A General Scalable Parallelizing of Winograd Algorithm for Matrix Multiplication on Distributed Memory Computer

Duc Kien NGUYEN, Quoc Trung HA

Abstract - **Winograd algorithm to multiply two $n \times n$ matrices reduces the asymptotic operation count from $O(n^3)$ of the traditional algorithm to $O(n^{2.8l})$. Thus for distributed memory computer, the association of Winograd algorithm and the parallel matrix multiplication algorithms always gives considerable results. The use of Winograd algorithm at inter-processor level requires us to solve more difficulty in designing but it forms the most effective algorithms. In this paper, we present a general model for this algorithm class and the solution optimized for this model.**

Index Terms - **Matrix multiplication, parallel algorithms, Winograd algorithm, recursion removal.**

I. Introduction

Matrix multiplication (MM) is one of the most fundamental operations in linear algebra and serves as the main building block in many different algorithms, including the solution of systems of linear equations, matrix inversion, evaluation of the matrix determinant and the transitive closure of a graph. In several cases the asymptotic complexities of these algorithms depend directly on the complexity of matrix multiplication - which motivates the study of possibilities to speed up matrix multiplication. Also, the inclusion of matrix multiplication in many benchmarks points at its role as a determining factor for the performance of high speed computations.

That is the reason why many scientists have worked hardly to improve the algorithm for matrix multiplication. Strassen was the first to introduce a better algorithm [3] for MM with $O(N^{\log_2 7})$ than the traditional algorithm (hereafter referred as T-algo) which need $O(N^3)$ operations. Then Winograd variant [6] of Strassen's algorithm has the same exponent but a slightly lower constant as the number of additions/subtractions is reduced from 18 to 15 (hereafter referred as W-algo). The record of complexity owed to Coppersmith and Winograd is $O(N^{2.376})$, resulted from arithmetic aggregation [2]. However, only Strassen's algorithm and the W-algo offer better performance than the T-method for matrices of practical sizes, say, less than 10^{20}[10].

There have been mainly two approaches to parallelize the W-algo. The first approach is to use the T-algo at the top level (between processors) and the W-algo at the bottom level (within one processor). The most commonly algorithms used T-method between processors include 1D-systolic [11], 2D-systolic [11], Fox (BMR) [13], Cannon [12], PUMMA [9], BiMMeR [14], SUMMA [15], DIMMA [16]. Since the W-algo is most efficient for large matrices (thanks to the great difference of complexity between the operation multiplication and the operation addition/subtraction of matrix), it is well suited to use at the top level, not the bottom level. The second approach is to use the W-algo at both the top and the bottom level. The first implementation applying this approach [7] on Intel Paragon reached performance better than T-algo. However, the W-algo in [7] requires that the number of processors used in the computation to be a power of seven. This is a severe restriction since many MIMD computers use hypercube or mesh architecture and powers of seven numbers of processors are not a natural grouping. Therefore, the [7]'s algorithm is not scalable. Moreover, it requires a large working space, with each matrix to be multiplied duplicated 3 or 4 times. For these reasons, in [17] Luo and Drake explored the possibility of other parallel algorithms with more practical potential: they introduced an algorithm which uses the W-algo at the top level and Fox algorithm at the bottom level. To continue, an improvement is introduced by [18]'s authors: algorithm SUMMA is used in the place of Fox algorithm at the bottom level.

We will study this algorithm class by using Cannon algorithm at bottom level and show that the total running time for the Winograd-Cannon algorithm decreases when the recursion level r increases. This result is also correct when we change Cannon algorithm at the bottom level by the other parallel MM algorithms. To use the Winograd algorithm at the top level, the most significant point is to determine the sub matrices after having recursively executed r time the Winograd formula (these sub matrices are corresponding to the nodes of level r in the execution tree of Winograd algorithm) and then to find the result matrix from these sub matrices (corresponding to the process mount the execution

Duc Kien NGUYEN: LRIA-Université Paris 8, France (e-mail: Kien.Duc-Nguyen@univ-paris8.fr).

Quoc Trung HA: Faculty Information Technology, Hanoi University of Technology, Vietnam (e-mail: Trung.Ha-Quoc@univ-paris8.fr).

tree). It is simple to solve this problem for a sequential machine, but for a parallel machine that makes a great difficulty. With a definite value of r, we can manually do it like [17] and [18] made (r = 1, 2, 3) but in general case, there does not exist yet the solution. In this paper, we present our method which can determine all the nodes at the unspecified level r in the execution tree of Winograd algorithm, and show the relation's expression between the result matrix and the sub matrices at the level recursion r, this expression allows us to calculate directly the result matrix from the sub matrices calculated by parallel matrix multiplication algorithms at the bottom level.

II. Background

A. Winograd Algorithm

We start by considering the formation of the matrix product $Q=XY$, where

$$Q \in \Re^{m\times n}, X \in \Re^{m\times k} \text{ and } Y \in \Re^{k\times n}.$$

We will assume that m, n, and k are all even integers. By partitioning

$$X = \begin{pmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{pmatrix},\ Y = \begin{pmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{pmatrix},\ \text{and Q} = \begin{pmatrix} Q_{00} & Q_{01} \\ Q_{10} & Q_{11} \end{pmatrix}$$

where

$$Q_{ij} \in \Re^{\frac{m}{2}\times\frac{n}{2}}, X_{ij} \in \Re^{\frac{m}{2}\times\frac{k}{2}} \text{ and } Y_{ij} \in \Re^{\frac{k}{2}\times\frac{n}{2}},$$

it can be shown [6, 11] that the following computations compute $Q = XY$:

$$\begin{aligned}
&S_0 = X_{10} + X_{11} \quad S_1 = S_0 - X_{00} \quad S_2 = X_{00} - X_{10} \\
&S_3 = X_{01} - S_1 \quad S_4 = Y_{01} - Y_{00} \quad S_5 = Y_{11} - S_4 \\
&S_6 = Y_{11} - Y_{01} \quad S_7 = S_5 - Y_{10} \\
&M_0 = S_1S_5 \quad M_1 = X_{00}Y_{00} \quad M_2 = X_{01}Y_{10} \\
&M_3 = S_2S_6 \quad M_4 = S_0S_4 \quad M_5 = S_3Y_{11} \qquad (1) \\
&M_6 = X_{11}S_7 \\
&T_0 = M_0 + M_1 \quad T_1 = T_0 + M_3 \\
&Q_{00} = M_1 + M_2 \quad Q_{01} = T_0 + M_4 + M_5 \\
&Q_{10} = T_1 - M_6 \quad Q_{11} = T_1 + M_4
\end{aligned}$$

The W-algo does the above computation recursively until one of the dimensions of the matrices is 1.

B. Cannon Algorithm

Cannon algorithm [12] is a commonly used parallel matrix multiply algorithm based on the T-algo. It can be used on any rectangular processor templates and on matrices of any dimensions [9]. For simplicity of discussion, we only consider square processor templates and square matrices. Suppose we have p^2 processors logically organized in a $p \times p$ mesh. The processor in i^{th} row and j^{th} column has coordinates (i, j), where $0 \le i, j \le p-1$. Let matrices X, Y, and Q be of size $m \times m$. For simplicity of discussion we assume m is divisible by p. Let $s = m/p$. All matrices are partitioned into $p \times p$ blocks of $s \times s$ sub matrices. The block with coordinates (i, j) is stored in the corresponding processor with the same coordinates. With the addition of a link between processors on opposite sides of the mesh (a torus interconnection), the mesh can be thought of as composed of rings of processors both in the horizontal and vertical directions. The Cannon method requires communication between the processors of each ring in the mesh. The blocks of the matrix X are passed in parallel to the left along the horizontal rings. The blocks of the matrix Y are passed to the top along the vertical rings. This communication pattern results in the shifting leftwards of matrix X and upwards of matrix Y. Let X_{ij}, Y_{ij}, Q_{ij} stand for the blocks of X, Y, Q respectively stored in the processor with coordinates (i, j). The following pseudo code describes the Cannon algorithm.

The complete i^{th} row of X is shifted leftward i times
(i.e., $X_{ij} \leftarrow X_{i,j+i}$)
The complete j^{th} column of Y is shifted upward j times
(i.e., $Y_{ij} \leftarrow X_{i+j,j}$)
$Q_{ij} = X_{ij}Y_{ij}$ for all processors (i, j)
DO $(p-1)$ times
    Shift X leftwards and Y upwards
    (i.e., $X_{ij} \leftarrow X_{i,j+1}$; $Y_{ij} \leftarrow Y_{i+1,j}$)
    $Q_{ij} = Q_{ij} + X_{ij}Y_{ij}$ for all processors
ENDDO

The running time of the Cannon algorithm consists of two parts: the communication time T_{shift} and the computation time T_{comp}. On the distributed memory computer, the communication time for a single message is

$$T = \alpha + \beta n,$$

Where α is the latency, β is the byte-transfer rate, and n is the number of bytes in the message. In the Cannon method, both matrices X and Y are shifted p times. There are a total of $2p$ shifts. The total latency is $2p\alpha$. In each shift a sub matrix of order $(m/p \times m/p)$ is passed from one processor to another, where m is the dimension of the matrices. Therefore the total byte transfer time is $2p\beta B(m/p)^2$, where B is the number of bytes used to store one entry of the matrices. The total communication time is

$$T_{shif} = 2p\alpha + \frac{2B\beta}{p}m^2. \qquad (2)$$

The computation time is

$$T_{comp} = \frac{2t_{comp}}{p^2} m^3, \quad (3)$$

where t_{comp} is the execution time for one arithmetic operation. Here we assume that floating point addition and multiplication has the same speed. The total running time is

$$T(m) = \frac{2t_{comp}}{p^2} m^3 + \frac{2B\beta}{p} m^2 + 2p\alpha . \quad (4)$$

In order to make the Cannon algorithm work, an additional working space of size m^2 is needed to temporarily store the products of the sub matrices of X and Y.

III. General Scalable Aparallelizing Winograd Algorithm

A. Winograd-Cannon Algorithm and Pattern to Store the Matrices for this Algorithm Class

The motivation for the Winograd-Cannon algorithm comes from the observation that the W-algo is most efficient for large matrices and therefore should be used at the top level (between processors) instead of the bottom level (within one processor). The 7 sub matrix multiplications of the W-algo at each recursion seem at first to lead to a task parallelism. The difficulty in implementing a task parallelism of the W-algo on a distributed memory computer results from the fact that the matrices must be distributed among the processors. Sub matrices in the W-algo must be stored in different processors and if tasks are spawned these sub matrices must be copied or moved to the appropriate processors [7].

For a distributed memory parallel algorithm the storage map of sub matrices to processors is a primary concern. If the sub matrices used in the W-algo are stored among processors in the same pattern at each level of recursion, then they can be added or multiplied together just as if they are stored within one processor. Here we introduce a new pattern to store the matrices. Figure 1 and 2 shows the pattern of storing matrix X with 6 x 6 blocks when the recursion level is 1. Figure 1 is from a matrix point-of-view. Note that the four sub matrices with 3 x 3 blocks are stored among the 9 processors in the same pattern. Figure 2 is from a processor point-of-view. Each processor stores one block of the four sub matrices. Figure 3 and 4 show the pattern when the recursion level is 2. The four sub matrices with 6 x 6 blocks are stored in the same pattern, as well as the 16 sub matrices with 3 x 3 blocks. This pattern can be easily replicated for higher levels of recursion.

These patterns of storing matrices make it possible for all the processors act like one processor. Each processor has a portion of each sub matrix at each recursion level. The addition (or subtraction) of sub matrices performed in the W-algo at all recursion levels can thus be performed in parallel without any inter processor communication. From the processor point-of-view, each processor does its local sub matrix additions and subtractions. Sub matrix multiplications are calculated recursively using the W-algo. At the last level of recursion the sub matrix multiplications are calculated using the BMR method. Therefore, the Winograd-Cannon algorithm uses the W-algo at the top level and the T-algo at the bottom level.

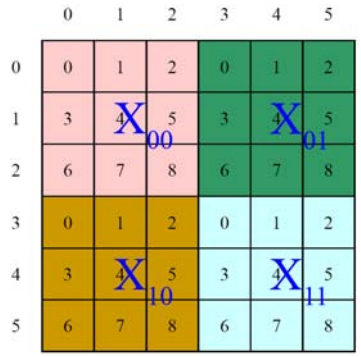

Figure 1: Matrix X with 6 × 6 blocks is distributed over a 3 × 3 processor template from a matrix point-of-view. The 9 processors are labeled from 0 to 8. This pattern is used in the Winograd-Cannon algorithm when the recursion level is 1.

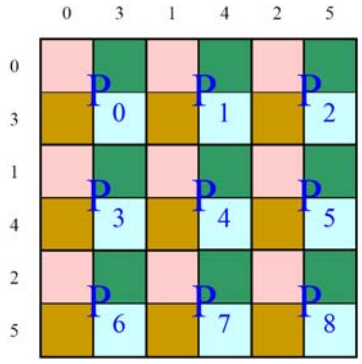

Figure 2: Same as Figure 1, but from a processor point-of-view.

Suppose the recursion level in the W-algo is r. Let $n=m/p$, $m_0=m/2$, and $n_0=m_0/p$. Assume n, m_0, and n_0 are all integers. Since there are 15 sub matrix additions and subtractions and 7 sub matrix multiplications in each recursion, the total running time for the Winograd-Cannon algorithm is

$$T(m) = 15T_{add}\left(\frac{m}{2}\right) + 7T\left(\frac{m}{2}\right) \quad (5)$$

where $T_{add}(\frac{m}{2})$ is the running time to add or subtract sub matrices of order $m/2$. Note that there are p^2 processors running in parallel. Therefore

$$T_{add}\left(\frac{m}{2}\right) = \frac{\left(\frac{m}{2}\right)^2 t_{comp}}{p^2} . \quad (6)$$

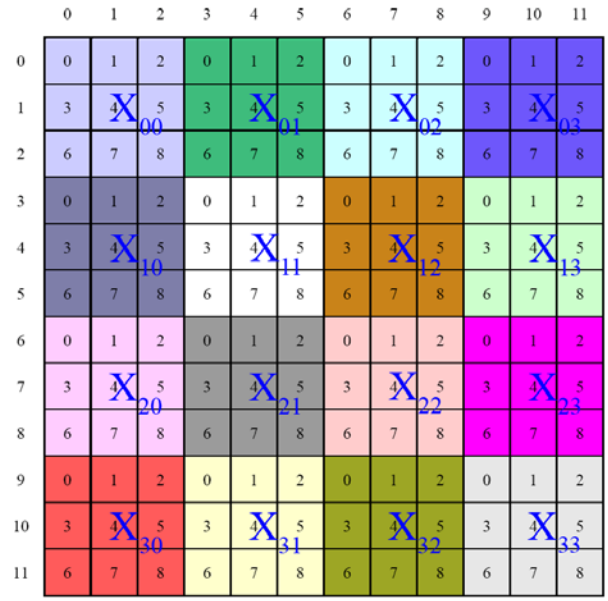

Figure 3: Matrix X with 12 × 12 blocks is distributed over a 3 × 3 processor template from a matrix point-of-view. The 9 processors are numbered from 0 to 8. This pattern is used in the Winograd-Cannon algorithm when the recursion level is 2.

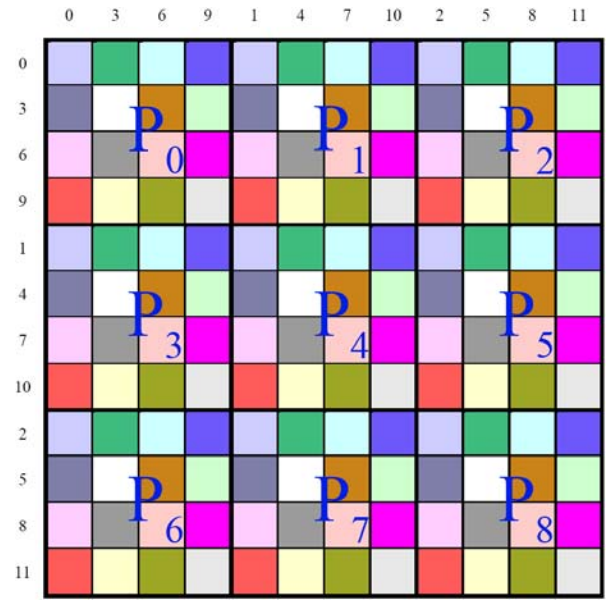

Figure 4: Same as Figure 3, but from a processor point-of-view.

Substitute the above formula into equation 8 we have

$$T(m) = \left(\frac{15t_{comp}}{4p^2}\right)m^2 + 7T\left(\frac{m}{2}\right) = sm^2 + 7T\left(\frac{m}{2}\right) \quad (7)$$

where $s = \left(\frac{15t_{comp}}{4p^2}\right)$. Use the above formula recursively to obtain

$$\begin{aligned}
T(m) &= sm^2 + 7T\left(\frac{m}{2}\right) \\
&= sm^2 + 7\left(s\left(\frac{m}{2}\right)^2 + 7T\left(\frac{m}{4}\right)\right) \\
&= sm^2\left(1+\frac{7}{4}\right) + 7^2 T\left(\frac{m}{2^2}\right) \\
&= sm^2\left(1+\frac{7}{4}\right) + 7^2\left(s\left(\frac{m}{4}\right)^2 + 7T\left(\frac{m}{8}\right)\right) \\
&= sm^2\left(1+\frac{7}{4}+\left(\frac{7}{4}\right)^2\right) + 7^3 T\left(\frac{m}{2^3}\right) \qquad (8) \\
&\vdots \\
&= sm^2\left(1+\frac{7}{4}+\cdots+\left(\frac{7}{4}\right)^{r-1}\right) + 7^r T\left(\frac{m}{2^r}\right) \\
&= sm^2 \frac{1-\left(\frac{7}{4}\right)^r}{1-\frac{7}{4}} + 7^r T(m_0) \\
&\approx \frac{4}{3}s\left(\frac{7}{4}\right)^r m^2 + 7^r T(m_0)
\end{aligned}$$

At the bottom level, the Winograd-Cannon algorithm uses the Cannon algorithm for sub matrix multiplications. Therefore we can use equation 4 to find $T(m_0)$. Substituting the value of $T(m_0)$ and s we have

$$\begin{aligned}
T(m) &\approx \frac{5\left(\frac{7}{4}\right)^r t_{comp}}{p^2} m^2 + 7^r\left(\frac{2t_{comp}}{p^2} m_0^3 + \frac{2B\beta}{p} m_0^2 + 2p\alpha\right) \qquad (9) \\
&= \left(\frac{7}{8}\right)^r \frac{2t_{comp}}{p^2} m^3 + \frac{5\left(\frac{7}{4}\right)^r t_{comp}}{p^2} m^2 + \left(\frac{7}{4}\right)^r \frac{2B\beta}{p} m^2 + 7^r(2p\alpha)
\end{aligned}$$

There are four terms in the above equation. The first term is a cubic term with respect to m. It is the computational dominant part and it decreases as the recursion level r increases. The second term is quadratic and it results from the additional sub matrix additions and subtractions in the W-algo. It increases as r increases. The last two terms represent the communication time. They increase as r increases. Since the first term is the dominant cubic term, the Winograd-Cannon algorithm should be faster than the Cannon algorithm when m is large enough.

B. *Recursion Removal in Fast Matrix Multiplication*

In formula 9, we showed that the total running time for the Winograd-Cannon algorithm decreases when the recursion level r increases. This result is also correct when we change Cannon algorithm at the bottom level by the other parallel MM algorithms. To use the Winograd algorithm at the top level, the most significant point is to determine the sub matrices after having recursively executed r time the formula 1 (these sub matrices are corresponding to the nodes of level r in the execution tree of Winograd algorithm) and then to find the result matrix from these sub matrices (the process to mount the execution tree). It is simple to solve this problem for a sequential machine, but for a parallel machine that makes a great difficulty. With a definite value of r, we can manually do it like [17] and [18] made (r = 1, 2, 3) but in general case, there does not exist yet the solution. The following part presents our method which can determine all the nodes at the unspecified level r in the execution tree of Winograd algorithm and determine the direct relation between the result matrix and the sub matrices at the level recursion r.

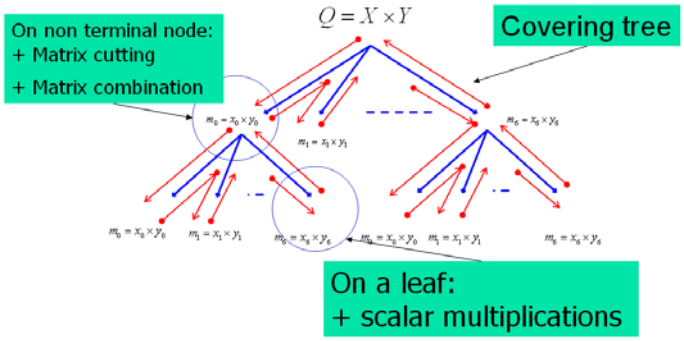

Figure 5: Behavior of Winograd algorithm

The Winograd matrix multiplication formula can be represented by:

$$
\begin{aligned}
m_0 &= x_{00} \times y_{00};\\
m_1 &= x_{01} \times y_{10};\\
m_2 &= \left(x_{10} + x_{11}\right) \times \left(-y_{00} + y_{01}\right);\\
m_3 &= \left(-x_{00} + x_{10} + x_{11}\right) \times \left(y_{00} - y_{01} + y_{11}\right);\\
m_4 &= \left(x_{00} - x_{10}\right) \times \left(-y_{01} + y_{11}\right);\\
m_5 &= \left(x_{10} + x_{11}\right) \times \left(y_{01} + y_{11}\right);\\
m_6 &= x_{11} \times \left(-y_{00} + y_{01} + y_{10} - y_{11}\right);\\
q_{00} &= m_0 + m_1;\\
q_{01} &= m_0 + m_2 + m_3 + m_5;\\
q_{10} &= m_0 + m_3;\\
q_{11} &= m_0 + m_3 + m_4 + m_6;
\end{aligned}
\qquad (10)
$$

We represent the execution of the algorithm like a tree (see figure 5) of the recursive calls. Each non terminal node corresponds to an execution of the algorithm, which comprises a matrix cutting entry by 4, in order to be able to create 7 calls representing the formulas 10. Until the leaf nodes, where we cannot cut out the matrices, the size of the matrices is 1, we make the scalar multiplication. After obtaining the scalar products, they are combined consecutively to obtain the produced matrices of higher sizes.

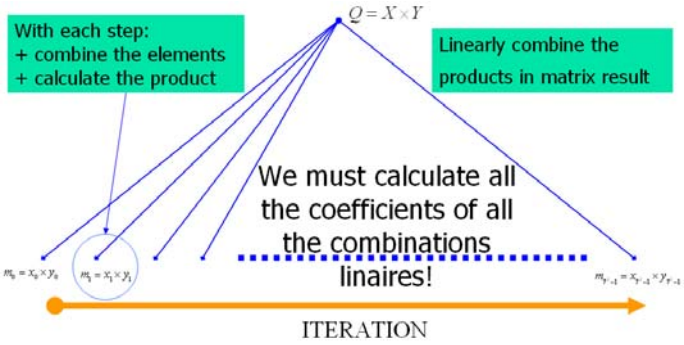

Figure 6: After recursion removal

By noticing that the operations matrix cutting are executed on each node of the tree, while the multiplications are executed only on the leafs, we has a natural idea to build an algorithm which will simulate calculations at low the level (on the leafs) of the tree as in the figure 6. To represent these nodes, it is just enough an iteration, because the number of these nodes is known. It remains to determine which calculation and which parameters it is necessary to realize in each iteration. In fact, on each node, calculation must be form:

$$
\begin{aligned}
m_l &= \sum_{i=0,1}\sum_{j=0,1} x_{ij} SX(l,i,j) \times \sum_{i=0,1}\sum_{j=0,1} y_{ij} SY(l,i,j)\\
l &= 0...6
\end{aligned}
\qquad (11)
$$

The coefficients $SX(l,i,j)$, $SY(l,i,j)$ are obtained while following the way which leads root to the node considered. Let us consider an unspecified element x_{ij}. The coefficient of x_{ij} in the representation of the node l is obtained like result of all the intermediate calculations, executed on each node in the way of the root to the node. On each node, the element's coefficient obtained is determined by:

- to which recursive call it corresponds,
- in which quarter of matrix the element is considered.

We represent the Winograd formula:

$$
\begin{aligned}
m_l &= \sum_{i=0,1}\sum_{j=0,1} x_{ij} SX(l,i,j) \times \sum_{i=0,1}\sum_{j=0,1} y_{ij} SY(l,i,j)\\
l &= 0...6 \text{ et } q_{ij} = \sum_{l=0}^{6} m_l SQ(l,i,j)
\end{aligned}
\qquad (12)
$$

In fact, the contents of SX, SY, and SQ are:

$$SX = \begin{pmatrix} -1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad SY = \begin{pmatrix} 1 & -1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & -1 & -1 & 1 \end{pmatrix}$$

$$SQ = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Each of 7k produces can be represented in the following way:

$$m_l = \sum_{i=0,n-1 j=0,n-1} x_{ij} SX_k(l,i,j) \times \sum_{i=0,n-1 j=0,n-1} y_{ij} SY_k(l,i,j)$$
$$l = 0...7^k\text{-1 et } q_{ij} = \sum_{l=0}^{7^k-1} m_l SQ_k(l,i,j) \tag{13}$$

In fact, $SX=SX_1$, $SY=SY_1$, $SQ=SQ_1$. We suppose $l=l_1l_2...l_k$ base on 7, $i=i_1i_2...i_k$ base on 2, $j=j_1j_2...j_k$ base on 2. The way of the root towards the node considered is expressed thereafter $l_1l_2...l_k$. We considers coefficient $SX_k(l,i,j)$ of element x_{ij} in the l^{st} product. At the first level, the recursive call number is m_1, element x_{ij} is in the district (i_1, j_1) of the matrix, therefore the coefficient of the matrix containing x_{ij} is $SX(l_1,i_1,j_1)$. At the second level, when this matrix is divided into 4 matrices, the matrix which contains x_{ij} will obtain coefficient $SX(l_2,i_2,j_2)$. So at the level s, the coefficient utilized is $SX(l_s,i_s,j_s)$. By applying the distributive and the associativeness of the addition and the multiplication scalars, the coefficient of x_{ij} is the product from 1 to k : $SX(l_1,i_1,j_1) \times SX(l_2,i_2,j_2) \times \ldots \times SX(l_s,i_s,j_s)$. More generally, we have:

$$SX_k(l,i,j) = \prod_{r=1}^{k} SX(l_r,i_r,j_r)$$
$$SY_k(l,i,j) = \prod_{r=1}^{k} SY(l_r,i_r,j_r) \tag{14}$$
$$SQ_k(l,i,j) = \prod_{r=1}^{k} SQ(l_r,i_r,j_r)$$

Apply (14) in (13) we have nodes leafs m_l and all the elements of result matrix.

To parallel Winograd algorithm, we stop at level r. We have

$$M_l = \sum_{\substack{i=0,2^r-1 \\ j=0,2^r-1}} X_{ij} SX_r(l,i,j) \times \sum_{\substack{i=0,2^r-1 \\ j=0,2^r-1}} Y_{ij} SY_r(l,i,j)$$
$$l = 0...7^r\text{-1 et } Q_{ij} = \sum_{l=0}^{7^r-1} M_l SQ_r(l,i,j) \tag{15}$$

with

$$X_{ij} = \begin{pmatrix} x_{i*2^{k-r},j*2^{k-r}} & \ldots & x_{i*2^{k-r},j*2^{k-r}+2^{k-r}-1} \\ \ldots & \ldots & \ldots \\ x_{i*2^{k-r}+2^{k-r}-1,j*2^{k-r}} & \ldots & x_{i*2^{k-r}+2^{k-r}-1,j*2^{k-r}+2^{k-r}-1} \end{pmatrix}$$

$$Y_{ij} = \begin{pmatrix} y_{i*2^{k-r},j*2^{k-r}} & \ldots & y_{i*2^{k-r},j*2^{k-r}+2^{k-r}-1} \\ \ldots & \ldots & \ldots \\ y_{i*2^{k-r}+2^{k-r}-1,j*2^{k-r}} & \ldots & y_{i*2^{k-r}+2^{k-r}-1,j*2^{k-r}+2^{k-r}-1} \end{pmatrix}$$

$$Q_{ij} = \begin{pmatrix} q_{i*2^{k-r},j*2^{k-r}} & \ldots & q_{i*2^{k-r},j*2^{k-r}+2^{k-r}-1} \\ \ldots & \ldots & \ldots \\ q_{i*2^{k-r}+2^{k-r}-1,j*2^{k-r}} & \ldots & q_{i*2^{k-r}+2^{k-r}-1,j*2^{k-r}+2^{k-r}-1} \end{pmatrix}$$

$$i = 0, 2^r - 1, j = 0, 2^r - 1$$

The product M_l of sub matrix $\begin{pmatrix} \sum_{\substack{i=0,2^r-1 \\ j=0,2^r-1}} X_{ij} SX_r(l,i,j) \end{pmatrix}$ and $\begin{pmatrix} \sum_{\substack{i=0,2^r-1 \\ j=0,2^r-1}} Y_{ij} SY_r(l,i,j) \end{pmatrix}$ will be calculated by parallel algorithms based on T-algo (Fox, Cannon, SUMMA, PUMMA, DIMMA…).

In continuation, we have directly sub matrix elements of result matrix by applying matrix additions

$$Q_{ij} = \sum_{l=0}^{7^r-1} M_l SQ_r(l,i,j)$$
$$i = 0, 2^r - 1, j = 0, 2^r - 1 \tag{16}$$

Here we have 2^{2r} elements → the parallelizing of these additions can be well done.

IV. Conclusion

For matrix multiplication on distributed memory computer, the use of Winograd algorithm at inter-processor level forms the most effective algorithms. A general model for this algorithm class has been presented: first of all, the expressions (13) and (14) make it possible to determine directly sub matrices at the specified level in the execution tree of Winograd algorithm then, the parallel matrix multiplication

algorithms are applied to calculate products of these sub matrices and finally, matrix result is calculated directly from these products thanks to the expressions (15) and (16).

This solution for the general case enables us to find algorithm optimal (which correspondent with a definite value of the level recursive and a parallel matrix multiplication algorithm at bottom level) for all the particular cases, moreover it open a complete new direction to parallelize the algorithms of Winograd class.

REFERENCES

[1] Pan, V. (1984). "How can we speed up matrix multiplication?" SIAM Review 26: 393-416

[2] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions *The 19th Annual ACM Conference on theory of computing* 1–6,New York, New York, United States,ACM Press 1987

[3] Strassen, Volker, "Gaussian Elimination is not Optimal", Numer. Math. 13, p. 354-356, 1969

[4] Winograd S. Some remarks on fast multiplication of polynomial *Traub*181-196, 1973

[5] Lavallée, I. (1982). Note sur le problème des tours de Hanoi. Acta Vietnamica

[6] S. Winograd, *On multiplication of* 2 x 2 *matrices,* Linear Algebra and its Applications, vol. 4, 1971, pp. 381-388.

[7] C.-C. Chou, Y.-F. Deng, G. Li, and Y. Wang, "Parallelizing Strassen's Method for Matrix Multiplication on Distributed Memory MIMD architectures," *Computers & Math. with Applications*, vol. 30, no. 2, p. 49, 1995

[8] DOUGLAS, C., HEROUX, M., SLISHMAN, G., *GEMMW: A Portable Level* 3 *BLAS Winograd Variant of Strassen's Matrix-Matrix Multiply Algorithm. Journal of Computational Physics 110* (1994), 1-10

[9] J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. *Concurrency: Practice and Experience,* 6(7):543-570, 1994

[10] LADERMAN, J., PAN, V., SHA, X., *On Practicad Algorithms for Accederated Matrix Multiplication. Linear Adgebra and Its Applications.* 1992, 557-588.

[11] Golub, G. H., and C. F. Van Loan, *Matrix Computations,* Johns Hopkins University Press, 2nd ed., 1989

[12] L. E. Cannon. A cellular computer to implement the kalman filter algorithm. 1969. Ph.D. Thesis, Montana State University

[13] G. C. Fox, S. W. Otto, and A. J. G. Hey. Matrix algorithms on a hypercube I: Matrix multiplication. *Parallel Computing,* 4:17-31, 1987

[14] S. Huss-Lederman, E. M. Jacobson, A. Tsao, and G. Zhang. Matrix Multiplication on the Intel Touchstone Delta. *Concurrency: Practice and Experience,* 6(7):571-594,1994

[15] R. van de Geijn and J. Watts. SUMMA Scalable Universal Matrix Multiplication Algorithm. LAPACK Working Note 99, technical report, University of Tennessee, 1995

[16] Jaeyoung Choi, "A Fast Scalable Universal Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers", *11th International Parallel Processing Symposium*, Geneva, SWITZERLAND, 1997

[17] Qingshan Luo and John B. Drake. A scalable parallel Strassen's matrix multiplication algorithm for distributed memory computers *Proceedings of the 1995 ACM symposium on Applied computing* 221–226, Nashville, Tennessee, United States, ACM Press 1995

[18] Brian Grayson, Ajay Shah and Robert van de Geijn "A High Performance Parallel Strassen Implementation," Department of Computer Sciences, The Unversity of Texas, TR-95-24, June 1995. Journal version: *Parallel Processing Letters*, Vol 6, No. 1 (1996) 3-12

[19] Pan V. How to multiply matrix faster *Springer-Verlag, L.N.C.S, Vol 179*, 1984

[20] Lavallée Ivan, Baala Hichem. Version itérative de la multiplication matricielle de Strassen. *C. R. Acad. Sci. Paris, 333, Série 1, p. 383-388*, 2001