Implementation Analysis of Strassen-Like Matrix Multiplication Algorithms Based on Block Decomposition

by

Chongchong Liu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Matrix multiplication is one of the most widely used operations in all computational fields of linear algebra. The complexity of the naive method for multiplying two $n \times n$ matrices requires $O(n^3)$ arithmetic operations over the ring in which the matrix entries lie. In 1969, Strassen proposed the first sub-cubic complexity algorithm for matrix multiplication. Strassen's algorithm (SA) multiplies two 2×2 matrices using 7 multiplications and 18 additions over the ring. Later Winograd proposed a variant of SA that requires 7 multiplications, but 15 additions over the ring. Algorithms that multiply two 2×2 matrices using 7 multiplications over the ring are called Strassen-like algorithms and have a complexity of $O(n^{2.81})$. Although asymptotically better algorithms exist, Strassen-like algorithms are considered to be most widely used sub-cubic complexity algorithm.

Recently, Cenk and Hasan proposed techniques to reduce the arithmetic cost of Strassen-like algorithms. The main technique is to decompose Strassen-like algorithms into three blocks, namely, component matrix formation (CMF), component multiplication (CM), and reconstruction (R). Each block is a recursive operation. In this thesis, we study these building blocks and investigate three optimization methods: the linearity property of CMF and R, limited recursion, and block recombination.

In this thesis, software implementation and hardware simulation are also performed to support the theoretical analysis. For software implementation, experiment results show that WV is approximately 15% faster than SA. Cenk and Hasan's techniques yield an improved WV (IWV) that considerably reduces the matrix multiplication time in software. For hardware simulation, we conclude from the synthesis results that WV consumes about 7.5% less logic elements than SA. IWV for different matrix sizes are also tested to successfully reduce resource utilization and timing cost.

Acknowledgements

First and foremost, I would like to thank Professor Anwar Hasan for giving me endless support and precious guidance that helped me to succeed. The research of this thesis was completed under his supervision. I am so lucky to have a supervisor who cared so much about my work, and who responded to my queries so promptly. It was his encouragement and patience to help me overcome all the difficulties that I met during studies. I am also indebted to Natural Sciences and Engineering Research Council of Canada for their generous financial support and to Electrical and Computer Engineering Department, University of Waterloo for the teaching assistantships during my MASc program.

I sincerely thank all the professors who taught me, expecially Professors Guang Gong, Nachiket Kapre, Andrew Morton, and Mahesh Tripunitara. I am grateful to Mason, Amiee and Ethan for their support and encouragement. I would like to express my heartfelt appreciation to all current and previous group members: Mohannad, Tanushree, Arshee, Crystal, Xiaolin for their insights and suggestions. I would also like to thank my friend Di Sang, Sigeng Chen, Jian, Rui Hong, Yuxuan Liu and Mier Ta for encouraging and motivating me a lot throughout this period.

Finally, I must express my profound gratitude to my family and to my best friend Jiling Luo for providing me with infinite love and encouragement in research and writing this thesis. This accomplishment would not have been possible without their support. Thank you all.

Dedication

This thesis is dedicated to my parents, Suoping Liu and Huanjun Liu, and my brother, Shuo Shuo Liu for their endless love, support and encouragement that motivated me to achieve this success. I love you all.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

BR Block Recombination

CM Component Multiplication

CMF Component Matrix Formation

IDE Integrated Development Environment

IWV Improved Winograd's Variant

LE Logic Elements

LUT Look-Up Table

OBO Original Block Organization

R Reconstruction

SA Strassen's Algorithm

WV Winograd's Variant

Chapter 1

Introduction

Linear algebra is an area of study on vectors and linear functions, broadly applied to computer applications, ranging from games to business [8]. It plays a principal role in all fields of mathematics. In this thesis, we will focus on the arithmetic operations of one kind of linear functions, namely matrix. In general terms, it is considered as the arrangement of information related to linear functions. The applications of matrix multiplications in engineering include digital image processing, electrical circuits, software engineering, graph problem solving, data mining, and security [25]. Over the past few decades, there has been a lot of research towards efficient matrix multiplication [27].

1.1 Motivation

In mathematics, matrix multiplication is an operation that generates a matrix by implementing linear computations on two matrices with entries in a certain *ring* [4]. Matrix multiplication plays a fundamental role in solving algorithmic linear algebra problems [10]. The computation time and resource cost of matrix multiplications have a great influence on the performance of variety of applications.

The naive method for multiplying two $n \times n$ matrices, as stated in the following section, costs n^3 multiplications and $n^3 - n^2$ additions over the ring in which the matrix entries lie. Thus it results in a complexity of $O(n^3)$ [20].

In 1969, Strassen [34] described an algorithm to improve matrix multiplication. It costs 7 multiplications and 18 additions over the ring while multiplying two 2×2 matrices. When we extend Strassen's algorithm to $n \times n$ matrix multiplication and use recursion, it takes $7n^{2.81} - 6n^2$ arithmetic operations (multiplications and additions combined). In 1971, Winograd [38] proposed a variant of the Strassen's algorithm. For multiplying two 2×2 matrices, Wingrad's variant requires 7 multiplications but 15 additions, yielding a total of $6n^{2.81} - 5n^2$ arithmetic operations for multiplying two $n \times n$ matrices. Any method that requires 7 multiplications to multiply two 2×2 matrices is called Strassen-like algorithm and has an asymptotic complexity of $O(n^{2.81})$ [7].

The first work reporting asymptotically better than Strassen's method is V.Y. Pan's $O(n^{2.781})$ [26] algorithm that uses trilinear aggregating techniques. Since then, other methods that are asymptotically better have been proposed, e.g., Winograd and Coppersmith's $O(n^{2.376})$ [11], Sothers' $O(n^{2.374})$ [35] and Williams' $O(n^{2.373})$ [37] algorithms. These algorithms are rarely used in practice because of the large constant factors in real implementations [15] [32].

In this thesis, we consider Strassen-like algorithms as they are more efficient in practice for large size n used in cryptographic applications. In order to further improve the computational complexity, Cenk and Hasan [7] proposed an improved Winograd's variant using block recombination and limited recursion.

1.2 Previous Work

1.2.1 Software Implementation of Matrix Multiplication

In [12], the authors presented a new fast matrix multiplication algorithm which is a hybrid combination of Strassen's algorithm and its Winograd's variant and showed the performance of this novel algorithm by implementing it on single and multi cores processors. It was concluded that the hybrid algorithm performed better than the naive matrix multiplication algorithm when matrix size is larger than 3000×3000.

In [22], Kouya outlined the performance of Strassen's algorithm, and Winograd's variant through benchmark tests. Winograd's variant was more efficient than Strassen's algorithm in time complexity.

In [30], the authors succeeded to implement matrix multiplication of Strassen's algorithm on NVIDIA GPU using CUDA. The recursion limit of Strassen's algorithm implemented on CPU was smaller than that of implementing it on GPU.

1.2.2 Hardware Implementation of Matrix Multiplication

In [21], Khayyat designed a flexible implementation of parallel matrix multiplication for FPGA devices by exploiting the use of blocks and parallelization. The experiment

was implemented using VHDL to verify correctness of design and tested on the Altera DE4 board, featuring a Stratix IV EP4SGX530C2 FPGA device. The experiment result showed that design scaled with respect to consumed resources. Increasing system size reduced maximum operating frequency, but improved system performance.

In [13], the authors introduced a design of 64-bit floating-point matrix multiplier optimized for FPGA implementations. Taking I/O bandwidth and memory limitation into consideration, an optimum scheme was proposed for better data locality and reusability. They implemented a scalable linear array of processing elements supporting proposed design using the Xilinx Virtex II pro technology. Better performance-area ratio was reported in comparison with previous work.

In [23], the authors talked about the implementation of Four Russians of Multiplication (M4RM), which is one of most efficient algorithms for dense matrix multiplication over the binary field. They reported an efficient tile-based hardware/software implementation of M4RM. The design of 64×64 and 128×128 block matrix multiplication was targeted to fit for FPGAs using System Verilog.

1.3 Background

In mathematics, a matrix is a set of symbols, numbers, or expressions arranged in horizontal and vertical lines within a rectangular array [19]. It originates from square arrays formed by coefficients and constants of an equation set [14].

The size of a matrix is defined by the number of rows and columns in the matrix. If a matrix has m rows and n columns, then it is called an $m \times n$ matrix [36]. It is denoted

as:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} = (a_{ij}) \in \mathcal{R}^{m \times n},$$

where the entry a_{ij} is in the i-th row and j-th column.

1.3.1 Naive Matrix Multiplication

Matrix multiplication refers to the product of two matrices with entries in a certain *ring* [28]. More specifically, assume that A is an $m \times p$ matrix and B is an $p \times n$ matrix:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1p} \\ a_{21} & a_{22} & \cdots & a_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mp} \end{pmatrix} \text{ and } B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{p1} & b_{p2} & \cdots & b_{pn} \end{pmatrix},$$

the matrix multiplication will produce a result matrix C with size of $m \times n$, denoted as:

$$C = A \times B = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{pmatrix},$$

such that

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{ip}b_{pj} = \sum_{k=1}^{p} a_{ik}b_{kj},$$

where $i = 1, \cdots, m$ and $j = 1, \cdots, n$ [33]. That is to say: c_{ij} is the multiply-and-add of the i-th row of A and j-th column of B as in Figure 1.1. The definition of matrix multiplication also determines the properties that it follows [16]:

- Non-Commutative Law. Given two matrices A and B of size $m \times n$ and $n \times m$ respectively, the size of matrix product AB is $m \times m$, and the size of matrix product BA is $n \times n$. In the case that $m \neq n$, clearly AB is not the same as BA. Even, when $m = n$, the two products AB and BA are not generally the same.

- Distributive Law. With respect to matrix addition/subtraction, it follows that matrix multiplication is distributive. i.e.,

$$A(B + C) = AB + AC \text{ and } (B - C)A = BA - CA.$$

- Associative Law. Assume that $A = (a_{ij})_{m \times n}$, $B = (b_{ij})_{p \times q}$, and $C = (c_{ij})_{r \times s}$, in which case that $n = p$ and $q = r$, $(AB)C = A(BC)$ will hold.

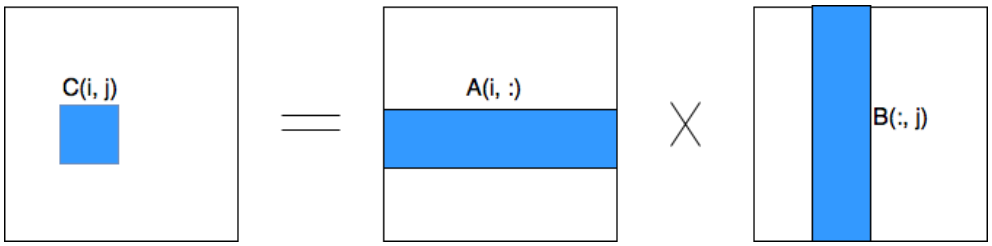

Figure 1.1: Naive Matrix Multiplication Method

1.3.2 Block Matrix Multiplication

To improve the performance of matrix multiplication, we can partition a matrix into several sub-matrices or blocks of smaller sizes. A block matrix multiplication refers to multiplying two matrices block by block.

For example, in the simplest case, A is an $M \times N$ matrix, and B is an $N \times M$ matrix. Consider A as a column matrix of m blocks where each block is a row vector, and

consider B as a row matrix of m blocks where each block is a column vector. Note that for the purpose of successfully implementing block matrix multiplication, the number of columns in A should be equal to the number of rows in B [5]. The commonly applied block matrix multiplication methodology is to fix block size as shown in Figure 1.2.

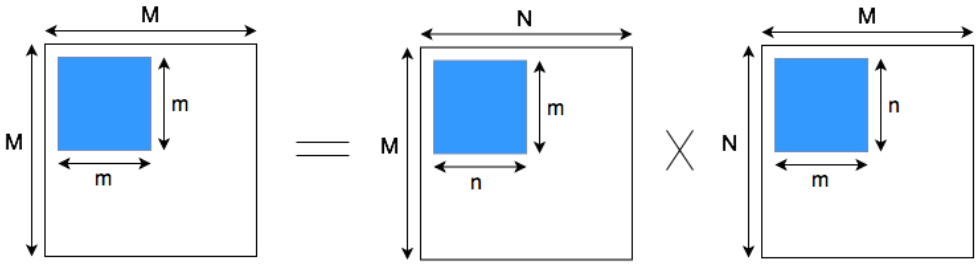

Figure 1.2: Naive Block Matrix Multiplication Method

Block matrix multiplication is facilitated for improving computing system performance. However, it greatly increases the communication complexity to be $O(n^3)$ because it takes plentiful time to process matrix into blocks. Therefore, it is necessary to optimize the computation and communication costs in terms of design and implementation [29].

1.3.3 Divide and Conquer

Recursion is a basic structure to construct data flow with repeatedly executing the body of a procedure. Recursion unrolling is a complicated methodology to optimize recursive procedures [31]. Divide and Conquer algorithm is a kind of recursion unrolling methods. It works by recursively dividing the main problem into two or more sub-problems, until these sub-problems could be small enough to be easily solved. It could help solve complex problems with a reduced degree of difficulty, but it also delays program execution. As shown in Figure 1.3, a typical divide-and-conquer algorithm is divided into 3 steps [9]:

- Divide. Split the main problem into several sub-problems.
- Conquer. Resolve these sub-problems recursively.
- Combine. Combine these solutions to produce the final result.

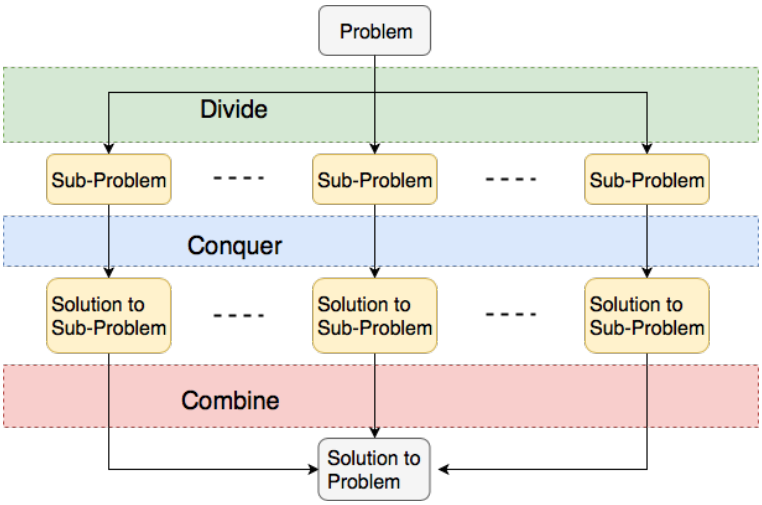

Figure 1.3: Divide and Conquer Algorithm

1.4 Scope of Work

In this thesis, we will discuss two Strassen-like algorithms, namely Strassen's algorithm and its Winograd's variant. The idea of Strassen-like algorithms is based on the divide-and-conquer rule in the sense that it also splits matrices into sub-matrices for sub-problems. We will make a detailed analysis and summarization of Cenk and Hasan's proposal of decomposing algorithm into three blocks. Several examples are introduced for instantiation and verification of each block operation. More techniques are investigated

to improve algorithm complexity including: the linearity property of two building blocks called CMF and R, limited recursion, and block recombination [7].

All realizations are carried out in software using C++ and simulated in hardware using Verilog. For software implementation, we will consider matrix multiplication with matrix dimension 2^7, 2^8 and 2^9. Experiments will show the timing result for each method. For hardware simulation, we implement it using Quartus II with Cyclone IV E family. Matrix size is chosen to be size 2^i, for $i = 1, \cdots, 5$. Logic elements, memory usage, maximum frequency, and clock cycles are considered as experiment metrics. We will combine the experimental results of both software implementation and hardware simulation to prove that improved Strassen-like algotihms are likely to provide better performance.

1.5 Thesis Organization

The organization of this thesis is as follows:

Chapter 2 provides the details of Strassen-like matrix multiplication algorithms. It presents the use of block decomposition to divide Strasesn's algorithm and Winograd's variant into three blocks. The computation complexity is also listed.

In Chapter 3, we review relevant known ideas to improve Strassen-like algorithms. The ideas for improving computation complexity include observing the linearity property of CMF and R, limited recursion and block recombination. They will be explained in details.

In Chapter 4 and 5, we present performance analysis of software implementation and hardware simulation. For software implementation, we demonstrate the pseudocode of each block and regard timing cost as the parameter to measure algorithms' performance.

For hardware simulation, the logic elements, memory, clock cycles, and maximum frequency are considered.

Chapter 6 includes a summary of this thesis work and future research scopes on improving and implementing Strassen-like matrix multiplication algorithms.

Chapter 2

Overview of Strassen-like Algorithms for Matrix Multiplication

Matrix multiplication is an operation widely used in scientific computing. A lot of research has been devoted to improve the efficiency of matrix multiplication. The work described in this thesis focuses on the widely used Strassen-like algorithms. To this end, this chapter describes the block decomposition of Strassen's algorithm and its Winograd's variant. Each block's arithmetic complexity will be provided in details. Unless stated otherwise, all the matrices considered in the rest of this thesis are defined with size of $n = 2^k$ where k is a positive integer.

2.1 Two Strassen-like Algorithms

2.1.1 Strassen's Algorithm

In 1969, Strassen made a great improvement on matrix multiplication by reporting an algorithm of complexity $O(n^{2.81})$. In the case of multiplying two 2×2 matrices, it only needs 7 multiplications and 18 additions instead of 8 multiplications and 4 additions in the naive method [18]. The flow of computation in the Strassen's algorithm (SA) is as follows.

Assume two 2×2 matrices for multiplication and write:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}, \quad C = A \times B.$$

- The first step is to perform additions/subtractions:

  $U_1 = a_{11} + a_{22}, \quad U_2 = a_{21} + a_{22}, \quad U_3 = a_{11} + a_{12}, \quad U_4 = a_{21} - a_{11}, \quad U_5 = a_{12} - a_{22},$

  $V_1 = b_{11} + b_{22}, \quad V_2 = b_{12} - b_{22}, \quad V_3 = b_{21} - b_{11}, \quad V_4 = b_{11} + b_{12}, \quad V_5 = b_{21} + b_{22}.$

- The second step is to produce the products $P_1, P_2, \cdots, P_7$:

  $P_1 = U_1 V_1, \quad P_2 = U_2 b_{11}, \quad P_3 = a_{11} V_2, \quad P_4 = a_{22} V_3,$

  $P_5 = U_3 b_{22}, \quad P_6 = U_4 V_4, \quad P_7 = U_5 V_5.$

- The final step is to compute:

  $c_{11} = P_1 + P_4 - P_5 + P_7, \quad c_{12} = P_2 + P_4,$

  $c_{21} = P_3 + P_5, \quad c_{22} = P_1 + P_3 - P_2 + P_6.$

- The four expressions immediately above lead to the output matrix:

$$C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}.$$

2.1.2 Winograd's Variant

For further improvement on matrix multiplication, Winograd [38] proposed a modification that requires 3 less additions than SA. This algorithm is constructed in the same manner as in SA. The flow of WV is given below.

Let us consider two 2×2 matrices and write:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}, \quad C = A \times B.$$

- The first step is to perform additions/subtractions:

  $V_1 = b_{22} - b_{12}$, $V_2 = V_1 + b_{11} = b_{11} + (b_{22} - b_{12})$, $V_3 = V_2 - b_{21} = (b_{11} - b_{12} + b_{22}) - b_{21}$,

  $U_1 = a_{11} - a_{21}$, $U_2 = a_{21} + a_{22}$, $V_4 = b_{12} - b_{11}$,

  $U_3 = U_1 - a_{22} = (a_{11} - a_{21}) - a_{22}$, $U_4 = U_3 + a_{12} = (a_{11} - a_{21} - a_{22}) + a_{12}$.

- The second step is to produce the products $P_1, P_2, \cdots, P_7$:

  $P_1 = a_{11}b_{11}$, $P_2 = a_{12}b_{21}$, $P_3 = a_{22}V_3$, $P_4 = U_1V_1$,

  $P_5 = U_2V_4$, $P_6 = U_4b_{22}$, $P_7 = U_3V_2$.

- The final step is to compute:

  $c_{11} = P_1 + P_2$, $c_{12} = ((P_1 - P_7) + P_5) + P_6$,

  $c_{21} = (P_1 - P_7) - P_3 + P_4$, $c_{22} = (P_1 - P_7 + P_5) + P_4$.

- The four expressions immediately above lead to the output matrix:

$$C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}.$$

It is important to be aware of the reuse of some expressions in WV. $(a_{11} - a_{21})$ is used both in U_1 and U_3, and $(b_{22} - b_{12})$ is used both in V_1 and V_2. $(a_{11} - a_{21} - a_{22})$ is used both in U_3 and U_4. $(b_{11} - b_{12} + b_{22})$ is used both in V_2 and V_3. $(P_1 - P_7)$ in c_{12} is also used in c_{21}. $(P_1 - P_7 + P_5)$ in c_{12} is also used in c_{22}. Therefore, following the rule of no repeating operations with same parameters, it's easy to find out that it only needs 8 and 7 additions/subtractions to obtain P_i's and c_{ij}'s respectively. The total arithmetic cost for implementing WV is 7 multiplications and 15 additions [6].

2.2 Block Decomposition of Strassen's Algorithm

In [17], the authors decompose the recursive algorithm into a couple of independent blocks. Based on this idea, Cenk and Hasan [7] provide a detailed decomposition of SA and WV into three main blocks as shown in Figure 2.1: component matrix formation (CMF), component multiplication (CM), and reconstruction (R).

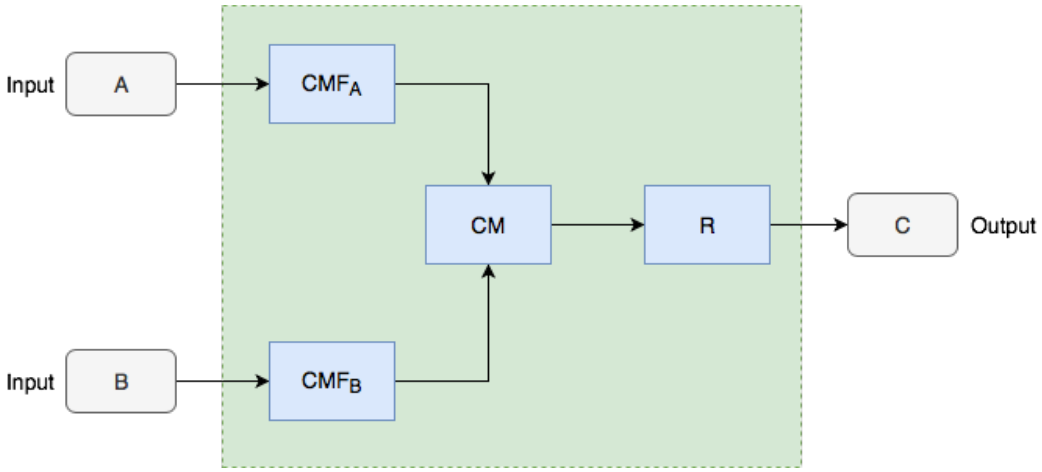

Figure 2.1: Architecture of Block Decompostion

The first step, CMF, is to deal with the input matrix A and B, and compute all needed linear combinations of entries in A and B. The size of output of CMF is determined by input matrix dimension n. Since CMF splits the input into 7 blocks in each recursion, the size will be $7^k = n^{\log_2 7}$ while it needs k recursions to unroll. The next step is to component-wise multiply those linear combinations of A and B. It is called CM, which yields the products $P_1, \cdots, P_7$. The size of CM's output is 7^k as well. The final step, called R, is to reconstruct these products with linear combinations in order to generate the final results c_{11}, c_{12}, c_{21}, and c_{22}.

2.2.1 SA's Component Matrix Formation

For two $n \times n$ matrices A and B, recursive CMF_A^{SA} and CMF_B^{SA} are defined as follows:

$$\begin{cases} U_1 = a_{11} + a_{22},\ U_2 = a_{21} + a_{22},\ U_3 = a_{11} + a_{12},\ U_4 = a_{21} - a_{11},\ U_5 = a_{12} - a_{22} \\ CMF_A^{SA}(A) = a_{11} \text{ for } n = 1 \\ CMF_A^{SA}(A) = \begin{pmatrix} CMF_A^{SA}(U_1), CMF_A^{SA}(U_2), CMF_A^{SA}(a_{11}), CMF_A^{SA}(a_{22}), \\ CMF_A^{SA}(U_3), CMF_A^{SA}(U_4), CMF_A^{SA}(U_5) \end{pmatrix} \text{ for } n \geq 2 \end{cases} \tag{2.1}$$

$$\begin{cases} V_1 = b_{11} + b_{22},\ V_2 = b_{12} - b_{22},\ V_3 = b_{21} - b_{11},\ V_4 = b_{11} + b_{12},\ V_5 = b_{21} + b_{22} \\ CMF_B^{SA}(B) = b_{11} \text{ for } n = 1 \\ CMF_B^{SA}(B) = \begin{pmatrix} CMF_B^{SA}(V_1), CMF_B^{SA}(b_{11}), CMF_B^{SA}(V_2), CMF_B^{SA}(V_3), \\ CMF_B^{SA}(b_{22}), CMF_B^{SA}(V_4), CMF_B^{SA}(V_5) \end{pmatrix} \text{ for } n \geq 2 \end{cases} \tag{2.2}$$

In this case, CMF applied to $n \times n$ matrix is unrolled into seven CMFs applied on half size matrices and it needs five additions/subtractions applied on $\frac{n}{2} \times \frac{n}{2}$ sub-matrices to compute $U_1, \cdots, U_5$ or $V_1, \cdots, V_5$. Thus, we can conclude that Strassen's CMF complexity

is:

$$M_{CMF}^{SA}(n) \leq 7M_{CMF}^{SA}\left(\frac{n}{2}\right) + 5\left(\frac{n}{2}\right)^2, \quad M_{CMF}^{SA}(1) = 0 \Longrightarrow M_{CMF}^{SA}(n) = \frac{5}{3}n^{\log_2 7} - \frac{5}{3}n^2.$$

2.2.2 SA's Component Multiplication

As shown in Figure 2.2, Component Multiplication is an operation where the corresponding component matrices, such as CMF_{A1} and CMF_{B1} formed from A and B, are multiplied. Since an $n \times n$ matrix leads to seven half size component matrices, SA's CM complexity is:

$$M_{CM}^{SA}(n) \leq 7M_{CM}^{SA}\left(\frac{n}{2}\right), \quad M_{CM}^{SA}(1) = 1 \Longrightarrow M_{CM}^{SA}(n) = n^{\log_2 7}.$$

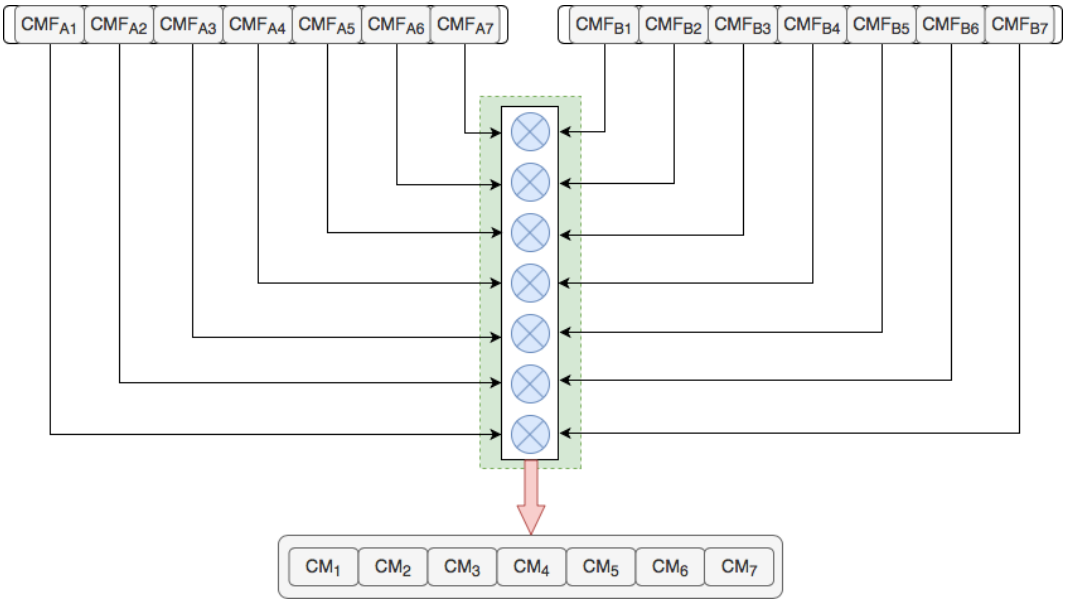

Figure 2.2: Architecture of Component Multiplicationn

2.2.3 SA's Reconstruction

Let $C = (C_1, C_2, \cdots, C_7)$ be the 7-tuple obtained after a CM operation. C is the input of the Reconstruction block. Its output is a matrix with size $n \times n$. The recursive Reconstruction algorithm is defined as follows:

$$\begin{cases} R^{SA}(C) = C_1 \text{ for } n = 1 \\ R^{SA}(C) = \begin{pmatrix} R^{SA}(C_1) \oplus R^{SA}(C_4) \ominus R^{SA}(C_5) \oplus R^{SA}(C_7), R^{SA}(C_3) \oplus R^{SA}(C_5), \\ R^{SA}(C_2) \oplus R^{SA}(C_4), R^{SA}(C_1) \ominus R^{SA}(C_2) \oplus R^{SA}(C_3) \oplus R^{SA}(C_6) \end{pmatrix} \text{ for } n \geq 2 \end{cases} \tag{2.3}$$

It is important to note that the length of C_i's for $i = 1, \cdots, 7$ is $n^{\log_2 7}/7 = 7^{k-1}$ and the size of $R(C)$ is $n^2 = 4^k$. Clearly, there are 7 R's applied on vectors of length 7^{k-1} and 8 additions/subtractions applied on matrices with size of $\frac{n}{2} \times \frac{n}{2}$. Thus, the complexity is:

$$M_R^{SA}(n) \leq 7M_R^{SA}\left(\frac{n}{2}\right) + 8\left(\frac{n}{2}\right)^2, \quad M_R^{SA}(1) = 0 \Longrightarrow M_R^{SA}(n) = \frac{8}{3}n^{\log_2 7} - \frac{8}{3}n^2.$$

2.3 Block Decomposition of Winograd's Variant

Using block decomposition proposed by Cenk and Hasan, Winograd's variant is also divided into three blocks: CMF, CM, and R. Below, we give the complexities of the CMF and R blocks. The CM block of Winograd's variant is the same as that discussed in the previous section for Strassen's algorithm and hence it is not repeated here.

2.3.1 WV's Component Matrix Formation

For Component Matrix Formation, WV only needs in total 8 additions rather than 10 additions in SA. That's a considerable improvement on arithmetic cost reduction. Consider two $n \times n$ matrices A and B as before. Then their CMFs for WV are defined as:

$$\begin{cases} U_1 = a_{11} - a_{21},\ U_2 = a_{21} + a_{22},\ U_3 = U_1 - a_{22},\ U_4 = U_3 + a_{12}, \\ CMF_A^{WV}(A) = a_{11} \text{ for } n = 1 \\ CMF_A^{WV}(A) = \begin{pmatrix} CMF_A^{WV}(a_{11}), CMF_A^{WV}(a_{12}), CMF_A^{WV}(a_{22}), CMF_A^{WV}(U_1), \\ CMF_A^{WV}(U_2), CMF_A^{WV}(U_4), CMF_A^{WV}(U_3) \end{pmatrix} \text{ for } n \geq 2 \end{cases} \tag{2.4}$$

$$\begin{cases} V_1 = b_{22} - b_{12},\ V_2 = b_{12} - b_{11},\ V_3 = b_{22} - V_2,\ V_4 = V_3 - b_{21}, \\ CMF_B^{WV}(B) = b_{11} \text{ for } n = 1 \\ CMF_B^{WV}(B) = \begin{pmatrix} CMF_B^{WV}(b_{11}), CMF_B^{WV}(b_{21}), CMF_B^{WV}(V_4), CMF_B^{WV}(V_1), \\ CMF_B^{WV}(V_2), CMF_B^{WV}(b_{22}), CMF_B^{WV}(V_3) \end{pmatrix} \text{ for } n \geq 2 \end{cases} \tag{2.5}$$

For an $n \times n$ matrix, WV's CMF is unrolled into 7 CMFs applied on half size matrices and it costs 4 additions of matrices of size $\frac{n}{2} \times \frac{n}{2}$. The complexity of CMF for WV is then:

$$M_{CMF}^{WV}(n) \leq 7M_{CMF}^{WV}\left(\frac{n}{2}\right) + 4\left(\frac{n}{2}\right)^2, \quad M_{CMF}^{WV}(1) = 0 \Longrightarrow M_{CMF}^{WV}(n) = \frac{4}{3}n^{\log_2 7} - \frac{4}{3}n^2.$$

Example 1. Consider an example when $n = 4$. We can split the matrix into four 2×2 sub-matrices as follows:

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

where

$$B_{11} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix},\ B_{12} = \begin{pmatrix} b_{13} & b_{14} \\ b_{23} & b_{24} \end{pmatrix},\ B_{21} = \begin{pmatrix} b_{31} & b_{32} \\ b_{41} & b_{42} \end{pmatrix},\ B_{22} = \begin{pmatrix} b_{33} & b_{34} \\ b_{43} & b_{44} \end{pmatrix}$$

In order to obtain V_1, V_2, V_3, V_4, we do linear combinations as:

$$V_1 = B_{22} - B_{12} = \begin{pmatrix} \underbrace{b_{33} - b_{13}}_{s_1} & \underbrace{b_{34} - b_{14}}_{s_2} \\ \underbrace{b_{43} - b_{23}}_{s_3} & \underbrace{b_{44} - b_{24}}_{s_4} \end{pmatrix},\ V_2 = B_{12} - B_{11} = \begin{pmatrix} \underbrace{b_{13} - b_{11}}_{s_5} & \underbrace{b_{14} - b_{12}}_{s_6} \\ \underbrace{b_{23} - b_{21}}_{s_7} & \underbrace{b_{24} - b_{22}}_{s_8} \end{pmatrix}$$

$$V_3 = B_{22} - V_2 = \begin{pmatrix} \underbrace{b_{33} - s_5}_{s_9} & \underbrace{b_{34} - s_6}_{s_{10}} \\ \underbrace{b_{43} - s_7}_{s_{11}} & \underbrace{b_{44} - s_8}_{s_{12}} \end{pmatrix},\quad V_4 = V_3 - B_{21} = \begin{pmatrix} \underbrace{s_9 - b_{31}}_{s_{13}} & \underbrace{s_{10} - b_{32}}_{s_{14}} \\ \underbrace{s_{11} - b_{41}}_{s_{15}} & \underbrace{s_{12} - b_{42}}_{s_{16}} \end{pmatrix}$$

Clearly, we can see that the total cost for obtaining V_i's and s_j's is 16 subtractions.

The next step is component matrix formation on these sub-matrices:

$$CMF^{WV}_{B_{11}} = (b_{11}, b_{21}, \underbrace{r_3 - b_{21}}_{r_4}, \underbrace{b_{22} - b_{12}}_{r_1}, \underbrace{b_{12} - b_{11}}_{r_2}, b_{22}, \underbrace{b_{22} - r_2}_{r_3}),$$

$$CMF^{WV}_{B_{21}} = (b_{31}, b_{41}, \underbrace{r_7 - b_{41}}_{r_8}, \underbrace{b_{42} - b_{32}}_{r_5}, \underbrace{b_{32} - b_{31}}_{r_6}, b_{42}, \underbrace{b_{42} - r_6}_{r_7}),$$

$$CMF^{WV}_{B_{22}} = (b_{33}, b_{43}, \underbrace{r_{11} - b_{43}}_{r_{12}}, \underbrace{b_{44} - b_{34}}_{r_9}, \underbrace{b_{34} - b_{33}}_{r_{10}}, b_{44}, \underbrace{b_{44} - r_{10}}_{r_{11}}),$$

$$CMF^{WV}_{V_1} = (s_1, s_3, \underbrace{r_{15} - s_3}_{r_{16}}, \underbrace{s_4 - s_2}_{r_{13}}, \underbrace{s_2 - s_1}_{r_{14}}, s_4, \underbrace{s_4 - r_{14}}_{r_{15}}),$$

$$CMF^{WV}_{V_2} = (s_5, s_7, \underbrace{r_{19} - s_7}_{r_{20}}, \underbrace{s_8 - s_6}_{r_{17}}, \underbrace{s_6 - s_5}_{r_{18}}, s_8, \underbrace{s_8 - r_{18}}_{r_{19}}),$$

$$CMF^{WV}_{V_3} = (s_9, s_{11}, \underbrace{r_{23} - s_{11}}_{r_{24}}, \underbrace{s_{12} - s_{10}}_{r_{21}}, \underbrace{s_{10} - s_9}_{r_{22}}, s_{12}, \underbrace{s_{12} - r_{22}}_{r_{23}}),$$

$$CMF^{WV}_{V_4} = (s_{13}, s_{15}, \underbrace{r_{27} - s_{15}}_{r_{28}}, \underbrace{s_{16} - s_{14}}_{r_{25}}, \underbrace{s_{14} - s_{13}}_{r_{26}}, s_{16}, \underbrace{s_{16} - r_{26}}_{r_{27}}).$$

What calls for special attention is that there are 28 subtractions needed to compute $CMF_{B_{11}}^{WV}$, $CMF_{B_{21}}^{WV}$, $CMF_{B_{22}}^{WV}$, $CMF_{V_1}^{WV}$, $CMF_{V_2}^{WV}$, $CMF_{V_3}^{WV}$, $CMF_{V_4}^{WV}$. Therefore, it requires 44 subtractions or additions to obtain original CMF_B^{WV}. It is 11 additions or subtractions less than that to obtain CMF_B^{SA}.

2.3.2 WV's Reconstruction

Following the component multiplication $CM(CMF_A^{WV}, CMF_B^{WV})$, it will generate a 7-tuple $C = (C_1, C_2, C_3, C_4, C_5, C_6, C_7)$ as input to the R block, which is defined below.

$$\begin{cases} R^{WV}(C) = C_1 \text{ for } n = 1 \\ R^{WV}(C) = \begin{pmatrix} R^{WV}(C_1) \oplus R^{WV}(C_2),\ \underbrace{S_1 \oplus R^{WV}(C_5)}_{S_2} \oplus R^{WV}(C_6), \\ \underbrace{R^{WV}(C_1) \ominus R^{WV}(C_7)}_{S_1} \ominus R^{WV}(C_3) \oplus R^{WV}(C_4),\ S_2 \oplus R^{WV}(C_4) \end{pmatrix} \text{ for } n \geq 2 \end{cases} \tag{2.6}$$

In the definition, there are 7 component additions or subtractions. Thus, we can calculate WV's Reconstruction complexity as:

$$M_R^{WV}(n) \leq 7M_R^{WV}\left(\frac{n}{2}\right) + 7\left(\frac{n}{2}\right)^2, \quad M_R^{WV}(1) = 0 \Longrightarrow M_R^{WV}(n) = \frac{7}{3}n^{\log_2 7} - \frac{7}{3}n^2.$$

Example 2. Consider the case where matrix dimension is 4. After component multiplication at the end of two rounds of recursion, the length of C will be 49. Here, we denote C as $(C_1, C_2, C_3, C_4, C_5, C_6, C_7)$ where $C_i = (P_{7i-6}, P_{7i-5}, \cdots, P_{7i})$, $i = 1, 2, \cdots, 7$. In order to obtain $R^{WV}(C)$, the first step is to compute $R^{WV}(C_i)$ as follows:

$$R^{WV}(C_1) = (P_1 + P_2, \underbrace{r_1 + P_5}_{r_2} + P_6, \underbrace{P_1 - P_7}_{r_1} - P_3 + P_4, r_2 + P_4),$$

$$R^{WV}(C_2) = (P_8 + P_9, \underbrace{r_3 + P_{12}}_{r_4} + P_{13}, \underbrace{P_8 - P_{14}}_{r_3} - P_{10} + P_{11}, r_4 + P_{11}),$$

$$R^{WV}(C_3) = (P_{15} + P_{16}, \underbrace{r_5 + P_{19}}_{r_6} + P_{20}, \underbrace{P_{15} - P_{21}}_{r_5} - P_{17} + P_{18}, r_6 + P_{18}),$$

$$R^{WV}(C_4) = (P_{22} + P_{23}, \underbrace{r_7 + P_{26}}_{r_8} + P_{27}, \underbrace{P_{22} - P_{28}}_{r_7} - P_{24} + P_{25}, r_8 + P_{25}),$$

$$R^{WV}(C_5) = (P_{29} + P_{30}, \underbrace{r_9 + P_{33}}_{r_{10}} + P_{34}, \underbrace{P_{29} - P_{35}}_{r_9} - P_{31} + P_{32}, r_{10} + P_{32}),$$

$$R^{WV}(C_6) = (P_{36} + P_{37}, \underbrace{r_{11} + P_{40}}_{r_{12}} + P_{41}, \underbrace{P_{36} - P_{42}}_{r_{11}} - P_{38} + P_{39}, r_{12} + P_{39}),$$

$$R^{WV}(C_7) = (P_{43} + P_{44}, \underbrace{r_{13} + P_{47}}_{r_{14}} + P_{48}, \underbrace{P_{43} - P_{49}}_{r_{13}} - P_{45} + P_{46}, r_{14} + P_{46}).$$

The above expressions cost 49 additions/subtractions. Furthermore, we need extra additions to compute L_i's:

$$L_1 = R^{WV}(C_1) \oplus R^{WV}(C_2),$$

$$L_2 = \underbrace{S_1 \oplus R^{WV}(C_5)}_{S_2} \oplus R^{WV}(C_6),$$

$$L_3 = \underbrace{R^{WV}(C_1) \ominus R^{WV}(C_7)}_{S_1} \ominus R^{WV}(C_3) \oplus R^{WV}(C_4),$$

$$L_4 = S_2 \oplus R^{WV}(C_4).$$

Clearly, there are 7 component operations in the above formula. As the size of $R^{WV}(C_i)$, S_1 and S_2 is 2×2, it will take 4 additions to do $\oplus$ or $\ominus$. Therefore, it takes 28 additions/subtractions in total to finally get L_i's where $i = 1, \cdots, 4$. As a result, computation of WV's $R(C)$ requires $49 + 28 = 77$ additions/subtractions.

Table 2.1: Complexity of various blocks of Strassen's algorithm and its Winograd's variant

Method	Operation	Recursive Formula		Complexity
Strassen's algorithm	CMF	$7M_{CMF}^{SA}\left(\frac{n}{2}\right)+5\left(\frac{n}{2}\right)^2$	$M_{CMF}^{SA}(1)=0$	$\frac{5}{3}n^{\log_2 7}-\frac{5}{3}n^2$
	CM	$7M_{CM}^{SA}\left(\frac{n}{2}\right)$	$M_{CM}^{SA}(1)=1$	$n^{\log_2 7}$
	R	$7M_{R}^{SA}\left(\frac{n}{2}\right)+8\left(\frac{n}{2}\right)^2$	$M_{R}^{SA}(1)=0$	$\frac{8}{3}n^{\log_2 7}-\frac{8}{3}n^2$
Winograd's variant	CMF	$7M_{CMF}^{WV}\left(\frac{n}{2}\right)+4\left(\frac{n}{2}\right)^2$	$M_{CMF}^{WV}(1)=0$	$\frac{4}{3}n^{\log_2 7}-\frac{4}{3}n^2$
	CM	$7M_{CM}^{WV}\left(\frac{n}{2}\right)$	$M_{CM}^{WV}(1)=1$	$n^{\log_2 7}$
	R	$7M_{R}^{WV}\left(\frac{n}{2}\right)+7\left(\frac{n}{2}\right)^2$	$M_{R}^{WV}(1)=0$	$\frac{7}{3}n^{\log_2 7}-\frac{7}{3}n^2$

2.4 Complexity Comparison

As discussed above, the decomposition method divides Strassen-like algorithms into four sub-blocks: $M_1 = CMF_A(A)$, $M_2 = CMF_B(B)$, $M_3 = CM(M_1, M_2)$, $M_4 = R(M_3)$. From the above Table 2.1, we can write the total complexity of SA and WV as follows:

$$\begin{aligned} M^{SA}(n) &= 2M_{CMF}^{SA}(n) + M_{CM}^{SA}(n) + M_{R}^{SA}(n) \\ &= 2\left(\frac{5}{3}n^{\log_2 7} - \frac{5}{3}n^2\right) + n^{\log_2 7} + \frac{8}{3}n^{\log_2 7} - \frac{8}{3}n^2 \\ &= 7n^{\log_2 7} - 6n^2 \end{aligned} \tag{2.7}$$

$$\begin{aligned} M^{WV}(n) &= 2M_{CMF}^{WV}(n) + M_{CM}^{WV}(n) + M_{R}^{WV}(n) \\ &= 2\left(\frac{4}{3}n^{\log_2 7} - \frac{4}{3}n^2\right) + n^{\log_2 7} + \frac{7}{3}n^{\log_2 7} - \frac{7}{3}n^2 \\ &= 6n^{\log_2 7} - 5n^2 \end{aligned} \tag{2.8}$$

Chapter 3

Recent Methods to Improve Strassen-like Algorithms

Recently, arithmetic complexities of SA and WV have been reduced by Cenk and Hasan [7]. Their approach is based on the linearity property of CMF and R operations, block recombination, and limited recursion. The linearity property of CMF and R operations refers to the superposition principle for reducing the number of addition operations. Limited recursion means the hybrid use of Strassen-like algorithms and the naive method with different cut-off values. It exploits to discover best performance by the trade-off of multiplications and additions. Block recombination is a method to explore the effect of rearranging of blocks. Reordering the data flow between two blocks could reduce the number of operation blocks.

3.1 Linearity Property of CMF and R Operations

As discussed in the previous chapter, to multiply two matrices A and B of size $n \times n$, the CMF operation computes all the necessary linear combinations of A_{ij}'s and B_{ij}'s, and the R operation linearly combines the vector products produced from CM.

For the case of $n = 1$, we have $A = a_{11}$, $B = b_{11}$. It is easy to see that:

$$\begin{aligned} CMF(A+B) &= CMF(a_{11}+b_{11}) \\ &= a_{11}+b_{11} \\ &= CMF(a_{11})+CMF(b_{11}) \\ &= CMF(A)+CMF(B) \end{aligned} \tag{3.1}$$

For the case of $n = N$, we have $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$, where A_{ij} and B_{ij} are sub-matrices of size $\frac{N}{2} \times \frac{N}{2}$. It is easy to see that:

$$\begin{aligned} CMF(A+B) &= CMF\left(\begin{pmatrix} A_{11}+B_{11} & A_{12}+B_{12} \\ A_{21}+B_{21} & A_{22}+B_{22} \end{pmatrix}\right) \\ &= (\underbrace{CMF(A_{11}+B_{11}), \cdots, CMF(A_{22}+B_{22})}_{7\ CMFs}) \\ &= (\underbrace{CMF(A_{11}), \cdots, CMF(A_{22})}_{7\ CMFs}) + (\underbrace{CMF(B_{11}), \cdots, CMF(B_{22})}_{7\ CMFs}) \\ &= CMF(A)+CMF(B) \end{aligned} \tag{3.2}$$

It clearly proves the linearity property of the CMF operation. The same induction method could also be used to prove that linearity property holds for the R operation.

3.1.1 SA's CMF Block based on Linearity

Based on the linearity property, SA's improved CMFs are illustrated as follows:

$$\begin{cases} U_1 = a_{11} + a_{22}, \quad U_2 = a_{21} + a_{22}, \quad U_3 = a_{11} + a_{12} \\ CMF_A^{SA}(A) = a_{11} \text{ for } n = 1 \\ CMF_A^{SA}(A) = \begin{pmatrix} CMF_A^{SA}(U_1),\ CMF_A^{SA}(U_2),\ CMF_A^{SA}(a_{11}), \\ CMF_A^{SA}(a_{22}),\ CMF_A^{SA}(U_3), \\ \underbrace{CMF_A^{SA}(U_2) \ominus CMF_A^{SA}(U_1)}_{Q_1}, \\ \underbrace{CMF_A^{SA}(U_3) \ominus CMF_A^{SA}(U_1)}_{Q_2} \end{pmatrix} \text{ for } n \geq 2 \end{cases} \tag{3.3}$$

$$\begin{cases} V_1 = b_{11} + b_{22}, \quad V_2 = b_{12} - b_{22}, \quad V_3 = b_{21} - b_{11} \\ CMF_B^{SA}(B) = b_{11} \text{ for } n = 1 \\ CMF_B^{SA}(B) = \begin{pmatrix} CMF_B^{SA}(V_1),\ CMF_B^{SA}(b_{11}),\ CMF_B^{SA}(V_2), \\ CMF_B^{SA}(V_3),\ CMF_B^{SA}(b_{22}), \\ \underbrace{CMF_B^{SA}(V_1) \oplus CMF_B^{SA}(V_2)}_{Q_3} \\ \underbrace{CMF_B^{SA}(V_1) \oplus CMF_B^{SA}(V_3)}_{Q_4} \end{pmatrix} \text{ for } n \geq 2 \end{cases} \tag{3.4}$$

It is a fact that:

$$CMF_A^{SA}(U_2) \ominus CMF_A^{SA}(U_1) = CMF_A^{SA}(U_2 - U_1) = CMF_A^{SA}(a_{21} - a_{11})$$

$$CMF_A^{SA}(U_3) \ominus CMF_A^{SA}(U_1) = CMF_A^{SA}(U_3 - U_1) = CMF_A^{SA}(a_{12} - a_{22})$$

$$CMF_B^{SA}(V_1) \oplus CMF_B^{SA}(V_2) = CMF_B^{SA}(V_1 + V_2) = CMF_B^{SA}(b_{11} + b_{12})$$

$$CMF_B^{SA}(V_1) \oplus CMF_B^{SA}(V_3) = CMF_B^{SA}(V_1 + V_3) = CMF_B^{SA}(b_{21} + b_{22})$$

Compared to SA's original CMF given in subsection 2.2.1, the new CMF given here requires fewer numbers of U_i's, V_i's, and CMFs, but it introduces Q_i's. In order to determine the complexity of the new CMF, one can note that the cost of $\oplus$ or $\ominus$ is $7^{(\log_2 n)-1}$, since the length of vector Q_i is $\frac{1}{7}n^{\log_2 7}$. It is also important to note that CMF applied on an $n \times n$ matrix is only unrolled into 5 CMFs applied on $\frac{n}{2} \times \frac{n}{2}$ matrices. So the complexity of the new CMF can be expressed as:

$$\begin{cases} M^{SA}_{CMF}(n) = 0 \text{ for } n = 1 \\ M^{SA}_{CMF}(n) \leq 5M^{SA}_{CMF}\left(\frac{n}{2}\right) + \frac{2}{7}n^{\log_2 7} + 3\left(\frac{n}{2}\right)^2 \text{ for } n \geq 2 \end{cases} \Longrightarrow M^{SA}_{CMF}(n) = n^{\log_2 7} + 2n^{\log_2 5} - 3n^2 \tag{3.5}$$

Example 3. This example will introduce the process to generate improved CMF operation on matrix A for SA when $n = 4$. Let A, and its sub-matrices A_{ij}'s be:

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

$$U_1 = A_{11} + A_{22} = \begin{pmatrix} \underbrace{a_{11} + a_{33}}_{s_1} & \underbrace{a_{12} + a_{34}}_{s_2} \\ \underbrace{a_{21} + a_{43}}_{s_3} & \underbrace{a_{22} + a_{44}}_{s_4} \end{pmatrix},$$

$$U_2 = A_{21} + A_{22} = \begin{pmatrix} \underbrace{a_{31} + a_{33}}_{s_5} & \underbrace{a_{32} + a_{34}}_{s_6} \\ \underbrace{a_{41} + a_{43}}_{s_7} & \underbrace{a_{42} + a_{44}}_{s_8} \end{pmatrix},$$

$$U_3 = A_{11} + A_{12} = \begin{pmatrix} \underbrace{a_{11} + a_{13}}_{s_9} & \underbrace{a_{12} + a_{14}}_{s_{10}} \\ \underbrace{a_{21} + a_{23}}_{s_{11}} & \underbrace{a_{22} + a_{24}}_{s_{12}} \end{pmatrix}.$$

It should be noted that in order to obtain U_1, U_2 and U_3, we need to perform 12 additions/subtractions since each sub-matrix has 4 entries. Thus, the next step is to compute the CMFs applied on A's sub-matrices with size of 2×2:

$$CMF_A^{SA}(U_1) = (\underbrace{s_1 + s_4}_{r_1}, \underbrace{s_3 + s_4}_{r_2}, s_1, s_4, \underbrace{s_1 + s_2}_{r_3}, \underbrace{r_2 - r_1}_{r_{16}}, \underbrace{r_3 - r_1}_{r_{17}}),$$

$$CMF_A^{SA}(U_2) = (\underbrace{s_5 + s_8}_{r_4}, \underbrace{s_7 + s_8}_{r_5}, s_5, s_8, \underbrace{s_5 + s_6}_{r_6}, \underbrace{r_5 - r_4}_{r_{18}}, \underbrace{r_6 - r_4}_{r_{19}}),$$

$$CMF_A^{SA}(A_{11}) = (\underbrace{a_{11} + a_{22}}_{r_7}, \underbrace{a_{21} + a_{22}}_{r_8}, a_{11}, a_{22}, \underbrace{a_{11} + a_{12}}_{r_9}, \underbrace{r_8 - r_7}_{r_{20}}, \underbrace{r_9 - r_7}_{r_{21}}),$$

$$CMF_A^{SA}(A_{22}) = (\underbrace{a_{33} + a_{44}}_{r_{10}}, \underbrace{a_{43} + a_{44}}_{r_{11}}, a_{33}, a_{44}, \underbrace{a_{33} - a_{34}}_{r_{12}}, \underbrace{r_{11} - r_{10}}_{r_{22}}, \underbrace{r_{12} - r_{10}}_{r_{23}}),$$

$$CMF_A^{SA}(U_3) = (\underbrace{s_9 + s_{12}}_{r_{13}}, \underbrace{s_{11} + s_{12}}_{r_{14}}, s_9, s_{12}, \underbrace{s_9 + s_{10}}_{r_{15}}, \underbrace{r_{14} - r_{13}}_{r_{24}}, \underbrace{r_{15} - r_{13}}_{r_{25}}),$$

$$Q_1 = (\underbrace{r_4 + r_1}_{r_{26}}, \underbrace{r_5 - r_2}_{r_{27}}, \underbrace{s_5 - s_1}_{r_{28}}, \underbrace{s_8 - s_4}_{r_{29}}, \underbrace{r_6 - r_3}_{r_{30}}, \underbrace{r_{18} - r_{16}}_{r_{31}}, \underbrace{r_{19} - r_{17}}_{r_{32}}),$$

$$Q_2 = (\underbrace{r_{13} - r_1}_{r_{33}}, \underbrace{r_{14} - r_2}_{r_{34}}, \underbrace{s_9 - s_1}_{r_{35}}, \underbrace{s_{12} - s_4}_{r_{36}}, \underbrace{r_{15} - r_3}_{r_{37}}, \underbrace{r_{20} - r_{16}}_{r_{38}}, \underbrace{r_{21} - r_{17}}_{r_{39}}).$$

As clearly shown above, it takes 39 additions/subtractions to compute r_i's, for $i = 1, \cdots, 39$ and 12 additions/subtractions to compute s_j's, for $j = 1, \cdots, 12$. Thus, it in total needs 51 additions/subtractions which is 4 less than the original SA's CMF.

3.1.2 WV's CMF Block based on Linearity

Applying the linearity property to WV's CMFs, we can write:

$$\begin{cases} U_1 = a_{11} - a_{21},\ U_2 = a_{21} + a_{22} \\ CMF_A^{WV}(A) = a_{11} \text{ for } n = 1 \\ CMF_A^{WV}(A) = \begin{pmatrix} CMF_A^{WV}(a_{11}),\ CMF_A^{WV}(a_{12}),\ CMF_A^{WV}(a_{22}), \\ CMF_A^{WV}(U_1),\ CMF_A^{WV}(U_2),\ \underbrace{T_1 \oplus CMF_A^{WV}(a_{12})}_{T_2}, \\ \underbrace{CMF_A^{WV}(U_1) \ominus CMF_A^{WV}(a_{22})}_{T_1} \end{pmatrix} \text{ for } n \geq 2 \end{cases} \tag{3.6}$$

$$\begin{cases} V_1 = b_{22} - b_{12},\ V_2 = b_{12} - b_{11} \\ CMF_B^{WV}(B) = b_{11} \text{ for } n = 1 \\ CMF_B^{WV}(B) = \begin{pmatrix} CMF_B^{WV}(b_{11}),\ CMF_B^{WV}(b_{21}),\ \underbrace{T_3 \ominus CMF_B^{WV}(b_{21})}_{T_4}, \\ CMF_B^{WV}(V_1),\ CMF_B^{WV}(V_2),\ CMF_B^{WV}(b_{22}) \\ \underbrace{CMF_B^{WV}(b_{22}) \ominus CMF_B^{WV}(V_2)}_{T_3} \end{pmatrix} \text{ for } n \geq 2 \end{cases} \tag{3.7}$$

It should be noted that:

$$T_1 = CMF_A^{WV}(U_1) \ominus CMF_A^{WV}(a_{22}) = CMF_A^{WV}(a_{11} - a_{21} - a_{22})$$

$$T_2 = T_1 \oplus CMF_A^{WV}(a_{12}) = CMF_A^{WV}(a_{11} - a_{21} - a_{22} + a_{12})$$

$$T_3 = CMF_B^{WV}(b_{22}) \ominus CMF_B^{WV}(V_2) = CMF_B^{WV}(b_{22} - b_{12} + b_{11})$$

$$T_4 = T_3 \ominus CMF_B^{WV}(b_{21}) = CMF_B^{WV}(b_{22} - b_{12} + b_{11} - b_{21})$$

The CMF applied on $n \times n$ matrix is unrolled into 5 CMFs applied on half size matrices. It only needs 2 component matrix additions to compute U_i's or V_i's, respectively.

Also it costs $\frac{1}{7}n^{\log_2 7}$ additions/subtractions to obtain T_1, T_2, T_3 and T_4, respectively. So the complexity of the new CMF for WV is:

$$\begin{cases} M_{CMF}^{WV}(n) = 0 \text{ for } n = 1 \\ M_{CMF}^{WV}(n) \leq 5M_{CMF}^{WV}\left(\frac{n}{2}\right) + \frac{2}{7}n^{\log_2 7} + 2\left(\frac{n}{2}\right)^2 \text{ for } n \geq 2 \end{cases} \Longrightarrow M_{CMF}^{WV}(n) = n^{\log_2 7} + n^{\log_2 5} - 2n^2 \tag{3.8}$$

Example 4. Consider A and its sub-matrices A_{ij}'s as:

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

$$U_1 = A_{11} - A_{21} = \begin{pmatrix} \underbrace{a_{11} - a_{31}}_{s_1} & \underbrace{a_{12} - a_{32}}_{s_2} \\ \underbrace{a_{21} - a_{41}}_{s_3} & \underbrace{a_{22} - a_{42}}_{s_4} \end{pmatrix}, \quad U_2 = A_{21} + A_{22} = \begin{pmatrix} \underbrace{a_{31} + a_{33}}_{s_5} & \underbrace{a_{32} + a_{34}}_{s_6} \\ \underbrace{a_{41} + a_{43}}_{s_7} & \underbrace{a_{42} + a_{44}}_{s_8} \end{pmatrix}.$$

It costs 8 additions/subtractions in total to obtain s_1, s_2, $\cdots$, s_8. The computation of CMFs applied to A's sub-matrices with size of 2×2 is shown as follows:

$$CMF_A^{WV}(A_{11}) = (a_{11}, a_{12}, a_{22}, \underbrace{a_{11} - a_{21}}_{r_1}, \underbrace{a_{21} + a_{22}}_{r_2}, \underbrace{r_3 + a_{12}}_{r_4}, \underbrace{r_1 - a_{22}}_{r_3}),$$

$$CMF_A^{WV}(A_{12}) = (a_{13}, a_{14}, a_{24}, \underbrace{a_{13} - a_{23}}_{r_5}, \underbrace{a_{23} + a_{24}}_{r_6}, \underbrace{r_7 + a_{14}}_{r_8}, \underbrace{r_5 - a_{24}}_{r_7}),$$

$$CMF_A^{WV}(A_{22}) = (a_{33}, a_{34}, a_{44}, \underbrace{a_{33} - a_{43}}_{r_9}, \underbrace{a_{43} + a_{44}}_{r_{10}}, \underbrace{r_{11} + a_{34}}_{r_{12}}, \underbrace{r_9 - a_{44}}_{r_{11}}),$$

$$CMF_A^{WV}(U_1) = (s_1, s_2, s_4, \underbrace{s_1 - s_3}_{r_{13}}, \underbrace{s_3 + s_4}_{r_{14}}, \underbrace{r_{15} + s_2}_{r_{16}}, \underbrace{r_{13} - s_4}_{r_{15}}),$$

$$CMF_A^{WV}(U_2) = (s_5, s_6, s_8, \underbrace{s_5 - s_7}_{r_{17}}, \underbrace{s_7 + s_8}_{r_{18}}, \underbrace{r_{19} + s_6}_{r_{20}}, \underbrace{r_{17} - s_8}_{r_{19}}),$$

$$T_1 = (\underbrace{s_1 - a_{33}}_{r_{21}}, \underbrace{s_2 - a_{34}}_{r_{22}}, \underbrace{s_4 - a_{44}}_{r_{23}}, \underbrace{r_{13} - r_9}_{r_{24}}, \underbrace{r_{14} - r_{10}}_{r_{25}}, \underbrace{r_{16} - r_{12}}_{r_{26}}, \underbrace{r_{15} - r_{11}}_{r_{27}}),$$
$$T_2 = (\underbrace{r_{21} + a_{13}}_{r_{28}}, \underbrace{r_{22} + a_{14}}_{r_{29}}, \underbrace{r_{23} + a_{24}}_{r_{30}}, \underbrace{r_{24} + r_5}_{r_{31}}, \underbrace{r_{25} + r_6}_{r_{32}}, \underbrace{r_{26} + r_8}_{r_{33}}, \underbrace{r_{27} + r_7}_{r_{34}}).$$

Thus, it needs 42 additions/subtractions in total. Clearly, the number of additions/subtractions is now 2 less than that in the original CMF for WV.

3.1.3 WV's R Block based on Linearity

As linearity property holds for R operation, it is important to investigate its effect on the complexity of R. Assume that $C = (C_1, C_2, C_3, C_4, C_5, C_6, C_7)$ where the length of each C_i is $\frac{1}{7}n^{\log_2 7}$ for $i = 1, \cdots, 7$. Applying the linearity property, R can be re-stated as:

$$\begin{cases} R(C) = C_1 \text{ for } n = 1 \\ R_1 = C_1 + C_2,\ R_2 = R(R_1),\ R_3 = C_1 - C_7,\ R_4 = R(R_3),\ R_5 = R(C_5) \\ R_6 = R_4 + R_5,\ R_7 = R(C_6),\ R_8 = R_6 + R_7,\ R_9 = R(C_3) \\ R_{10} = R(C_4),\ R_{11} = R_{10} - R_9,\ R_{12} = R_4 + R_{11},\ R_{13} = R_6 + R_{10}, \\ R(C) = (R_2,\ R_8,\ R_{12},\ R_{13}) \text{ for } n \geq 2 \end{cases} \tag{3.9}$$

Below is a verification of the correctness of R operation given in Equation (3.9).

$$\begin{aligned} R(C) &= (R_2,\ R_8,\ R_{12},\ R_{13}) \\ &= (R(C_1 + C_2),\ R_6 + R_7,\ R_4 + R_{11},\ R_6 + R_{10}) \\ &= (R(C_1 + C_2),\ R_4 + R_5 + R_7,\ R_4 + R_{10} - R_9,\ R_4 + R_5 + R(C_4)) \\ &= (R(C_1 + C_2),\ R(C_1 - C_7) + R(C_5) + R(C_6), \\ &\qquad R(C_1 - C_7) + R(C_4) - R(C_3),\ R(C_1 - C_7) + R(C_5) + R(C_4)) \\ &= (R(C_1) + R(C_2),\ R(C_1) - R(C_7) + R(C_5) + R(C_6), \\ &\qquad R(C_1) - R(C_7) + R(C_4) - R(C_3),\ R(C_1) - R(C_7) + R(C_5) + R(C_4)) \end{aligned} \tag{3.10}$$

Thus, the result is the same as that in Equation (2.6). But the cost of implementing Equation (3.9) is less than that of Equation (2.6). This can be explained as follows. In Equation (3.9), $R(C)$ is the output from block Reconstruction, whose size is $n \times n$, and $R(C_i)$ or $R(R_i)$ is of size $\frac{n}{2} \times \frac{n}{2}$. R_1 and R_3 are each simply the result of addition/subtraction involving two C_i's. So the arithmetic cost to obtain R_1 and R_3 is $\frac{2}{7}n^{\log_2 7}$. R_2, R_4, R_5, R_7, R_9 and R_{10} are 6 $R(\frac{n}{2})$'s. Computation of R_6, R_8, R_{11}, R_{12} and R_{13} is matrix addition/subtraction on matrices of size $\frac{n}{2} \times \frac{n}{2}$. Thus the cost is $5(\frac{n}{2})^2$. Therefore, we have a conclusion that:

$$\begin{cases} M_R^{WV}(n) = 0 \text{ for } n = 1 \\ M_R^{WV}(n) \leq 6M_R^{WV}\left(\frac{n}{2}\right) + \frac{2}{7}n^{\log_2 7} + 5\left(\frac{n}{2}\right)^2 \text{ for } n \geq 2 \end{cases} \Longrightarrow M_R^{WV}(n) = 2n^{\log_2 7} + \frac{n}{2}n^{\log_2 6} - \frac{5}{2}n^2. \tag{3.11}$$

Example 5. Consider the case $n = 4$. The length of R's input is 49, which is also the output of CM. Consider it as $C = (C_1, C_2, C_3, C_4, C_5, C_6, C_7)$ where $C_i = (P_{7i-6}, \cdots, P_{7i})$. The R's computing process is as follows:

$$R_1 = C_1 + C_2 = (\underbrace{P_1 + P_8}_{r_1}, \underbrace{P_2 + P_9}_{r_2}, \underbrace{P_3 + P_{10}}_{r_3}, \underbrace{P_4 + P_{11}}_{r_4}, \underbrace{P_5 + P_{12}}_{r_5}, \underbrace{P_6 + P_{13}}_{r_6}, \underbrace{P_7 + P_{14}}_{r_7}),$$

$$R_2 = R(R_1) = (r_1 + r_2, \underbrace{r_8 + r_5}_{r_9} + r_6, \underbrace{r_1 - r_7}_{r_8} - r_3 + r_4, r_9 + r_4),$$

$$R_3 = C_1 - C_7 = (\underbrace{P_1 - P_{43}}_{r_{10}}, \underbrace{P_2 - P_{44}}_{r_{11}}, \underbrace{P_3 - P_{45}}_{r_{12}}, \underbrace{P_4 - P_{46}}_{r_{13}}, \underbrace{P_5 - P_{47}}_{r_{14}}, \underbrace{P_6 - P_{48}}_{r_{15}}, \underbrace{P_7 - P_{49}}_{r_{16}}),$$

$$R_4 = R(R_3) = (r_{10} + r_{11}, \underbrace{r_{17} + r_{14}}_{r_{18}} + r_{15}, \underbrace{r_{10} - r_{16}}_{r_{17}} - r_{12} + r_{13}, r_{18} + r_{13}),$$

$$R_5 = R(C_5) = (P_{29} + P_{30}, \underbrace{r_{19} + P_{33}}_{r_{20}} + P_{34}, \underbrace{P_{29} - P_{35}}_{r_{19}} - P_{31} + P_{32}, r_{20} + P_{32}),$$

$$R_6 = R_4 + R_5 = (s_1, s_2, s_3, s_4), \ R_8 = R_6 + R_7 = (s_5, s_6, s_7, s_8,)$$

$$R_7 = R(C_6) = (P_{36} + P_{37}, \underbrace{r_{21} + P_{40}}_{r_{22}} + P_{41}, \underbrace{P_{36} - P_{42}}_{r_{21}} - P_{38} + P_{39}, r_{22} + P_{39}),$$

$$R_9 = R(C_3) = (P_{15} + P_{16}, \underbrace{r_{23} + P_{19}}_{r_{24}} + P_{20}, \underbrace{P_{15} - P_{21}}_{r_{23}} - P_{17} + P_{18}, r_{24} + P_{18}),$$

$$R_{10} = R(C_4) = (P_{22} + P_{23}, \underbrace{r_{25} + P_{26}}_{r_{26}} + P_{27}, \underbrace{P_{22} - P_{28}}_{r_{25}} - P_{24} + P_{25}, r_{26} + P_{25}),$$

$$R_{11} = R_{10} - R_9 = (s_9, s_{10}, s_{11}, s_{12}), \;\; R_{12} = R_4 + R_{11} = (s_{13}, s_{14}, s_{15}, s_{16})$$

$$R_{13} = R_6 + R_{10} = (s_{17}, s_{18}, s_{19}, s_{20})$$

In the computation process, it requires 7 additions/subtractions in order to get R_1, R_2, R_3, R_4, R_5, R_7, R_9, R_{10} respectively and 4 additions/subtractions to get R_6, R_8, R_{11}, R_{12}, R_{13} respectively. The total arithmetic operation cost is therefore 76, which is 1 less than the original WV's Reconstruction algorithm.

3.1.4 Complexity Comparison

Table 3.1 lists the complexities of each block operation in its original form and after applying the linearity property. As it can be seen in the table, the application of the linearity property lowers the complexity of each block operation.

Table 3.1: Complexity Comparison

Operation	Original form	After linearity property applied
SA's CMF	$\frac{5}{3}n^{\log_2 7} - \frac{5}{3}n^2$	$n^{\log_2 7} + 2n^{\log_2 5} - 3n^2$
WV's CMF	$\frac{4}{3}n^{\log_2 7} - \frac{4}{3}n^2$	$n^{\log_2 7} + n^{\log_2 5} - 2n^2$
WV's R	$\frac{7}{3}n^{\log_2 7} - \frac{7}{3}n^2$	$2n^{\log_2 7} + \frac{1}{2}n^{\log_2 6} - \frac{5}{2}n^2$

3.2 Block Recombination of Improved Winograd's Variant

Block recombination means to reorganize the block decomposition of Strassen-like algorithms. It usually associates with limited recursion to achieve better performance.

3.2.1 Block Recombination Method

Let matrices $A_{2n\times 2n}$, $B_{2n\times 2n}$, and $C_{2n\times 2n}$ be:

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} \end{pmatrix}.$$

We know that the matrix multiplication is reduced to the computation of four separate instances of multiply-and-add, i.e., $a_{11}b_{11}+a_{12}b_{21}$, $a_{11}b_{12}+a_{12}b_{22}$, $a_{21}b_{11}+a_{22}b_{21}$, and $a_{21}b_{12}+a_{22}b_{22}$. Applying a Strassen-like algorithm to an instance, for example: $a_{11}b_{11}+a_{12}b_{21}$, the original block organization (*OBO*) computation flow is:

$$Q_1 = CMF_A(a_{11}),\ Q_2 = CMF_B(b_{11}),\ Q_3 = CMF_A(a_{12}),$$

$$Q_4 = CMF_B(b_{21}),\ Q_5 = CM(Q_1, Q_2),\ Q_6 = CM(Q_3, Q_4),$$

$$Q_7 = R(Q_5),\ Q_8 = R(Q_6),\ Q_9 = Q_7 + Q_8.$$

There are 9 block operations in the computation process which determines the arithmetic cost. Note that Q_9 requires a matrix addition (MA), whose complexity is n^2. Thus, the total arithmetic cost to compute $a_{11}b_{11}+a_{12}b_{21}$ is:

$$M(n) = 4M_{CMF} + 2M_{CM} + 2M_R + M_{MA}.$$

The further improvement of block recombination is built on the operation: $Q_9 = Q_7 + Q_8$. Since $Q_7 = R(Q_5)$ and $Q_8 = R(Q_6)$, we have: $Q_9 = R(Q_5) + R(Q_6)$. It is the linearity property of R that: $R(Q_5) + R(Q_6) = R(Q_5 + Q_6)$. Since vectors Q_5 and Q_6 are each of length $n^{\log_2 7}$, the cost to compute $R(Q_5 + Q_6)$ is $M_R + n^{\log_2 7}$. The improved computation flow, denoted as Block Recombination (*BR*), is shown in Figure 3.1, where blocks MA and VA are for matrix and vector addition and their arithmetic costs are n^2 and $\log_2 7$, respectively.

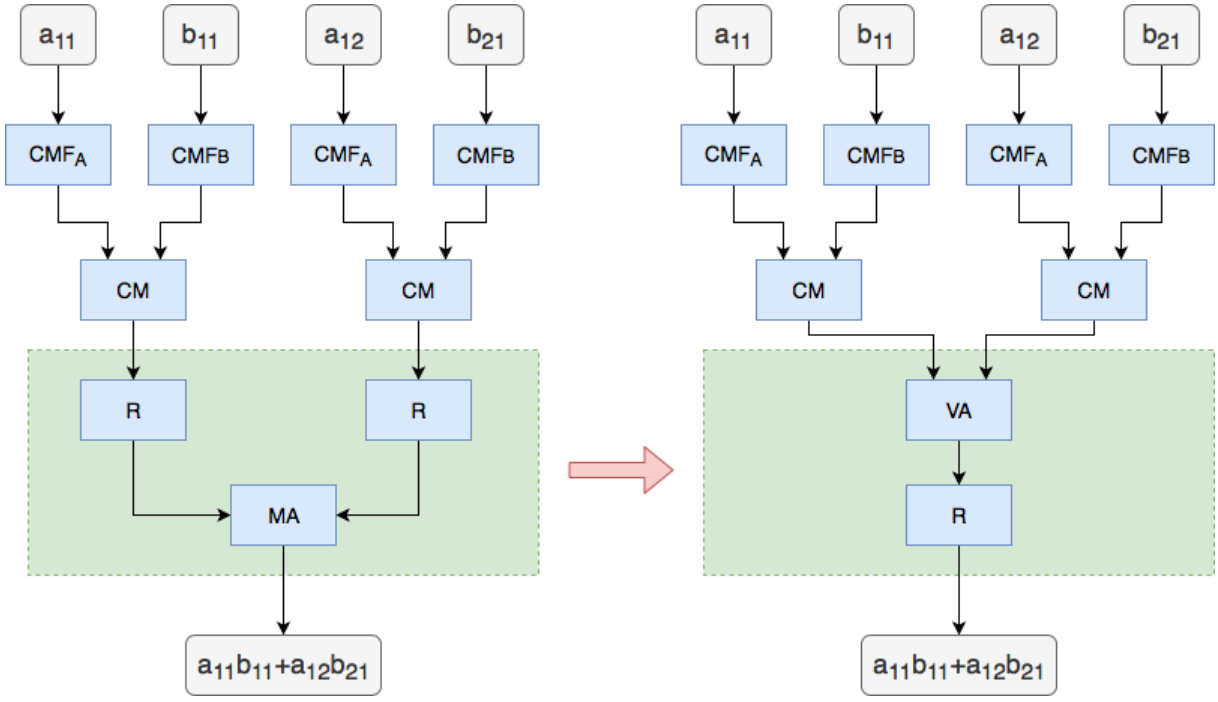

Figure 3.1: Architecture of Block Recombination

So the arithmetic cost of the improved architecture is: $M(n) = 4M_{CMF} + 2M_{CM} + M_{VA} + M_R$.

Detailed complexities of original block organization (OBO) and block recombina-

tion (BR) used with different Strassen-like methods to compute $a_{11}b_{11} + a_{12}b_{21}$ are shown in Table 3.2. Each complexity could be derived from Table 2.1 and 3.1. Improved SA is to apply linearity property of CMF on Strassen's algorithm and Improved WV is to apply linearity property of CMF and R on Winograd's variant.

Table 3.2: Complexity Comparison of OBO and BR

Algorithm	Complexity of OBO	Complexity of BR
SA	$14n^{\log_2 7} - 11n^2$	$\frac{37}{3}n^{\log_2 7} - \frac{28}{3}n^2$
Improved SA	$\frac{34}{3}n^{\log_2 7} + 8n^{\log_2 5} - \frac{49}{3}n^2$	$\frac{29}{3}n^{\log_2 7} + 8n^{\log_2 5} - 22n^2$
WV	$12n^{\log_2 7} - 9n^2$	$\frac{32}{3}n^{\log_2 7} - \frac{23}{3}n^2$
Improved WV	$10n^{\log_2 7} + n^{\log_2 6} + 4n^{\log_2 5} - 12n^2$	$9n^{\log_2 7} + \frac{1}{2}n^{\log_2 6} + 4n^{\log_2 5} - \frac{21}{2}n^2$

3.2.2 Block Recombination with Limited Recursion

The way to combine block recombination and limited recursion gives the best arithmetic complexity. For the case when limited recursion value is $m = 2$, the input matrices A and B are initially formed into 2×2 blocked matrices, so we can write:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}.$$

Thus, the matrix multiplication of $n \times n$ matrices is transformed into eight matrix multiplications: $a_{11}b_{11}$, $a_{12}b_{21}$, $a_{11}b_{12}$, $a_{12}b_{22}$, $a_{21}b_{11}$, $a_{22}b_{21}$, $a_{21}b_{12}$ and $a_{22}b_{22}$, and four $\frac{n}{2} \times \frac{n}{2}$ matrix additions. Then we apply Strassen-like algorithms on the sub-matrix multiplications. The first step is to obtain: $CMF(a_{11})$, $CMF(a_{12})$, $CMF(a_{21})$, $CMF(a_{22})$, $CMF(b_{11})$, $CMF(b_{12})$, $CMF(b_{21})$, $CMF(b_{22})$ and therefore the total cost is $8M_{CMF}\left(\frac{n}{2}\right)$.

The next step is to do component multiplications on those CMFs, and its costs is $8M_{CM}\left(\frac{n}{2}\right)$. After getting those vector products, we put the VA in front of R. It needs 4 VAs. The cost will be $4\left(\frac{n}{2}\right)^{\log_2 7}$. The last step is to implement R operations on those four sums. The cost is $4M_R\left(\frac{n}{2}\right)$. The total cost in recursive format is:

$$M(n) = 8M_{CMF}\left(\frac{n}{2}\right) + 8M_{CM}\left(\frac{n}{2}\right) + 4M_{VA}\left(\frac{n}{2}\right) + 4M_R\left(\frac{n}{2}\right).$$

Since Winograd's variant always gives better arithmetic complexity, in the following sections we only discuss the hybrid use of limited recursion and block recombination applied on improved Winograd's variant. We can refer to the complexity of each block in Table 2.1 and 3.1. So when the limited recursion value is $m = 2$, the improved WV algorithm obtains a better complexity:

$$M(n) = 4n^{\log_2 7} + \frac{1}{3}n^{\log_2 6} + \frac{8}{5}n^{\log_2 5} - \frac{13}{2}n^2.$$

In order to make a general instance, if the initially formed blocked matrices are of size $2^i \times 2^i$, the authors of [7] concluded the improved arithmetic complexity as:

$$M(n) = 2^{2i+1}M_{CMF}\left(\frac{n}{2^i}\right) + 2^{3i}M_{CM}\left(\frac{n}{2^i}\right) + \left(2^{3i} - 2^{2i}\right)M_{VA}\left(\frac{n}{2^i}\right) + 2^{2i}M_R\left(\frac{n}{2^i}\right).$$

Table 3.3 shows the complexities with different limited recursion values. From the table, we see that when $i = 3$, i.e., $m = 8$, the matrices are initially divided into $2^3 \times 2^3$ blocked matrices, the IWV would provide the best arithmetic complexity.

Table 3.3: Complexities for IWV with Different Limited Recursion Values

i	1	2	3	4
CMF	$\frac{8}{7}n^{\log_2 7}+\frac{8}{5}n^{\log_2 5}$-$4n^2$	$\frac{32}{49}n^{\log_2 7}+\frac{32}{25}n^{\log_2 5}$-$4n^2$	$\frac{128}{343}n^{\log_2 7}+\frac{128}{125}n^{\log_2 5}$-$4n^2$	$\frac{512}{2401}n^{\log_2 7}+\frac{512}{625}n^{\log_2 5}$-$4n^2$
CM	$\frac{8}{7}n^{\log_2 7}$	$\frac{64}{49}n^{\log_2 7}$	$\frac{512}{343}n^{\log_2 7}$	$\frac{4096}{2401}n^{\log_2 7}$
VA	$\frac{4}{7}n^{\log_2 7}$	$\frac{48}{49}n^{\log_2 7}$	$\frac{448}{343}n^{\log_2 7}$	$\frac{3840}{2401}n^{\log_2 7}$
R	$\frac{8}{7}n^{\log_2 7}+\frac{1}{3}n^{\log_2 6}$-$\frac{5}{2}n^2$	$\frac{32}{49}n^{\log_2 7}+\frac{8}{36}n^{\log_2 6}$-$\frac{5}{2}n^2$	$\frac{128}{343}n^{\log_2 7}+\frac{32}{216}n^{\log_2 6}$-$\frac{5}{2}n^2$	$\frac{512}{2401}n^{\log_2 7}+\frac{128}{1296}n^{\log_2 6}$-$\frac{5}{2}n^2$
Total	$4n^{\log_2 7}+\frac{1}{3}n^{\log_2 6}+$ $\frac{8}{5}n^{\log_2 5}$-$\frac{13}{2}n^2$	$\frac{176}{49}n^{\log_2 7}+\frac{8}{36}n^{\log_2 6}+$ $\frac{32}{25}n^{\log_2 5}$-$\frac{13}{2}n^2$	$\frac{1216}{343}n^{\log_2 7}+\frac{32}{216}n^{\log_2 6}+$ $\frac{128}{125}n^{\log_2 5}$-$\frac{13}{2}n^2$	$\frac{8960}{2401}n^{\log_2 7}+\frac{128}{1296}n^{\log_2 6}+$ $\frac{512}{625}n^{\log_2 5}$-$\frac{13}{2}n^2$

3.3 Conclusion

Observing the linearity property of R and CMF operations brings benefits on the arithmetic complexity. It reduces the arithmetic cost from $6n^{\log_2 7} - 5n^2$ to $5n^{\log_2 7} + \frac{1}{2}n^{\log_2 6} + 2n^{\log_2 5} - \frac{13}{2}n^2$ over the binary field. When we combine the use of limited recursion and block recombination on the improved Winograd's variant, we can get the least arithmetic complexity among Strassen-like algorithms for matrix multiplication as:

$$\frac{1216}{343}n^{\log_2 7} + \frac{32}{216}n^{\log_2 6} + \frac{128}{125}n^{\log_2 5} - \frac{13}{2}n^2.$$

Chapter 4

Software Implementation

For software implementation of Strassen-like algorithms, all algorithms are coded in C++ and compiled using Xcode which is an integrated development environment (IDE). A performance comparison based on timing is made amongst various algorithms implemented on two different machines for matrix dimension 2^7, 2^8 and 2^9. The improved design using the linearity property of CMF and R operations discussed in the previous chapter is implemented. Our implementation also incorporates the block recombination and limited recursion techniques. In order to simplify design in software realization and hardware simulation, all the matrices are defined over the binary field $GF(2)$ of two elements: $\{0, 1\}$, where addition and multiplication are simply logical XOR and AND operations, respectively.

The main purpose of this chapter is to investigate *relative* performance of software implementations of SA, WV and IWV. Our implementations are not intended to match timing results achieved by commercial or open-source software.

4.1 Implementation Analysis

Strassen-like algorithms use the divide-and-conquer technique to perform matrix multiplication. In theory, it is much faster than the naive method. However, in practice some features of the divide-and-conquer approach limits its own performance. Firstly, Strassen-like algorithms recursively split the input matrices until the dimension reaches 1, which requires a great amount of temporary memory to store intermediate data. Secondly, recursion is slower due to the overhead of maintaining the stack.

4.1.1 Matrix Construction

Assume that $C = A \times B$, where A, B, C are each an $n \times n$ matrix. The input matrix A is assumed to be divided into four sub-matrices with dimension $\frac{n}{2}$: A_{11}, A_{12}, A_{21} and A_{22}, and input matrix B is also divided into four sub-matrices with dimension $\frac{n}{2}$: B_{11}, B_{12}, B_{21} and B_{22}. We define a *bernoulli_distribution* class $u(e)$ to produce *bool* values, where e is a $default\_random\_engine$ class that generates pseudo-random numbers. These *bool* values are generated as entries of the input matrices, each of which is 8 bits.

4.1.2 Programming Language

C++, a kind of object-oriented programming (OOP) language, has been used to develop the program since OOP provides code reusability. Inheritance, a feature of OOP, makes it possible for the subclasses of data object sharing some characteristics from the main class. It ensures more accurate coding and thorough analysis on data. It is easy to maintain and modify existing code by removing and creating new objects.

Enterprises are likely to use C++ instead of Java to develop applications that heavily depend on speed and resource usage. The most important reason is that C++ code runs faster, since the first job of Java during run-time is to be interpreted, while C++ is to be compiled to be binaries and implemented instantly. C++ succeeds to achieve a tradeoff between programming abstractions and implementation details.

4.1.3 Requirement Analysis

The main objective is to carry out matrix multiplication using Strassen-like algorithms. Execution time is to compare the algorithms. For example, the procedure of IWV with limited recursion value $m = 2$ is as follows:

- Generate and divide each of input matrices into four block sub-matrices.
- Use naive block matrix multiplication to generate four sub-matrix multiply-and-add.
- For each sub-matrix multiply-and-add, use block recombination to call blocks CMF, CM, VA, R to generate final products.

4.2 Strassen's Algorithm

This section discusses the pseudocode of realizing Strassen's matrix multiplication algorithm based on block decomposition. The first part provides the pseudocode for Component Matrix Formation of input matrix A and B. The second part provides the pseudocode for block Reconstruction.

4.2.1 Pseudocode of SA's Component Matrix Formation

Algorithms 1 and 2 are component matrix formation methods to linearly combine the sub-matrices, call itself on half-sized matrices, and produce result vector $K1$ and $K2$ with length of $n^{\log_2 7}$. Clearly, we can see that in each CMF block, it uses 5 component additions/subtractions to generate U, T, Q, R and V. The whole recursion is unrolled into seven sub-recursions. Each sub-recursion returns a vector with length of $7^{\log_2 n-1}$. These returned vectors are formed to be the output vector.

Algorithm 1 SA's Component Matrix Formation of Input Matrix A

Require: Matrix Dimension N, Matrix A

Ensure: Output Vector $K1$

function $CMF_A^{SA}(N, A)$
  if $N = 1$ **then**
    return A
  else
    $n \leftarrow \frac{N}{2}$
    $U \leftarrow A_{11} + A_{22}$, $T \leftarrow A_{21} + A_{22}$, $Q \leftarrow A_{11} + A_{12}$, $R \leftarrow A_{21} - A_{11}$, $V \leftarrow A_{12} - A_{22}$
    $V_1 \leftarrow CMF_A^{SA}(n, U)$, $V_2 \leftarrow CMF_A^{SA}(n, T)$
    $V_3 \leftarrow CMF_A^{SA}(n, A_{11})$, $V_4 \leftarrow CMF_A^{SA}(n, A_{22})$
    $V_5 \leftarrow CMF_A^{SA}(n, Q)$, $V_6 \leftarrow CMF_A^{SA}(n, R)$, $V_7 \leftarrow CMF_A^{SA}(n, V)$
    $K1 \leftarrow (V_1, V_2, V_3, V_4, V_5, V_6, V_7)$
  end if
  Free U, T, Q, R, V
  return $K1$
end function

Algorithm 2 SA's Component Matrix Formation of Input Matrix B

Require: Matrix Dimension N, Matrix B

Ensure: Output Vector $K2$

function $CMF_B^{SA}(N, B)$
  if $N = 1$ **then**
    return B
  else
    $n \leftarrow \frac{N}{2}$
    $U \leftarrow B_{11} + B_{22}$, $T \leftarrow B_{12} - B_{22}$, $Q \leftarrow B_{21} - B_{11}$
    $R \leftarrow B_{11} + B_{12}$, $V \leftarrow B_{21} + B_{22}$
    $V_1 \leftarrow CMF_B^{SA}(n, U)$, $V_2 \leftarrow CMF_B^{SA}(n, B_{11})$
    $V_3 \leftarrow CMF_B^{SA}(n, T)$, $V_4 \leftarrow CMF_B^{SA}(n, Q)$
    $V_5 \leftarrow CMF_B^{SA}(n, B_{22})$, $V_6 \leftarrow CMF_B^{SA}(n, R)$
    $V_7 \leftarrow CMF_B^{SA}(n, V)$
    $K2 \leftarrow (V_1, V_2, V_3, V_4, V_5, V_6, V_7)$
  end if
  Free U, T, Q, R, V
  return $K2$
end function

4.2.2 Pseudocode of SA's Reconstruction

Assume that $K = (K_1, K_2, K_3, K_4, K_5, K_6, K_7)$. Algorithm 3, SA's Reconstruction shows the procedure how to linearly reconstruct the input vector K, which is the vector product of $K1$ and $K2$, into an output matrix C with size of $n \times n$. It needs 3, 1, 1 and 3 component additions/subtractions, respectively, to compute U, T, Q and R. The total cost is 8. Clearly, $U = C_{11}$, $T = C_{12}$, $Q = C_{21}$ and $R = C_{22}$.

Algorithm 3 SA's Reconstruction

Require: Vector Length N, Vector K

Ensure: Output Matrix C

function $R^{SA}(N, K)$
  if $N = 1$ **then**
    return K
  else
    $n \leftarrow \frac{N}{7}$
    $V_i \leftarrow R^{SA}(n, K_i), i = 1, \cdots, 7$
    $U \leftarrow V_1 + V_4 - V_5 + V_7$
    $T \leftarrow V_3 + V_5$
    $Q \leftarrow V_2 + V_4$
    $R \leftarrow V_1 - V_2 + V_3 + V_6$
    $C \leftarrow (U, T, Q, R)$
  end if
  Free $V_1, V_2, V_3, V_4, V_5, V_6, V_7$
  return C
end function

4.3 Winograd's Variant

4.3.1 Pseudocode of WV's Component Matrix Formation

WV's Component Matrix Formation, Algorithms 4 or 5, consumes 1 less computations on sub-matrices. However, it results in the dependency relationship between U and Q, R and T. In function $CMF_A^{WV}(N,\ A)$, Q is the subtraction between U and A_{22}, and R is the addition of Q and A_{12}.

Algorithm 4 WV's Component Matrix Formation of Input Matrix A

Require: Matrix Dimension N, Matrix A

Ensure: Output Vector $K1$

function $CMF_A^{WV}(N, A)$
  if $N = 1$ **then**
    return A
  else
    $n \leftarrow \frac{N}{2}$
    $U \leftarrow A_{11} - A_{21}$, $T \leftarrow A_{21} + A_{22}$, $Q \leftarrow U - A_{22}$, $R \leftarrow Q + A_{12}$
    $V_1 \leftarrow CMF_A^{WV}(n, A_{11})$, $V_2 \leftarrow CMF_A^{WV}(n, A_{12})$
    $V_3 \leftarrow CMF_A^{WV}(n, A_{22})$, $V_4 \leftarrow CMF_A^{WV}(n, U)$
    $V_5 \leftarrow CMF_A^{WV}(n, T)$, $V_6 \leftarrow CMF_A^{WV}(n, R)$, $V_7 \leftarrow CMF_A^{WV}(n, Q)$
    $K1 \leftarrow (V_1, V_2, V_3, V_4, V_5, V_6, V_7)$
  end if
  Free U, T, Q, R
  return $K1$
end function

Algorithm 5 WV's Component Matrix Formation of Input Matrix B

Require: Matrix Dimension N, Matrix B

Ensure: Output Vector $K2$

1: **function** $CMF_B^{WV}(N, B)$
2: **if** $N = 1$ **then**
3: **return** B
4: **else**
5: $n \leftarrow \frac{N}{2}$
6: $U \leftarrow B_{22} - B_{12}$, $T \leftarrow B_{12} - B_{11}$
7: $Q \leftarrow T + B_{22}$, $R \leftarrow Q - B_{21}$
8: $V_1 \leftarrow CMF_B^{WV}(n, B_{11})$, $V_2 \leftarrow CMF_B^{WV}(n, B_{21})$
9: $V_3 \leftarrow CMF_B^{WV}(n, R)$, $V_4 \leftarrow CMF_B^{WV}(n, U)$
10: $V_5 \leftarrow CMF_B^{WV}(n, T)$, $V_6 \leftarrow CMF_B^{WV}(n, B_{22})$
11: $V_7 \leftarrow CMF_B^{WV}(n, Q)$
12: $K2 \leftarrow (V_1, V_2, V_3, V_4, V_5, V_6, V_7)$
13: **end if**
14: Free U, T, Q, R
15: **return** $K2$
16: **end function**

4.3.2 Pseudocode of WV's Reconstruction

Assume that K is divided into 7 sub-vectors: K_1, K_2, K_3, K_4, K_5, K_6 and K_7. Algorithm 6, WV's Reconstruction, is unrolled into 7 recursive subroutines. The algorithm then performs 7 linear component additions/subtractions based on those sub-matrix products. It needs 1, 3, 2 and 1 component additions/subtractions, respectively, to compute U, T, Q and R. The total number of operations is 7, which is 1 less than SA's Reconstruction.

Algorithm 6 WV's Reconstruction

Require: Vector Length N, Vector K

Ensure: Output Matrix C

function $R^{WV}(N, K)$
  if $N = 1$ **then**
    return K
  else
    $V_i \leftarrow R^{WV}(\frac{N}{7}, K_i), i = 1, \cdots, 7$
    $U \leftarrow V_1 + V_2$
    $Q \leftarrow \underbrace{V_1 - V_7}_{S_1} - V_3 + V_4$
    $T \leftarrow \underbrace{S_1 + V_5}_{S_2} + V_6$
    $R \leftarrow S_2 + V_4$
    $C \leftarrow (U, T, Q, R)$
  end if
  Free $V_1, V_2, V_3, V_4, V_5, V_6, V_7$
  return C
end function

4.4 Improved Winograd's Variant

4.4.1 Pseudocode of IWV's Component Matrix Formation

For IWV's CMF, as shown in Algorithms 7 and 8, it is only unrolled into 5 sub-recursions, which is 2 less than the original WV's CMF. It significantly improves this block's performance. Furthermore, this block costs 2 less computations on sub-matrices and 2 more computations on produced vectors to obtain V_6 and V_7.

Algorithm 7 Improved WV's Component Matrix Formation of Input Matrix A

Require: Matrix Dimension N, Matrix A

Ensure: Output Vector $K1$

function $CMF_A^{IWV}(N, A)$
  if $N = 1$ **then**
    return A
  else
    $n \leftarrow \frac{N}{2}$
    $U \leftarrow A_{11} - A_{21}$, $T \leftarrow A_{21} + A_{22}$
    $V_1 \leftarrow CMF_A^{IWV}(n, A_{11})$, $V_2 \leftarrow CMF_A^{IWV}(n, A_{12})$
    $V_3 \leftarrow CMF_A^{IWV}(n, A_{22})$, $V_4 \leftarrow CMF_A^{IWV}(n, U)$, $V_5 \leftarrow CMF_A^{IWV}(n, T)$
    $V_7 \leftarrow V_4 - V_3$, $V_6 \leftarrow V_7 + V_2$
    $K1 \leftarrow (V_1, V_2, V_3, V_4, V_5, V_6, V_7)$
  end if
  Free U, T, Q, R
  return $K1$
end function

Algorithm 8 Improved WV's Component Matrix Formation of Input Matrix B

Require: Matrix Dimension N, Matrix B

Ensure: Output Vector $K2$

1: **function** $CMF_B^{IWV}(N, B)$
2: **if** $N = 1$ **then**
3: **return** B
4: **else**
5: $n \leftarrow \frac{N}{2}$
6: $U \leftarrow B_{22} - B_{12}$, $T \leftarrow B_{12} - B_{11}$
7: $V_1 \leftarrow CMF_B^{IWV}(n, B_{11})$, $V_2 \leftarrow CMF_B^{IWV}(n, B_{21})$
8: $V_4 \leftarrow CMF_B^{IWV}(n, U)$, $V_5 \leftarrow CMF_B^{IWV}(n, T)$
9: $V_6 \leftarrow CMF_B^{IWV}(n, B_{22})$
10: $V_7 \leftarrow V_6 - V_5$
11: $V_3 \leftarrow V_7 - V_2$
12: $K2 \leftarrow (V_1, V_2, V_3, V_4, V_5, V_6, V_7)$
13: **end if**
14: Free U, T, Q, R, V
15: **return** $K2$
16: **end function**

4.4.2 Pseudocode of IWV's Reconstruction

The improved Reconstruction block calls itself fewer times and use less matrix additions/subtractions on $\frac{n}{2} \times \frac{n}{2}$ matrices. It costs 2 additions/subtractions based on sub-vectors to produce U and T . Most importantly, this block is only unrolled into 6 recursive subroutines to get R_1, R_2, R_3, R_5, R_7, and R_8. Furthermore, it needs 5 extra additions/subtractions on sub-matrices to compute R_4, R_6, R_9 and R_{10}, as shown in Algorithm 9.

Algorithm 9 Improved WV's Reconstruction

Require: Vector Length N, Vector K

Ensure: Output Matrix C

function $R^{IWV}(N, K)$
  if $N = 1$ **then**
    return K
  else
    $n \leftarrow \frac{N}{7}$
    $U \leftarrow K_1 + K_2$, $T \leftarrow K_1 - K_7$
    $R_1 \leftarrow R(n, U)$, $R_2 \leftarrow R(n, T)$, $R_3 \leftarrow R(n, K_5)$
    $R_4 \leftarrow R_2 + R_3$, $R_5 \leftarrow R(n, K_6)$, $R_6 \leftarrow R_4 + R_5$
    $R_7 \leftarrow R(n, K_3)$, $R_8 \leftarrow R(n, K_4)$
    $R_9 \leftarrow R_8 - R_7 + R_2$, $R_{10} \leftarrow R_4 + R_8$
    $C \leftarrow (R_1, R_6, R_9, R_{10})$
  end if
  return C
end function

4.5 Performance Analysis

In this section, we present the implementation performance of naive matrix multiplication method and Strssen-like algorithms based on block decomposition. The implementation is for matrix size $2^i \times 2^i$, for $i = 7, 8, 9$. All timing values in milliseconds are obtained by averaging results from 100 experiments for each of three matrix dimensions.

Table 4.1 lists the time consumption of implementing different matrix multiplication methods on Macbook Pro with Intel Core i5 processor and 16 GB memory. We should note that IWV_2, IWV_4 and IWV_8 means that IWV is block recombined with different limited recursion values for $m = 2, 4, 8$, respectively.

Table 4.1: Timing Cost of Implementing Matrix Multiplication Methods on Macbook Pro

Matrix Size	SA	WV	IWV	IWV_2	IWV_4	IWV_8	Naive Method
128×128	1009	859	324	224	155	108	8
256×256	7118	5988	1895	1293	944	672	69
512×512	50043	42226	11314	7684	5282	3735	570

From Table 4.1, we can conclude that WV has better performance than SA, and the optimization techniques greatly reduce WV's time consumption. For example, when matrix dimension $n = 256$, SA takes 7188 ms, WV takes 5988 ms, IWV takes 1895 ms. With the limited recursion value of m, the timing for IWV improves to 1293 ms, 944 ms and 672 ms for $m = 2$, 4 and 8, respectively. As shown in Figure 4.1, with matrix dimension increasing from 128 to 256 to 512, the average growth rate is approximately 7.04, 7.01, 5.91, 5.86, 5.95, 5.89, 8.45, respectively, for SA, WV, IWV, IWV_2, IWV_4, IWV_8 and the naive method. It concludes that Strassen-like matrix multiplication algorithms

have relatively lower timing growth than the naive method. Therefore, we expect that at this growth rate, IWV_8 outperforms the naive method before the dimension reaches 2^{15}.

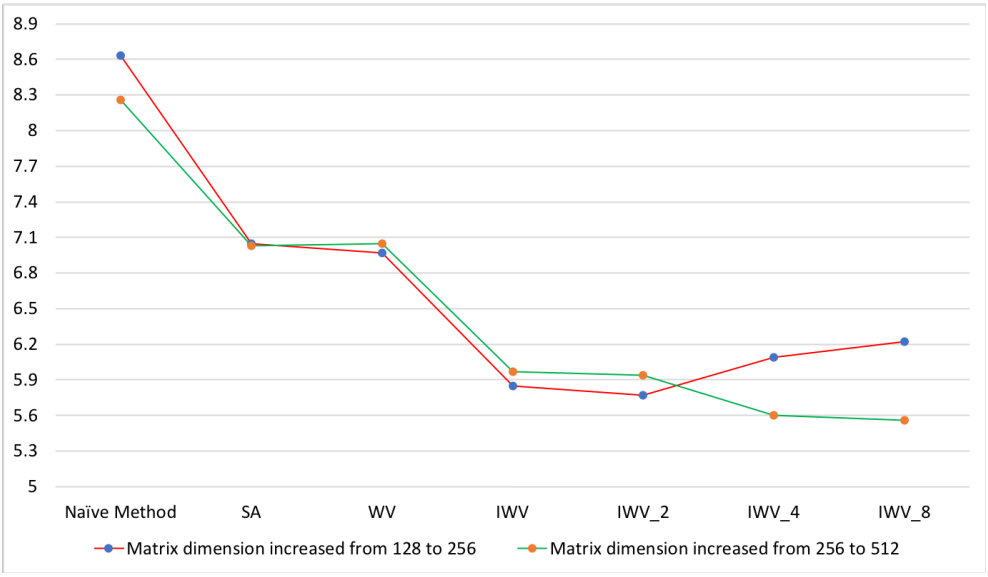

Figure 4.1: Growth Rate of Multiple Matrix Multiplication Methods' Timing Cost

Table 4.2 shows the timing performance of impelementing various matrix multiplication methods on Macbook Air with Intel Core i5 processor and 8 GB memory.

Table 4.2: Timing Cost of Implementing Matrix Multiplication Methods on Macbook Air

Matrix Size	SA	WV	IWV	IWV_2	IWV_4	IWV_8	Naive Method
128×128	1352	1107	485	285	204	141	8
256×256	9382	7826	2486	1700	1154	845	76
512×512	68884	58289	14718	9945	6840	4826	667

From Table 4.1 and 4.2, we can conclude that machines with more computing resources can considerably reduce the methods' running time.

Chapter 5

Hardware Simulation

For hardware simulation, we use Altera Quartus II Prime 18.0, which is a programmable logic device design software by Intel, to compile design, perform timing analysis, simulate, and synthesis. In order to verify the correctness of computation results, we write test-bench applied in ModelSim (Intel FPGA Starter Edition 10.5b) to check register status and output data. All these algorithms are programmed in Verilog [1]. The fitted device is Cyclone IV E. All Strassen-like algorithms are implemented for matrix size $2^i \times 2^i$, for $i = 1,\ 2, \cdots,\ 5$. For IWV implementation, we also consider the application of block re-combination and different limited recursion values.

5.1 System Design

5.1.1 System's Overall Architecture

The basic idea of implementing Strassen-like algorithms is based on block decomposition. When the size of the matrix under consideration becomes big, a matrix cannot be sent into the central unit for computation within a single clock cycle. Thus, registers for temporarily storing these input data are necessary.

Take 16×16 matrix multiplication as example. It needs three registers of size 256 bits each, one $CMF^{A}_{16\times16}$, one $CMF^{B}_{16\times16}$, one $CM_{16\times16}$, and one $R_{16\times16}$. Figure 5.1 shows the system's overall structure.

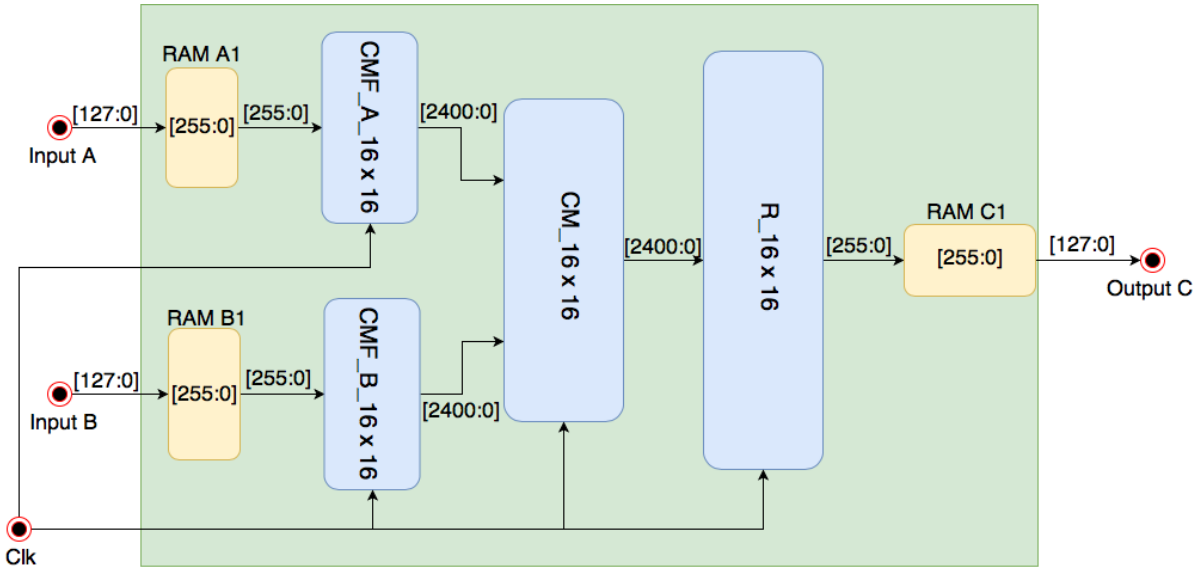

Figure 5.1: System's Overall Architecture of Matrix Multiplication Algorithms Based on Block Decomposition

5.1.2 Module Instantiation

Verilog module, like functions in C++, is a piece of code that can be reused within a program. It provides the template where we can build actual objects. Module can be invoked from other modules, which is denoted as module instantiation. When we instantiate a module, we need to specify the connections to ports of the module. The simplest way to instantiate a module in top module is to wire the ordered ports up within a named instance. We need to keep the ports mapping.

Module instantiation enables hierarchical design in Verilog. Hierarchical design usually includes a top level module and several lower level modules. It enlightens the realization design of recursive algorithms. For example, $CMF_A^{4\times4}$ is a module realizing component matrix formation of input matrix A with size 4×4. In IWV, it needs to instantiate five $CMF_A^{2\times2}$ with different names, as shown in Figure 5.2.

```
Block_Decomposition_WV_CMF_X_2by2 MCFX1(
   .clk(clk),
   .A(A1[0][0]),
   .Res(C1)
);
Block_Decomposition_WV_CMF_X_2by2 MCFX2(
   .clk(clk),
   .A(A1[0][1]),
   .Res(C2)
);
Block_Decomposition_WV_CMF_X_2by2 MCFX3(
   .clk(clk),
   .A(A1[1][1]),
   .Res(C3)
);
Block_Decomposition_WV_CMF_X_2by2 MCFX4(
   .clk(clk),
   .A(R1),
   .Res(C4)
);
Block_Decomposition_WV_CMF_X_2by2 MCFX5(
   .clk(clk),
   .A(R2),
   .Res(C5)
);
```

Figure 5.2: Lower Level Module Instantiation of $CMF_A^{4\times4}$

5.2 System Parameter Setting

5.2.1 Cyclone IV devices

Altera's new Cyclone IV FPGA device family extends the feature of Cyclone FPGA series. Especially, Cyclone IV E devices are best used for low-cost, small-form-factor applications widely applied in wireless, wireline, broadcast, industrial, and communication industries. The voltages that Cyclone IV E device family offers are of 1.0V and 2.0V. The device family usually provides 6K to 115K logic blocks, 94 to 535 user I/Os, and up to 4 Mb of embedded memory, which is considered as low-cost and low-power FPGA fabric. Table 5.1 lists the device resources limit of part of Cyclone IV E device family [2].

Table 5.1: Resource of Cyclone IV E Device Family

Resources	EP4CE40	EP4CE55	EP4CE75	EP4CE115
Logic Elements	39600	55856	75408	114480
Embedded Memory(Kbits)	1134	2340	2745	3888
Embedded 18×18 multipliers	116	154	200	266
General-purpose PLLs	4	4	4	4
Global Clock Network	20	20	20	20
Maximum User I/O	532	374	426	528

5.2.2 Logic Elements

Logic elements (LE) are considered as the smallest units of logic in the Cyclone IV device architecture. LEs are put packed firmly and offer high-performance specifications

with reasonable logic usage. As shown in Figure 5.3, each LE should include:

- A four-input look-up table(LUT). LUT is used to implement any functions with four variables.
- Programmable Register.
- Carry Chain Connection and Register Chain Connection
- Register Packing Support

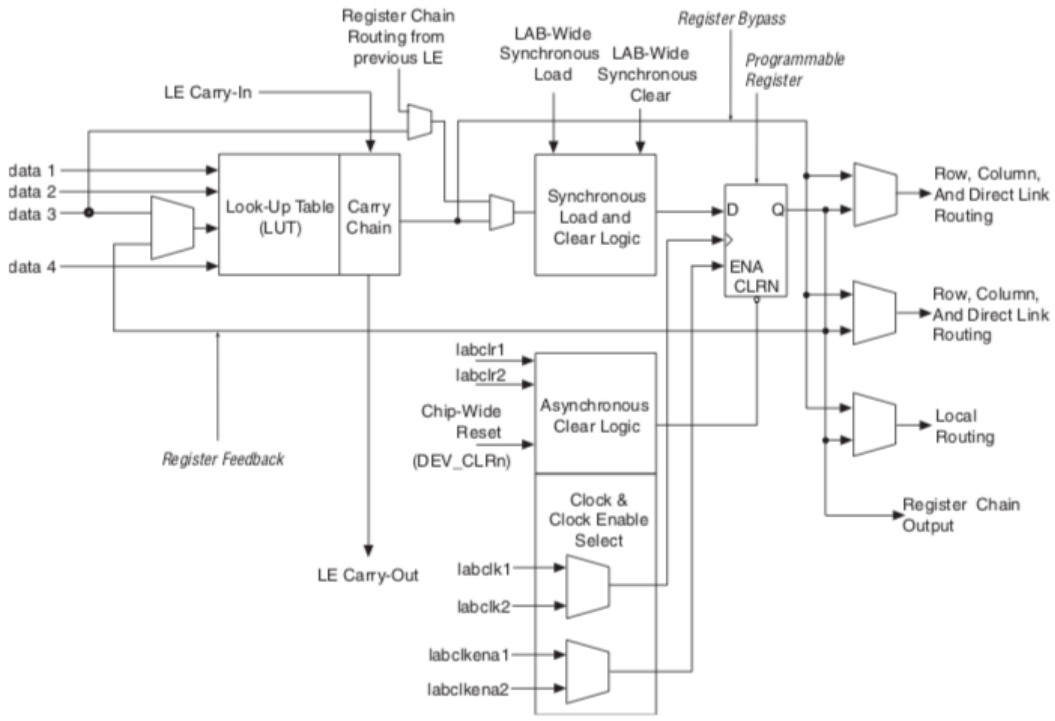

Figure 5.3: Cyclone IV Device LEs

5.2.3 Input/Output Format

For 16×16 and 32×32 modules, the width of data input and output ports is customized to be 128 bits for better timing performance. Take 32×32 matrix multiplication module as an example, for input matrix A, the total 1024 bits from $A[0][0]$ to $A[31][31]$ are packed as a vector with length of 1024 bits. Therefore, it totally needs 8 clock cycles to subsequently store all input data into register. It should be noted that in the 8 clock cycles, input matrix B is also successfully transmitted to a register at the same time. Thus, the input is sent over 8 cycles. When all arithmetic operations are done, the product matrix is ready to be sent out. The product is in the form 1-dimension vector with length of 1024 bits. The timing cost for output is also 8 cycles [23].

5.3 Sytem Circuit

For the realization of recursions, we adopt nested module instantiation methodology. We first design modules for smaller matrices or vectors, and then instantiate them in the top module. For example, the CMF module of SA for $n = 4$ needs to instantiate 7 CMF modules of SA for $n = 2$. In order to specifically explain the hardware implementation, we illustrate the circuit with diagrams respectively for each method. It is easy to find out how many logic component operations are used in the circuit.

5.3.1 Block Decomposition Circuit of SA

In Figure 5.4, block decomposition circuit of SA, the SA is divided into 3 clusters of blocks: $CMF \Longrightarrow CM \Longrightarrow R$. It instantiates 14 CMFs, 7 CMs, and 7 Rs. In this case, it totally takes 10 XORs to compute the input of CMFs and 8 XORs to reconstruct these vectors R_i's ($i = 1, \cdots, 7$). The cluster of CMs require 7 AND gates, since each CM is a bit level multiplication over $GF(2)$.

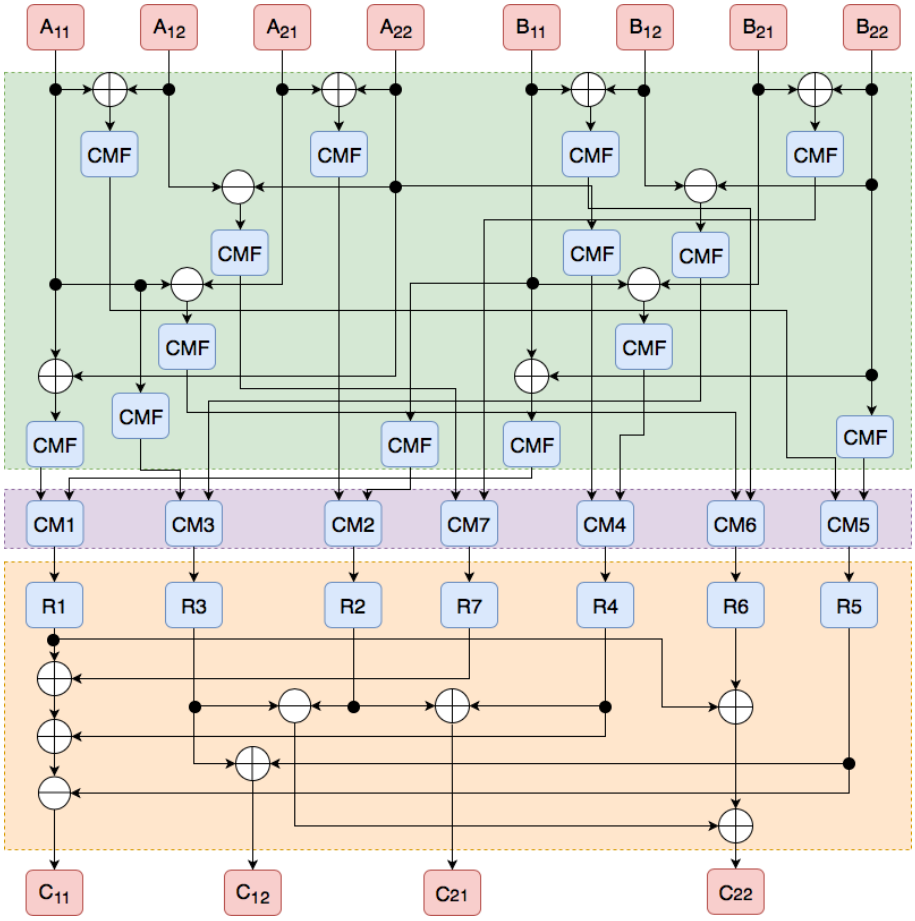

Figure 5.4: Block Decomposition Circuit of SA for $n = 2$

5.3.2 Block Decomposition Circuit of WV

In Figure 5.5, block decomposition circuit of WV, there are 2 less XORs to compute the input of CMFs and 1 less XOR to reconstruct these vectors R_i's $(i = 1, \cdots, 7)$ than SA. The number of AND gates are the same with that in SA.

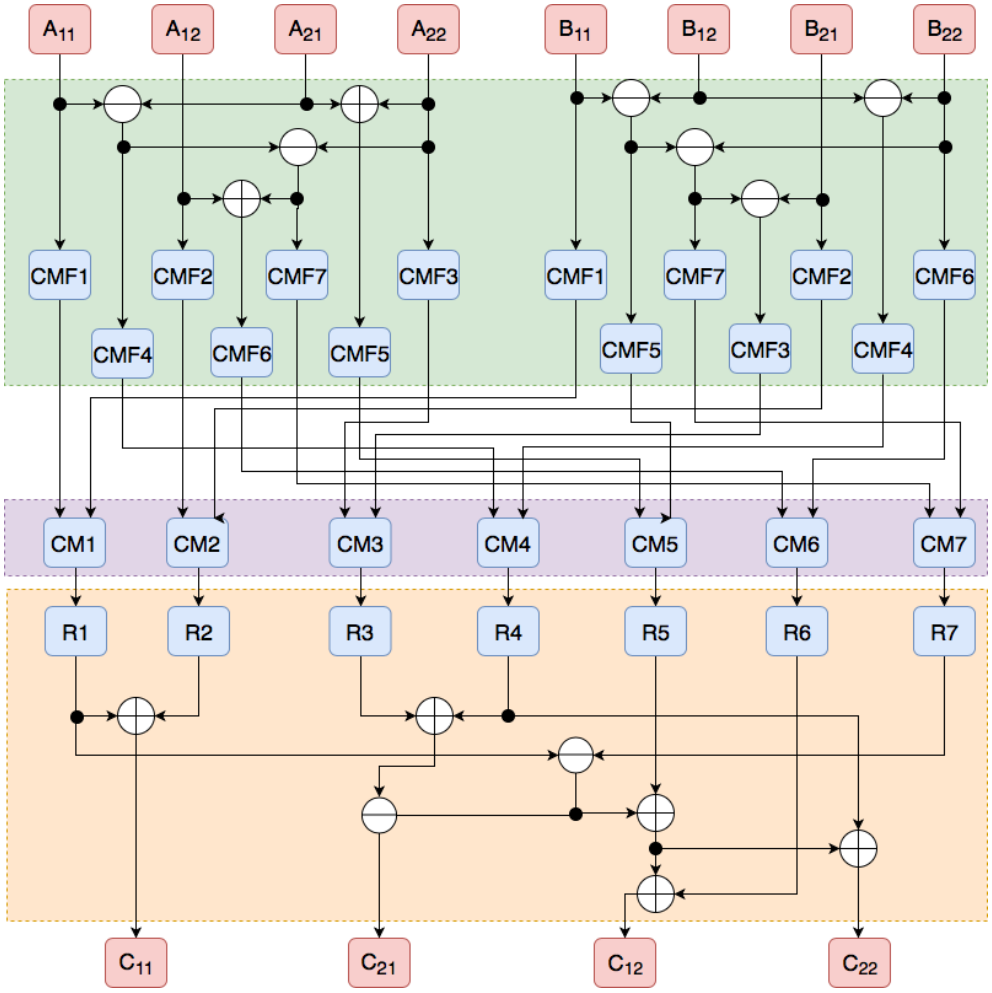

Figure 5.5: Block Decomposition Circuit of WV for $n = 2$

5.3.3 Block Decomposition Circuit of IWV

In Figure 5.6, block decomposition circuit of IWV, it only instantiates 10 CMFs, 7 CMs, and 6 Rs. Furthermore, it totally takes 4 XORs to compute the input of CMFs, 4 XORs to compute the input of CMs, 2 XORs to compute the output of CMs, and 5 XORs to reconstruct these vectors R_i's ($i = 1, \cdots, 7$). A total of 7 AND gates are needed in CMs.

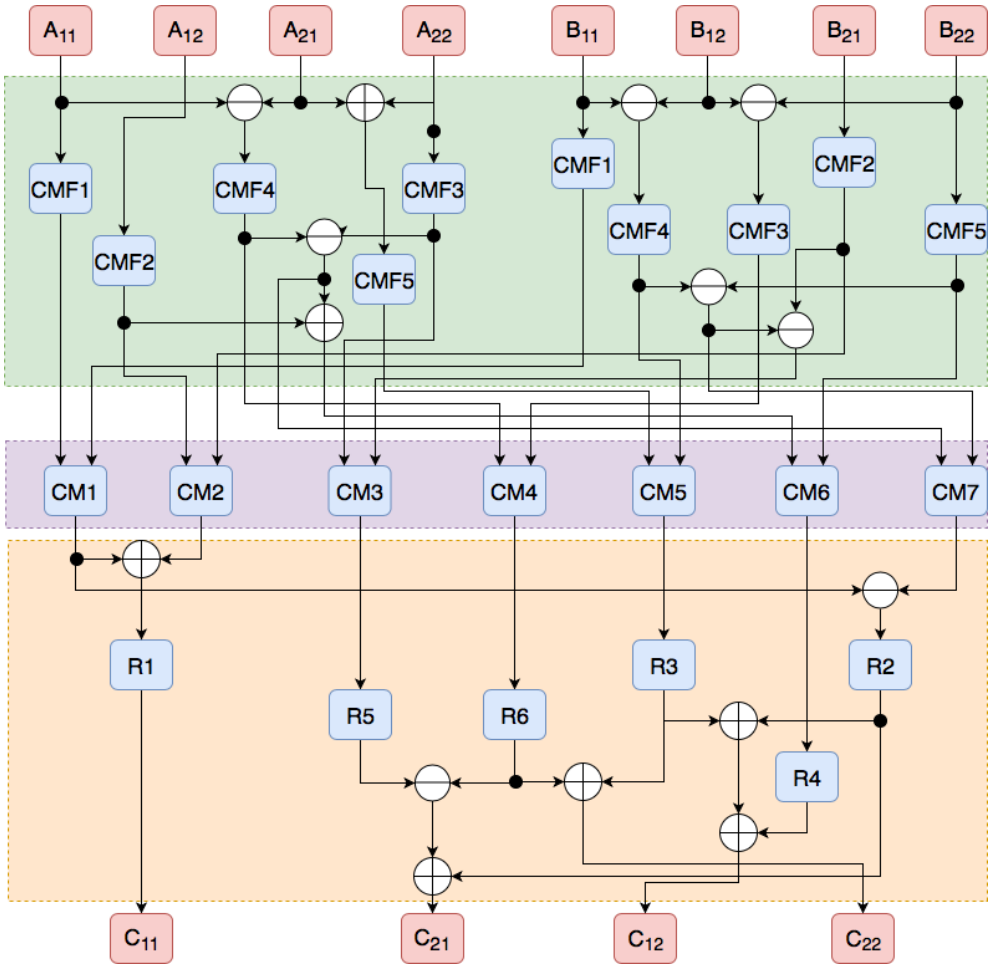

Figure 5.6: Block Decomposition Circuit of Improved WV for $n = 2$

5.4 Performance Evaluation

5.4.1 Performance Comparison

Table 5.2 summarizes the details of performance metrics of SA, WV, and IWV based on block decomposition. IWV is based on observing the linearity property of WV's CMF and R.

For physical resource utilization, we can see that the usage of logic elements in WV is reduced around 7% than SA for matrix dimension ranging from 2^2 to 2^5. On the other hand, IWV tends to use more logic elements than WV, and for the matrix dimensions used in our hardware simulation the differences are 51%, 51%, 54% and 58%, respectively. However, the memory bits are reduced by 59.2%, 74.1%, 62.5% and 69.4%, respectively.

In terms of clock rate, maximum frequency is higher by 5.6%, 28.5% and 76.7% compared to those of WV for matrix sizes 4×4, 8×8, 16×16. In Table 5.2, we also list the number of clock cycles needed by the three methods. By dividing these clock cycle counts by the maximum frequency, we get computation time. From the data in the table, we can conclude that the computation time required by IWV is less than that by WV. We also note that with the increase of matrix dimension from 2 to 16, the rate of reduction in computation time by IWV tends to be larger.

Table 5.2: Performance Metrics of SA, WV and IWV Based on Block Decomposition

Metrics		2×2	4×4	8×8	16×16	32×32
SA	Logic Elements	58	291	2038	14320	106691
	Pins (total)	13	49	193	385	385
	Total Memory Bits	0	441	2744	16807	168070
	Clock Cycles	7	12	15	21	*
	Max Frequency (MHz)	675.22	358.68	274.5	113.9	*
	Computation Time (ns)	10.37	33.46	54.64	184.37	*
WV	Logic Elements	58	269	1884	13242	99314
	Pins (total)	13	49	193	385	385
	Total Memory Bits	0	441	2744	16807	168070
	Clock Cycles	7	12	15	21	*
	Max Frequency (MHz)	654.88	339.67	260.62	115.04	*
	Computation Time (ns)	10.69	35.33	57.56	182.55	*
IWV	Logic Elements	58	406	2839	20426	157293
	Pins (total)	13	49	193	385	385
	Total Memory Bits	0	180	712	6308	51428
	Clock Cycles	7	12	15	21	*
	Max Frequency (MHz)	654.88	358.68	334.78	203.29	*
	Computation Time (ns)	10.69	33.46	44.81	103.30	*

5.4.2 Performance Optimization

In order to further improve the resource utilization, we exploit the use of block recombination and limited recursion. For accurate analysis, we only employ these methods

on matrix size of 16×16 and 32×32. Table 5.3 summarizes the details of performance metrics of IWV based on block decomposition with different limited recursion values m when matrix size is 16×16. From the table, we see that the size of initially formed matrix has a big influence on the overall performance. The logic element usage is reduced from 14826 to 10322 to 8686. The total memory bits are reduced from 2416 to 1644 to 1116. The number of clock cycles is reduced from 21 to 15 to 11.

Table 5.3: Performance Metrics of IWV Based on Block Decomposition for 16×16 Matrix

Metrics	Matrix Size 16×16		
	$m = 2$	$m = 4$	$m = 8$
Logic Elements	14826	10322	8686
Total Memory Bits	2416	1644	1116
Clock Cycles	21	15	11
Max Frequency (MHz)	99.26	134.37	103.56
Computation Time (ns)	211.16	111.63	106.22

Table 5.4 provides the details of performance metrics of IWV based on block decomposition with different limited recursion values when matrix size is 32×32. The logic element usage is reduced from 105873 to 70152 to 59639. The total amount of memory bits is reduced from 18816 to 9644 to 6912. The number of clock cycles is reduced from 35 to 30 to 27. Maximum clock frequency increases from 39.73 MHz to 55.32 MHz to 72.24 MHz. Computation time is reduced from 880.95 ns to 542.30 ns to 373.35 ns.

Table 5.4: Performance Metrics of IWV Based on Block Decomposition for 32×32 Matrix

Metrics	Matrix Size 32×32		
	$m = 2$	$m = 4$	$m = 8$
Logic Elements	105873	70152	59639
Total Memory Bits	18816	9664	6912
Clock Cycles	35	30	27
Max Frequency (MHz)	39.73	55.32	72.24
Computation Time (ns)	880.95	542.30	373.35

Chapter 6

Concluding Remarks

6.1 Summary

In this thesis, we have considered Strassen-like algorithms for matrix multiplications, specifically Strassen's algorithm and its variant by Winograd. These two algorithms extend the idea of block matrix multiplication, divide-and-conquer, to reduce the algorithm complexity from $O(n^3)$ to $O(n^{2.81})$.

In order to make analysis of these two fast matrix multiplication methods, we have reviewed Cenk and Hasan's idea to divide the whole algorithm into four blocks: CMF_A, CMF_B, CM, and R. Each block is considered as a recursive function. Several examples were instantiated to verify the correctness of these blocks. We have also investigated three methodology: linearity property of CMF and R, block recombination, and limited recursion, to improve Winograd's variant for better performance. Complexities of improved methods have also been listed for comparisons.

Software implementation and hardware simulation have both been performed to

support the theoretical analysis. For software implementation, we adopted C++ as programming language and realized matrix multiplications for dimensions 128, 256 and 512. In our software realizations, WV is about 15% faster than SA. Observing the linearity property of CMF and R operations improves WV by 66%. The combined use of block recombination and limited recursion reduces the timing by up to 67%.

For hardware simulation, we have used Verilog to realize it. Due to the limit of hardware resources, we have restricted our matrix size to $2^i \times 2^i$, for $i = 1, 2, \cdots, 5$. From the synthesis results, we have concluded that WV consumed about 7.5% less logic elements than SA. A number of optimization techniques have been employed to considerably reduce the number of logic elements, the total amount of storage (i.e., memory bits) and the running time.

Above all, we are able to make conclusions that WV is better than SA both in software implementation and hardware simulation. The optimization methods introduced by Cenk and Hasan have been tested to considerably improve WV's performance with regard to execution time and resource utilization.

6.2 Future Work

As can be seen from the previous chapters, Strassen-like algorithms have asymptotically lower complexities than the naive method. But in real experiment, Strassen-like algorithms are not efficient enough when they are applied on smaller size matrices. They incur a huge amount of extra time to realize the recursion unrolled to a certain level. For larger matrices, they require massive memory storage and hence it is worth to try implementing them on a computer system with a huge amount of high-speed storage [24].

For hardware simulation analysis, it is easy to exceed FPGA device resource limitations if we directly unroll all recursions and compute them. Therefore, it would be interesting to try implementing Strassen-like algorithms using Application Specific Integrated Circuits (ASIC).

References

[1] Ambika, Anuradha S. High Speed UART Design Using Verilog. International Journal of Advanced Research in Computer and Communication Engineering, Pages 140-142, Vol.5, Issue 2, February 2016.

[2] Altera-corporation. Cyclone IV Device Handbook. Volume 1, April 2014 available at https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-iv/cyclone4-handbook.pdf?language=en_US. Accessed 1 November 2018.

[3] Andris Ambainis, Yuval Filmus, François Le Gall. Fast Matrix Multiplication: Limitations of the Laser Method, arXiv:1411.5414. Technical Report, 2014 available at https://arxiv.org/pdf/1411.5414.pdf. Accessed October 12, 2018.

[4] Howard Anton, Chris Rorres. Elementary Linear Algebra, 11th Edition. John Wiley & Sons. November 2013.

[5] Jeff A. Bilmes, Krste Asanovic, Chee-Whye Chin, James Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. ICS '97 Proceedings of the 11th International Conference on Supercomputing, Pages 340-347, July 1997.

[6] Brice Boyer, Jean-Guillaume Dumas, Clément Pernet, Wei Zhou. Memory Efficient Scheduling of Strassen-Winograd's Matrix Multiplication Algorithm. International Symposium on Symbolic and Algebraic Computation, Pages 55-62, July 2009.

[7] Murat Cenk, M. Anwar Hasan. On the arithmetic complexity of Strassen-like matrix multiplications. Journal of Symbolic Computation, Pages 484-501, Volume: 80, Part 2, May-June 2017.

[8] Davis Cherney, Tom Denton, Rohit Thomas, Andrew Waldron. Linear Algebra. First Edition. Davis California, 2013 available at https://www.math.ucdavis.edu/~linear/linear-guest.pdf. Accessed 10 October 2018.

[9] Vineet Choudhary. Introduction to Divide and Conquer (D&C) Algorithm Design Paradigm. Available at https://developerinsider.co/introduction-to-divide-and-conquer-algorithm-design-paradigm. Accessed November 20 2018.

[10] Henry Cohn, Robert Kleinberg, Balazs Szegedy, Christopher Umans. Group-theoretic Algorithms for Matrix Multiplication. Proceedings of the 46th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, Pages 379-388, 23-25 October, 2005.

[11] Don Coppersmith, Shmuel Winograd. Matrix Multiplication via Arithmetic Progressions. Journal of Symbolic Computation, Volume 9, Issue 3, Pages 251-280, March 1990.

[12] Paolo D'Alberto, Alexandru Nicolau. Adaptive Winograd's matrix multiplications. ACM Transactions on Mathematical Software, Pages 3:11-3:23, Volume: 36, Issue: 1, March 2009.

[13] Yong Dou, S. Vassiliadis, G. K. Kuzmanov, G. N. Gaydadjiev. 64-bit Floating-Point FPGA Matrix Multiplication. Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays, Pages 86-95, February 20-22, 2005.

[14] The Editors of Encyclopaedia Britannica. Matrix Mathematics. Available at https://www.britannica.com/science/matrix-mathematics. Accessed 2 November 2018.

[15] François Le Gall. Faster Algorithms for Rectangular Matrix Multiplication. Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2012), pp. 514-523, arXiv:1204.1111, doi:10.1109/FOCS.2012.80.

[16] Ian Goodfellow, Yoshua Bengio, Aaron Courville. Deep Learning. The MIT Press, Nov. 18 2016.

[17] M. Anwar Hasan, Nicolas Meloni, Ashkan Hosseinzadeh Namin, Christophe Negre. Block Recombination Approach for Subquadratic Space Complexity Binary Field Multiplication Based on Toeplitz Matrix-Vector Product. IEEE TRANSACTIONS ON COMPUTERS, Pages 151-163, Volume: 61, NO. 2, FEBRUARY 2012.

[18] Ivo Hedtke. Strassen's Matrix Multiplication Algorithm for Matrices of Arbitrary Order. Bulletin of Mathematical Analysis and Applications, Pages 269-277, Volume: 3, Issue: 2, 2011.

[19] Roger A. Horn, Charles R. Johnson. Matrix Analysis. Cambridge University Press, 1985.

[20] Ayaz Khan, Optimizing the Matrix Multiplication Using Strassen and Winograd Algorithms with Limited Recursions on Many-Core International Journal of Parallel Programming, Pages 801-830, Volume: 44, Issue: 4, August 2016.

[21] Ahmad Khayyat. Analysis-Driven Design of Parallel Floating-Point Matrix Multiplication for Implementation in Reconfigurable Logic. PhD thesis, Queen's University, Canada, 2013.

[22] Tomonori Kouya. Accelerated Multiple Precision Matrix Multiplication using Strassen's Algorithm and Winograd's Variant. JSIAM Letters, Volume: 6, October 2014.

[23] Vivek Kumar, Vinary B. Y. Kumar, Sachin B. Patkar. FPGA-based Implementation of M4RM for Matrix Multiplication over GF(2). 18th International Symposium on VLSI Design and Test, Pages 1-2, 2014.

[24] Juby Mathew, Dr. R Vijaya Kumar. Comparative Study of Strassen's Matrix Multiplication Algorithm. International Journal of Computer Science and Technology, Pages 749-754, Volume: 3, Issue: 1, January-March 2012.

[25] Khaled Matrouk, Abdullah Al-Hasanat, Haitham Alasha'ary, Ziad Al-Qadi, Hasan Al-Shalabi. Analysis of Matrix Multiplication Computational Methods. European Journal of Scientific Research, Pages 258-266, Volume: 121, No.3, 2014.

[26] V. Ya. Pan. New Fast Algorithms for Matrix Operations. SIAM J. Comput., 9(2): 321-342, 1980.

[27] V. Ya. Pan. Fast Matrix Multiplication and its Algebraic Neighbourhood. SB MATH, 208(11), Pages 1661-1704, 2017.

[28] Kaare Brandt Petersen, Michael Syskind Pedersen. The Matrix Cookbook. Version: November 15, 2012.

[29] Jean-No:el Quintin, Khalid Hasanov, Alexey Lastovetsky. Hierarchical Parallel Matrix Multiplication on Large-Scale Distributed Memory Platforms. 42nd International Conference on Parallel Processing, Pages: 754-762, October 1-4 2013.

[30] Utsab Ray, Tapan Kumar Hazra, Utpal Kumar Ray. Matrix Multiplication using Strassen's Algorithm on CPU & GPU. International Journal of Computer Sciences and Engineering, Pages 98-105, Volume: 4, Issue: 10, 2016.

[31] Radu Rugina, Martin Rinard. Recursion Unrolling for Divide and Conquer Programs. Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers, Pages: 34-48, August 10-12 2000.

[32] Sara Robinson. Toward an Optimal Algorithm for Matrix Multiplication. SIAM News, Volume 38, Number 9, November 2005.

[33] A.K. Sharma. Text Book of Matrix. Discovery Publishing House, India, 2004.

[34] Volker Strassen. Gaussian Elimination is not Optimal. Numer. Math. 13: 354-356, 1969.

[35] Andrew Stothers. On the Complexity of Matrix Multiplication. PhD thesis, University of Edinburgh, 2010.

[36] Gilbert Strang. Linear Algebra and Its Applications, Foutth Edition. Thomson Brooks/Cole, 2006.

[37] Virginia Vassilevska Williams, Multiplying Matrices Faster Than Coppersmith-Winograd. STOC '12 Proceedings of the forty-fourth annual ACM symposium on Theory of computing, Pages 887-898, 2012.

[38] S. Winograd. On Multiplication of 2×2 Matrices. Linear Algebra and Application, 4: 381-388, 1971.