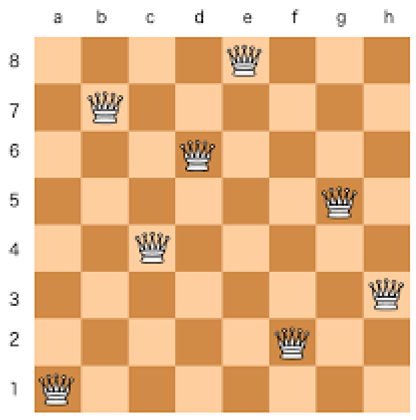


Figure 1 N-Queen problem

C. *Divide and Conquer*

Divide and Conquer technique is an approach that recursively break down a solution into multiple sub-problem until it become simple enough to be solved directly. Problem that can be solved with this technique must have an independent subproblem, or one subproblem must not depend on other subproblem to be solved directly.

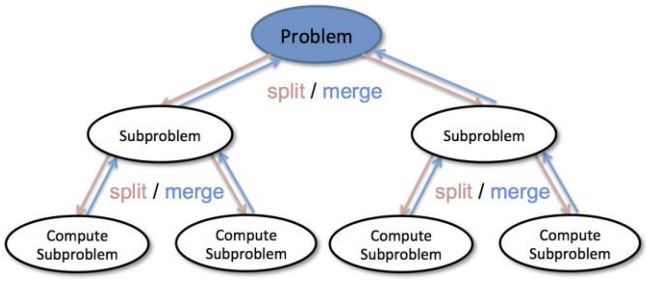


Figure 2 Divide and Conquer visualization (Taken from https://medium.com/cracking-the-data-science-interview/divide-and-conquer-algorithms-b135681d08fc)

D. *Master Theorem*

In the analysis of algorithm, Master Theorem for Divide-and-Conquer recurrences provides an asymptotic analysis complexity of the given recurrences. If T(n) defined as the total time for the algorithm with an input size n, and f(n) defined as the amount of time taken at the top level of recurrences, master theorem can provide an asymptotic analysis for T(n) with the following form

$$\mathrm{T(n)} = \mathrm{a}\left(\mathrm{T}\left(\frac{\mathrm{n}}{b}\right)\right) + cn^d$$

To get the asymptotic complexity of divide-and-conquer algorithm using master theorem, we can divide the algorithm by its T(n)

- If $\mathrm{a} < \mathrm{b^d}$, then the complexity is $\mathrm{O}(\mathrm{n^d})$
- If $\mathrm{a} = \mathrm{b^d}$, then the complexity is $\mathrm{O}(\mathrm{n^d}\, logn)$
- If $\mathrm{a} > \mathrm{b^d}$, then the complexity is $\mathrm{O}(\mathrm{n}^{\log_b a})$

III. Matrix Multiplication Verifier Algorithm

In the following subtopic, we will discuss how to verify whether a matrix is the result of multiplication of 2 other given matrix. Formally, given 3 matrix A, B, and C, we will check whether the following equation is true

$$\mathrm{AB} = \mathrm{C}$$

To ease the calculation of our complexity, we will use the same dimension for each matrix, namely n-by-n.

A. *Brute Force*

Our first approach to verify the correctness of a matrix multiplication algorithm is by using brute force method. Brute force method for this algorithm is to calculate the multiplication of A and B and check whether it's equals to C or not. To calculate the given matrix, we will use the straight-forward definition of matrix multiplication operation.

For example, we define A, B, and C as 2-by-2 matrix with the following element

$$\mathrm{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \mathrm{B} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}, \mathrm{C} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

To check whether $\mathrm{AB} = \mathrm{C}$, we will check whether all of the following equation are true

$$\begin{aligned} \mathrm{c}_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\ c_{22} &= a_{21}b_{11} + a_{22}b_{21} \end{aligned}$$

Because in the definition it will have 3-nested loop, this approach will have the complexity of $\mathrm{O}(\mathrm{n}^3)$

B. *Strassen Algorithm*

Our second approach to verify the correctness of a matrix multiplication algorithm is by using Strassen Algorithm to calculate the result of the multiplication directly. Strassen Algorithm uses Divide-and-Conquer approach. The basic idea of Strassen Algorithm is to recursively minimize the use of multiplication and adding more addition operation. This algorithm assumes that multiplication operation uses more time than addition operation.

For example, we define A, B, C as 2^{d}-by-2^{d} matrix, and $\mathrm{A_{ij}}, B_{ij}, C_{ij}$ as $2^{\mathrm{d-1}}$-by-$2^{\mathrm{d-1}}$ submatrix of A, B, and C respectively. We can separate matrix A, B, C as follow

$$\mathrm{A} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \mathrm{B} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \mathrm{C} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

The brute force method to verify whether AB = C is to recursively check whether all of the following equation are true

$$\begin{aligned} \mathrm{C}_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{11} + A_{22}B_{21} \end{aligned}$$

As we can see, we need 8 multiplication to verify whether AB = C and $\mathrm{O}(\mathrm{n}^2)$ time to add 2 matrixes. We can denote the total time of this brute force method as follow

$$\mathrm{T(n)} = 8T\left(\frac{n}{2}\right) + O(n^2)$$

With the master theorem, we can get the complexity of this method

$$\mathrm{T(n)} = \mathrm{O}(\mathrm{n}^{\log_8 2}) = O(n^3)$$

To reduce the number of multiplications, Strassen Algorithm uses intermediate variable to minimize the multiplication.

$$M_0 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_1 = (A_{21} + A_{22})B_{11}$$
$$M_2 = A_{11}(B_{12} - B_{22})$$
$$M_3 = A_{22}(B_{21} - B_{11})$$
$$M_4 = (A_{11} + A_{12})B_{22}$$
$$M_5 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$C_{11} = M_0 + M_3 - M_4 + M_6$$
$$C_{12} = M_2 + M_4$$
$$C_{21} = M_1 + M_3$$
$$C_{22} = M_0 - M_1 + M_2 + M_5$$

We can denote the total time of the Strassen Algorithm as follow

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

With the master theorem, we can get the complexity of this method

$$T(n) = O\left(n^{\log_7 2}\right) = O(n^{2.807})$$

As we can see, Strassen algorithm have a smaller complexity than the brute force method. But because asymptotic analysis does not include constant factor such as overhead in recursive function, there would be some difference in real-life scenario.

C. *Freivald's Algorithm*

Our third approach to verify the matrix multiplication algorithm is by using Freivald's Algorithm. Freivald's Algorithm is a probabilistic randomized algorithm used to verify matrix multiplication. The following procedure is the procedure to verify the given multiplication matrix with Freivald's Algorithm

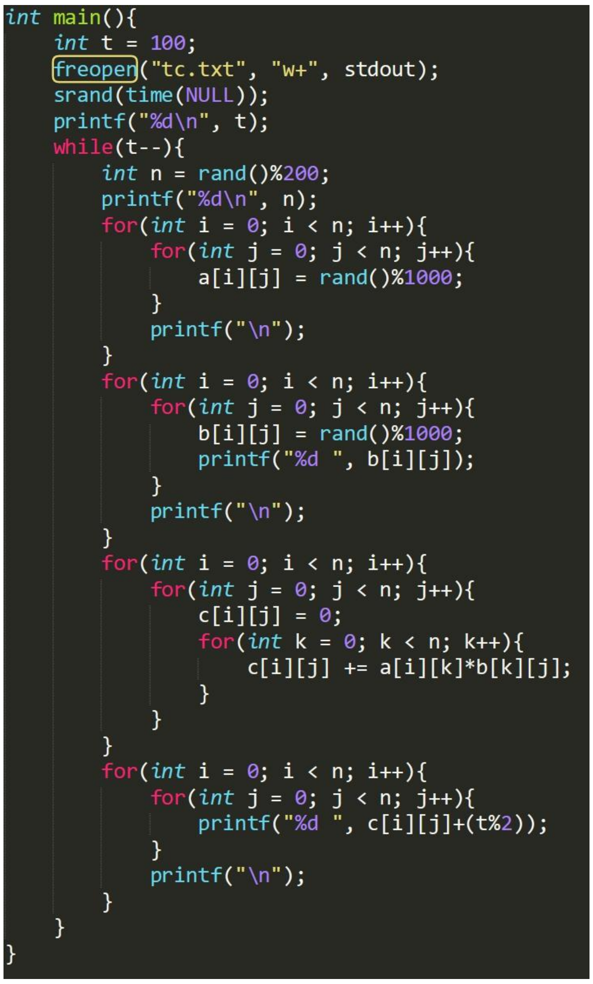
1. Generate an n-by-1 random 0/1 vector $\vec{r}$
2. Compute $\vec{P} = A\ x\ (B\vec{r}) - C\vec{r}$
3. Return true if $\vec{P} = (0,0,\dots,0)^T$, return false otherwise

This algorithm has $O(n^2)$ complexity. If $AB = C$, then the algorithm always returns true. But if $AB \neq C$, then the algorithm has $\frac{1}{2}$ chance to return false. To decrease the probability of error, we can iterate this algorithm multiple time. If we run the algorithm for k times, then this algorithm has $O(kn^2)$ complexity with the probability of error $\frac{1}{2^k}$.

IV. Verify Matrix Multliplication Algorithm

A. *Verifier Method*

To compare our matrix multiplication correctness verifier algorithm, we will create a testcase where there is a slight difference between matrix C and matrix AB. Figure 3 is our template test case generator. For each text file, we will create 100 randomized case where each case has matrix A, B, C with size no more than 20, 50, 100, 200, 500, and 1000. For our testcases, we will alternate the correct case and the wrong case. For the wrong cases, they will have a slight difference with the correct multiplication.

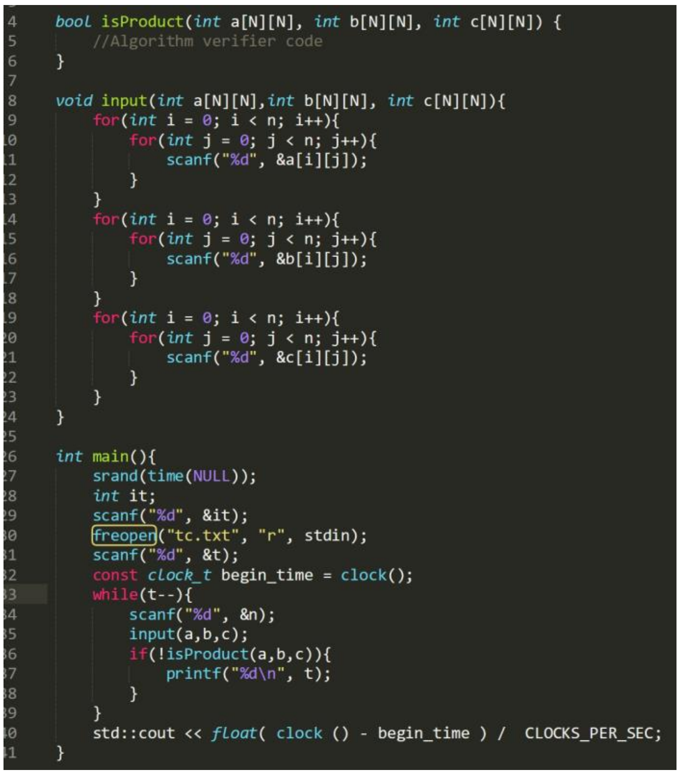


```
int main(){
    int t = 100;
    freopen("tc.txt", "w+", stdout);
    srand(time(NULL));
    printf("%d\n", t);
    while(t--){
        int n = rand()%200;
        printf("%d\n", n);
        for(int i = 0; i < n; i++){
            for(int j = 0; j < n; j++){
                a[i][j] = rand()%1000;
            }
            printf("\n");
        }
        for(int i = 0; i < n; i++){
            for(int j = 0; j < n; j++){
                b[i][j] = rand()%1000;
                printf("%d ", b[i][j]);
            }
            printf("\n");
        }
        for(int i = 0; i < n; i++){
            for(int j = 0; j < n; j++){
                c[i][j] = 0;
                for(int k = 0; k < n; k++){
                    c[i][j] += a[i][k]*b[k][j];
                }
            }
        }
        for(int i = 0; i < n; i++){
            for(int j = 0; j < n; j++){
                printf("%d ", c[i][j]+(t%2));
            }
            printf("\n");
        }
    }
}
```

Figure 3 Test case generator

To verify the correctness algorithm, we will use the same template for our algorithm checker and modify it accordingly, based on implementation detail of the given approach, such as the data structure that are needed and whether we need another parameter. For example, Freivald's Algorithm needed the number of iterations in its parameter to increases its correctness.

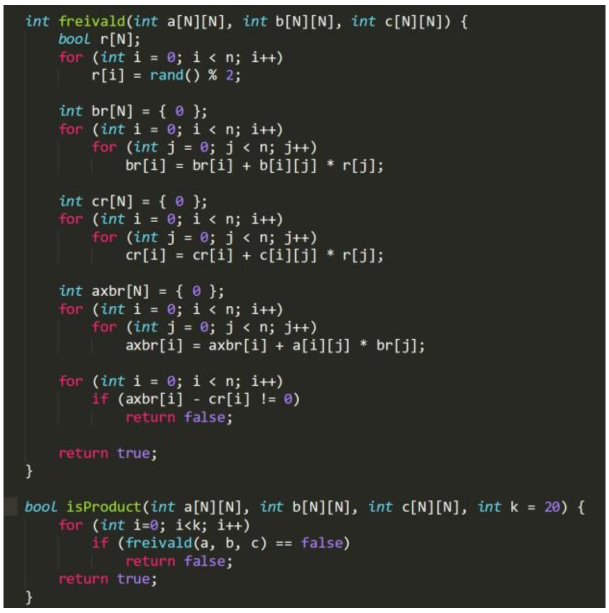
Figure 4 shows our algorithm verifier template. Our verifier algorithm will be called inside isProduct function.



```
bool isProduct(int a[N][N], int b[N][N], int c[N][N]) {
    //Algorithm verifier code
}

void input(int a[N][N],int b[N][N], int c[N][N]){
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            scanf("%d", &a[i][j]);
        }
    }
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            scanf("%d", &b[i][j]);
        }
    }
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            scanf("%d", &c[i][j]);
        }
    }
}

int main(){
    srand(time(NULL));
    int it;
    scanf("%d", &it);
    freopen("tc.txt", "r", stdin);
    scanf("%d", &t);
    const clock_t begin_time = clock();
    while(t--){
        scanf("%d", &n);
        input(a,b,c);
        if(!isProduct(a,b,c)){
            printf("%d\n", t);
        }
    }
    std::cout << float( clock () - begin_time ) /  CLOCKS_PER_SEC;
}
```

Figure 4 Algorithm verifier template

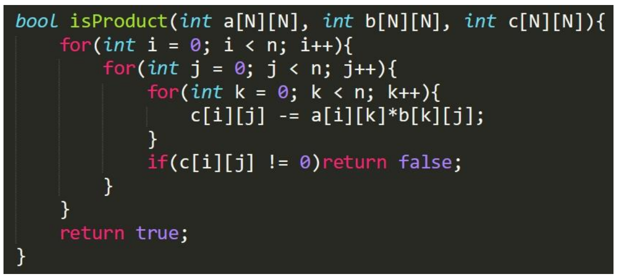
Our first verifier will use Freivald's Algorithm. We will use k = 20 to verify the correctness of a matrix multiplication with high accuracy, having an error less than $\frac{1}{2^k} \approx 9.5 * 10^{-7}$. This will be enough for our testcase where there is only 100 matrix in our testcases. The implementation of this algorithm can be seen in Figure 5.



```
int freivald(int a[N][N], int b[N][N], int c[N][N]) {
    bool r[N];
    for (int i = 0; i < n; i++)
        r[i] = rand() % 2;

    int br[N] = { 0 };
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            br[i] = br[i] + b[i][j] * r[j];

    int cr[N] = { 0 };
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cr[i] = cr[i] + c[i][j] * r[j];

    int axbr[N] = { 0 };
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            axbr[i] = axbr[i] + a[i][j] * br[j];

    for (int i = 0; i < n; i++)
        if (axbr[i] - cr[i] != 0)
            return false;

    return true;
}

bool isProduct(int a[N][N], int b[N][N], int c[N][N], int k = 20) {
    for (int i=0; i<k; i++)
        if (freivald(a, b, c) == false)
            return false;
    return true;
}
```

Figure 5 Freivald's algorithm (Taken from https://www.geeksforgeeks.org/freivalds-algorithm/ with modification)

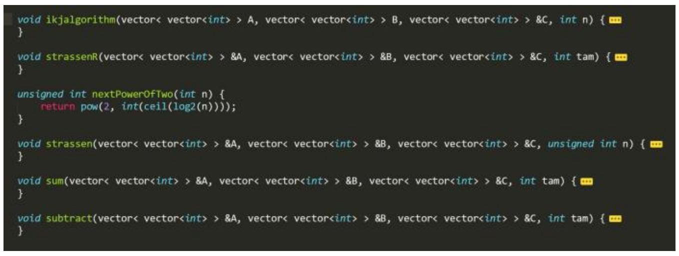
Our second verifier will use brute force method as its routine. This algorithm will directly use the definition of matrix multiplication to calculate the correctness of AB = C. In the implementation of this algorithm, we will not use any optimization such as optimizing cache memory. The implementation of this algorithm can be seen in Figure 6.



```
bool isProduct(int a[N][N], int b[N][N], int c[N][N]){
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            for(int k = 0; k < n; k++){
                c[i][j] -= a[i][k]*b[k][j];
            }
            if(c[i][j] != 0)return false;
        }
    }
    return true;
}
```

Figure 6 Brute Force Verifier

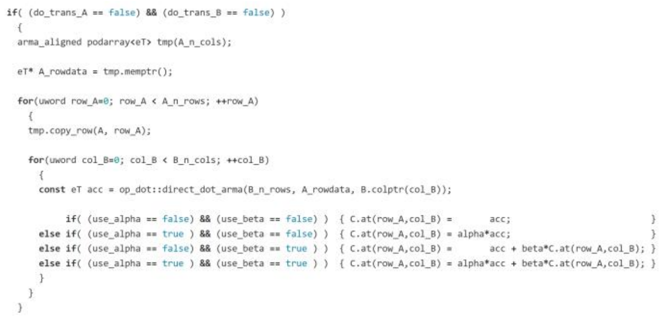
Our third verifier will use Strassen Algorithm as its routine. This algorithm will calculate matrix AB by divide and conquer method with some multiplication reduction to reduce the complexity of this algorithm. The implementation of this algorithm can be seen in Figure 7.



```
void ikjalgorithm(vector< vector<int> > A, vector< vector<int> > B, vector< vector<int> > &C, int n) { ⋯
}

void strassenR(vector< vector<int> > &A, vector< vector<int> > &B, vector< vector<int> > &C, int tam) { ⋯
}

unsigned int nextPowerOfTwo(int n) {
    return pow(2, int(ceil(log2(n))));
}

void strassen(vector< vector<int> > &A, vector< vector<int> > &B, vector< vector<int> > &C, unsigned int n) { ⋯
}

void sum(vector< vector<int> > &A, vector< vector<int> > &B, vector< vector<int> > &C, int tam) { ⋯
}

void subtract(vector< vector<int> > &A, vector< vector<int> > &B, vector< vector<int> > &C, int tam) { ⋯
}
```

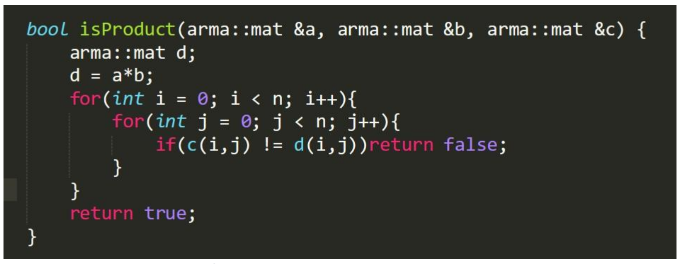
Figure 7 Header of Strassen Algorithm (taken from https://martin-thoma.com/strassen-algorithm-in-python-java-cpp/ with modification)

For our fourth verifier, we will use C++ Linear Algebra libraries called Armadillo. Armadillo provides matrix multiplication method in the library with their own implementation method. The implementation of Armadillo matrix multiplication method can be seen in Figure 8, and our verifier implementation using Armadillo can be seen in Figure 9.



```
if( (do_trans_A == false) && (do_trans_B == false) )
 {
 arma_aligned podarray<eT> tmp(A_n_cols);

 eT* A_rowdata = tmp.memptr();

 for(uword row_A=0; row_A < A_n_rows; ++row_A)
   {
   tmp.copy_row(A, row_A);

   for(uword col_B=0; col_B < B_n_cols; ++col_B)
     {
     const eT acc = op_dot::direct_dot_arma(B_n_rows, A_rowdata, B.colptr(col_B));

          if( (use_alpha == false) && (use_beta == false) )  { C.at(row_A,col_B) =       acc;                              }
     else if( (use_alpha == true ) && (use_beta == false) )  { C.at(row_A,col_B) = alpha*acc;                              }
     else if( (use_alpha == false) && (use_beta == true ) )  { C.at(row_A,col_B) =       acc + beta*C.at(row_A,col_B); }
     else if( (use_alpha == true ) && (use_beta == true ) )  { C.at(row_A,col_B) = alpha*acc + beta*C.at(row_A,col_B); }
     }
   }
 }
```

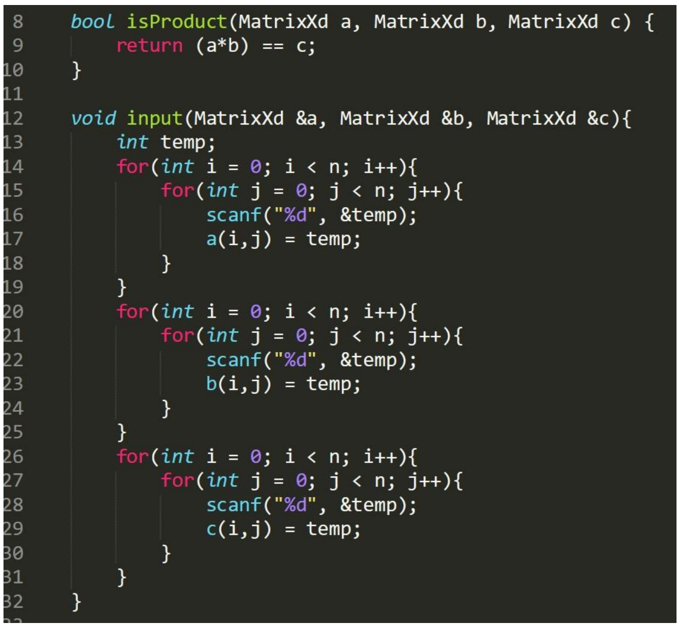
Figure 8 Main Routine of C++ Armadillo Matrix Multiplication (taken from https://martin-thoma.com/strassen-algorithm-in-python-java-cpp/)



```cpp
bool isProduct(arma::mat &a, arma::mat &b, arma::mat &c) {
    arma::mat d;
    d = a*b;
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            if(c(i,j) != d(i,j))return false;
        }
    }
    return true;
}
```

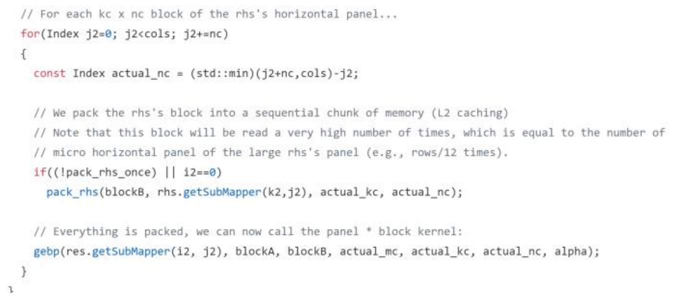
Figure 9 Verify Using Armadillo Matrix Multiplication

For our fifth verifier, we will use another Linear Algebra library called Eigen. Eigen is a lightweight Linear Algebra library, contrary to Armadillo. Syntax for simple matrix multiplication in Eigen is similar to Armadillo. Figure 10 shows our implementation for matrix multiplication using Eigen, and Figure 11 shows part of implementation of Eigen Matrix multiplication



```cpp
bool isProduct(MatrixXd a, MatrixXd b, MatrixXd c) {
    return (a*b) == c;
}

void input(MatrixXd &a, MatrixXd &b, MatrixXd &c){
    int temp;
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            scanf("%d", &temp);
            a(i,j) = temp;
        }
    }
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            scanf("%d", &temp);
            b(i,j) = temp;
        }
    }
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            scanf("%d", &temp);
            c(i,j) = temp;
        }
    }
}
```

Figure 10 Verify using Eigen Matrix Multiplication



```cpp
// For each kc x nc block of the rhs's horizontal panel...
for(Index j2=0; j2<cols; j2+=nc)
{
  const Index actual_nc = (std::min)(j2+nc,cols)-j2;

  // We pack the rhs's block into a sequential chunk of memory (L2 caching)
  // Note that this block will be read a very high number of times, which is equal to the number of
  // micro horizontal panel of the large rhs's panel (e.g., rows/12 times).
  if((!pack_rhs_once) || i2==0)
    pack_rhs(blockB, rhs.getSubMapper(k2,j2), actual_kc, actual_nc);

  // Everything is packed, we can now call the panel * block kernel:
  gebp(res.getSubMapper(i2, j2), blockA, blockB, actual_mc, actual_kc, actual_nc, alpha);
}
```

Figure 11 Part of Implementation of Eigen Matrix Multiplication (taken from https://github.com/libigl/eigen/blob/master/Eigen/src/Core/products/GeneralMatrixMatrix.h)

B. *Result*

Using the previous verifier algorithm, Figure 12 shows the time needed to verify the correctness of matrix multiplication. We can see from the Figure that Freivald's Algorithm outperform another algorithm in the bigger testcases. On smaller testcases, all algorithm has similar performance, but because Armadillo have some implementation overhead and Strassen have recurrence overhead, they're slightly slower.

Another interesting point from here is that although brute force method has bigger complexity than Strassen Algorithm, Brute Force method perform better on all testcase than Strassen Algorithm. This is because Strassen Algorithm that we choose overuse function and function recurrence in their implementation. This causes more function overhead, and results in slower time.

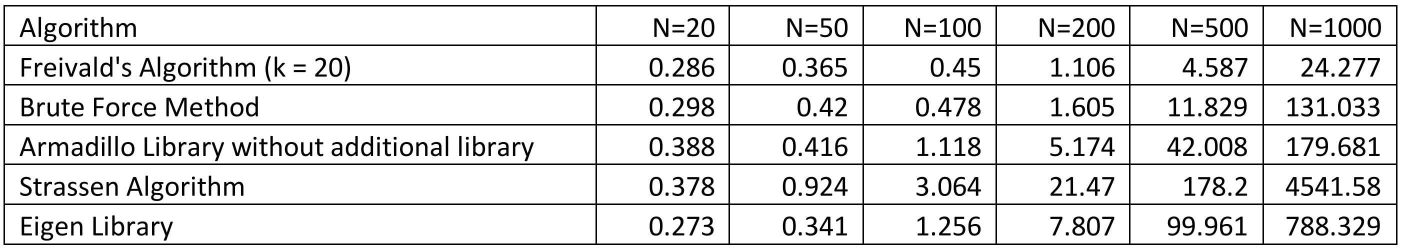
For Armadillo result, they have a bit slower time than straightforward brute force method. This is because they have implementation overhead and are not optimized for matrix multiplication directly.

For Eigen result, they excel in small testcase size because of its lightweight nature, resulting in smaller overhead. But because they're not optimized for bigger cases, they have slower time than armadillo, but still faster than Strassen.

V. Conclusion

For verifying the correctness of Matrix Multiplication result, Freivald's Algorithm outperform another algorithm that directly multiply the given matrix. But because of its probabilistic nature, we need to determine sufficient iteration that are needed to confidently claim the given matrix multiplication correctness.

Another interesting point that we get is to get a better algorithm to multiply 2 matrixes better than the brute force method, we need to carefully implement the proposed algorithm to not have useless overhead. Because brute force method almost doesn't have useless overhead and they can get better performance with the use of cache memory, they are better than badly implemented algorithm with smaller complexity but higher constant factor.



Algorithm	N=20	N=50	N=100	N=200	N=500	N=1000
Freivald's Algorithm (k = 20)	0.286	0.365	0.45	1.106	4.587	24.277
Brute Force Method	0.298	0.42	0.478	1.605	11.829	131.033
Armadillo Library without additional library	0.388	0.416	1.118	5.174	42.008	179.681
Strassen Algorithm	0.378	0.924	3.064	21.47	178.2	4541.58
Eigen Library	0.273	0.341	1.256	7.807	99.961	788.329

Figure 12 Time Needed for Verifying Matrix Multiplication

VI. ACKNOWLEDGMENT

I would like to express my deep gratitude to Mrs Nur Ulfa Maulidevi, Mrs. Masayu Leylia Khodra, and Mr. Rinaldi Munir as our lecturer in Algorithm Strategy Course for the knowledge that they shared upon us. I also would like to thank my family and friends for help and support while this paper is created. I also would like to thank all programming language creator that open-source their language implementation, as their contribution to make their source-code open helps me test my algorithm implementation.

REFERENCES

[1] Huang, J., Smith, T., Henry, G. and Geijn, R. (2019). *Strassen's Algorithm Reloaded*. [online] Jianyuhuang.com. Available at: http://jianyuhuang.com/papers/sc16.pdf [Accessed 25 Apr. 2019].

[2] Bentley, Jon Louis*;* Haken, Dorothea*;* Saxe, James B. *(September 1980), "A general method for solving divide-and-conquer recurrences",* ACM SIGACT News, ***12** (3): 36–44,* doi*:*10.1145/1008861.1008865K.

[3] Munir, R. (2019). *Divide and Conquer*. [online] Informatika.stei.itb.ac.id. Available at: http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Algoritma-Divide-and-Conquer-(2018).pdf [Accessed 25 Apr. 2019].

VII. 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, 25 April 2019

TTD

[illegible]

Adyaksa Wisanggeni 13517091