Recursion removal in fast matrix multiplication

Trung HA QUOC,

***Abstract*— Recursion's removal improves the efficiency of recursive algorithms, especially algorithms with large formal parameters, such as fast matrix multiplication algorithms. In this article, a general method of breaking recursions in fast matrix multiplication algorithms is introduced, which is generalized from recursions removal of a specific fast matrix multiplication algorithm of Winograd.**

***Index Terms*— matrix multiplication, recursions removal, program transformation**

I. INTRODUCTION

Avoiding recursions in non trivial cases is always a challenge for computer scientists. Recursion is an elegant way to design algorithms, but its implementations are inefficient in practice, especially when the formal parameters are large in size. Thus for matrix multiplication, Straßen [Str1969], Winograd [Win1973] and Pan [Pan1984] have designed numerous recursive algorithms improving theoretical complexity of matrix multiplication. Unfortunately these algorithms or schemas are strongly recursive (7 interleaved recursive calls in [Str1969] and [Win1973]) and they have unacceptable performance in practice. In a previous article,[LaBa2000] have presented an iterative schema of Straßen MM algorithm without managing stacks. In another article [Lav1982] has shown how a peculiar coding of the Hanoï towers problem gives a direct iterative algorithm, not induced from a recursive one. In the present paper, firstly we combine those results to break recursions in the Winograd MM algorithm [Win1973] and generalize the recursion removal to the class of all the bilinear MM recursive algorithms defined by Pan in [Pan1984].

Matrix multiplication has many applications in computer graphics, linear computing and many other domains. 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 for MM (with $n^{log_2 7}$)than the classical one which need n^3 operations. Winograd improved the algorithm. And Pan in [Pan1984] has generalized all the bilinear algorithms for MM [Pan1984]. Bilinear algorithms include APA algorithms, algorithms of direct sum products and λ algorithms. The record of complexity owed to Coppersmith et Winograd in [**?**] is $n^2.397$, resulted from arithmetic aggregation. The common point behind these algorithms is to m a given complexity for some fixed size matrices. Then, by applying the formulas the algorithm recursively, replacing elements with matrices, the formula is applied for certain (unlimited in size) matrices. The schemas are simple, especially for those algorithms based on small size matrices, but they are strongly recursive and give unsatisfied performance. In [LaBa2000] has given a non recursive frame for the implementation of Straßen MM algorithm without using stacks. In order to do this, the number of steps for the execution of the algorithm must be determined precisely at the beginning of the execution. In [Lav1982], the idea behind the iterative solution of the Hanoï tower problem is to use a specific coding to determine exactly what to do in each step. In this work, firstly, we combine the results in [LaBa2000] and [Lav1982] for breaking recursions of MM algorithm of Winograd [Win1973]. Our major contribution is breaking recursions in all bilinear algorithmes, independently of the basic algorithm. The rest of the paper is organized as followed. In section 1 we introduce an iterative version of Winograd algorithm. In section 2 we generalize it for all bilinear MM algorithms which are defined in [Pan1984].

LRIA, 41 rue G. Lussac,75005 Paris. Email Trung.HA-QUOC@univ-paris8.fr

II. BREAKING RECURSIONS IN WINOGRAD'S FAST MM ALGORITHM

A. Winograd's Algorithm

According to the usual algorithm, 2 matrices multiplication is done as follows:

$$q_{ij} = \sum_{k=1}^{n} x_{ik} \times y_{kj} \tag{1}$$

(1) induces n^3 scalar multiplications and n^3 additions. For 2x2 matrices, Winograd has used the formulas [Win1973]

$$\begin{array}{l} p_0 = x_{00} \times y_{00}; \quad p_1 = x_{01} \times y_{10}; \\ p_2 = (x_{10} + x_{11}) \times (y_{01} - y_{00}) \quad ; \\ p_3 = (x_{10} + x_{11} - x_{00}) \times (y_{11} + y_{00} - y_{01}); \\ p_4 = (x_{00} - x_{10}) \times (y_{11} - y_{01}); \\ p_5 = (x_{10} + x_{11}) \times (y_{11} + y_{01}); \\ p_6 = x_{11} \times (y_{10} - y_{11} + y_{01} - y_{00}); \\ q_{00} = p_0 + p_1; \\ q_{01} = p_5 + p_2 + p_0 + p_3; \\ q_{10} = p_0 + p_3; \\ q_{11} = p_0 + p_3 + p_4 + p_6; \end{array} \tag{2}$$

For any pair of 2×2 matrices, the operation includes 7 multiplications and 15 additions/subtractions. By replacing the number x_{ij} and y_{ij} with 2×2 matrices, we can do the multiplication of 4×4 matrices with 49 multiplication and 165 additions. When we repeat it recursively, we obtain the algorithm for $2^k \times 2^k$ matrices. The number of scalars multiplications will be 7^k. For matrices with size other than $2^k \times 2^k$, zero's can be added to obtain the size $2^k \times 2^k$. This is the idea behind fast recursive matrix multiplication of Winograd.

B. Behavior of the recursive algorithm in the case $n = 2^k$

Without loss of generality, we can examine only the $2^k \times 2^k$ matrices. We start with two matrices X_{2^k} and Y_{2^k}. First,

the algorithm calculates 7 sums of $4.X_{2^{k-1}}$ matrices and 7 sums of $4.Y_{2^{k-1}}$ matrices. Second, it calls the same algorithm 7 times for multiplications of 7 pairs of those sums. For each multiplication the algorithm is executed with matrices of the size 2^{k-1} as parameters. For each computation of each multiplication, it computes 7 pairs of $X_{2^{k-2}}$ and $Y_{2^{k-2}}$ matrices and calls itself with the matrices of size 2^{k-2} (the behavior is repeated, but with another parameters). The procedure is repeated until X_1s and Y_1s are obtained and then scalar multiplications are effectively executed. At this point, the breakdown phase is finished. Now from m_1s, which are the products of X_1s and Y_1s it calculates m_2s, then from m_2s to m_4s and so on to m_{2^k}s, which are elements of the matrix-results. The up phase is finished and the resulting matrix is done.

This algorithm used exactly 7^k multiplications and $O(7^k)$ additions/subtractions. All of the multiplications are done in the first phase. In the first phase, the required memory space is $O(7^k)$ and at the second phase is also $O(7^k)$. The total memory space required is $O(7^k)$. But if we consider that stack operations strongly affects the performance of the program, the number of push and pop operations is $O(7^k)$, and with all of the matrix parameters (though in decreasing sizes), we can see that the performance cannot meet the theoretic expectation. This is why we look for an iterative algorithm without stack management. The starting point is nothing other than the recursive algorithm. The idea is to convert (implement) the recursive algorithm into an algorithm with the same theoretical complexity, but with better efficiency by avoiding stack management.

C. The iterative algorithm

The idea of the iterative algorithm is to compute the same atomic calculus as the recursive one but directly, as in the final step of the recursive algorithm but without the intermediate computing. It is possible because the number of the scalar multiplications is determinable (7^k), so we know exactly how many multiplications we must do. The rest is to determine exactly which to do and with which parameters in each step. A multiplication scalar must be done in each step. The rest is to compute 2 factors of the multiplication, each comes from one matrix. We can consider that each factor is the sum of all elements from each matrix, with coefficient 0s, -1, or 1. The execution of the recursive algorithm can be described by an execution tree [Wirth1976]. In such a representation each scalar multiplication is associated with a leaf of the execution tree. The path from the root to the leaf indicates the calls recursive leading to the corresponding multiplication. Thus, by the fact that all computations in each call are linear, we can composite them at only one computation at a leaf. At the leaf, the coefficient of each element is obtained by combining of all computation in the path from the root. In each recursive call, the coefficient obtained for each element is depended to:

- The index of the call.
- In which quarter one finds the element in the division of the matrix by 4 submatrices.

Let's express Winograd's formulas as arrays SX, SY, SQ such that:

$$\begin{aligned} m_l &= \sum_{i=0,1\,j=0,1} x_{ij}.SX(l,i,j) \\ &\times \sum_{i=0,1\,j=0,1} y_{ij}SY(l,i,j), l = 0...6 \\ \text{and} \quad q_{ij} &= \sum_{l=0}^{6} m_l SQ(l,i,j) \end{aligned} \tag{3}$$

Each of the 7^k multiplications has the following form

$$\begin{aligned} m_l &= \sum_{i=0,n-1\,j=0,n-1} x_{ij}SX_k(l,i,j) \\ &\times \sum_{i=0,n-1\,j=0,n-1} b_{ij}SY_k(l,i,j),\ l = 0...7^k - 1, \\ \text{and} \quad q_{ij} &= \sum_{l=0}^{7^k-1} m_l SQ(l,i,j) \end{aligned} \tag{4}$$

Let $l = l_1 l_2 \dots l_k$ base 7, $i = i_1 i_2 \dots i_k$ base 2, $j = j_1 j_2 \dots j_k$ base 2 The path from root to any leaf can be determined exactly, so the choice of a recursive call at each level is determined, respectively $l_1 l_2 \dots l_k$. We observe the coefficient $SX_k(l,i,j)$ of the element x_{ij} in the l's multiplication. At the first level, the choice is m_1, and the element x_{ij} is in the sub matrix quarter i_1, j_1 so the coefficient of the matrix contents x_{ij} is $SX(l_1, i_1, j_1)$. At the second level, when the matrix divides in 4 matrices, the one contents x_{ij} obtains the coefficient $SX(l_2, i_2, j_2)$. And that is correct for any level of the tree. As all the computations in each calls are additions, the coefficient of x_{ij} is the product from 1 to k: $SX(l_1, i_1, j_1) \times SX(l_2, i_2, j_2) \times \dots SX(l_k, i_k, j_k)$. The general formula must be:

$$SX_k(l,i,j) = \prod_{r=1}^{k} SX(l_r, i_r, j_r) \tag{5}$$

From (5), we can determine the coefficient of the element x_{ij} in the l-th multiplication. Only the elements with nonzero coefficients have meaning in the computation. Now let $MA_k(l)$ is the set of element with nonzero coefficient in l-th calculation, pairing with the values of the coefficient. Formally:

$$MX_k(l) = \{(i, j, SX_k(l,i,j)) | SX_k(l,i,j) \neq 0\} \tag{6}$$

The values in MX_1 are, respectively

$$MX(0) = \{(0,0,1),(1,1,1)\}; MX_1(1) = \{(1,0,1),(1,1,1)\}$$

….. Use (6) in (5) we have

$$MX_k(l) = \prod_{r=1}^{k} MX_1(l_r) \tag{7}$$

. The product is defined by combining two first members of elements of the set, and multiplying the thirt. For example

$$\begin{aligned} MX_1(0) \times MX_1(1) = \{&(01,00,1),(01,01,1), \\ &(11,10,1),(11,11,11)\} \end{aligned}$$

Similarly we have

$$MY_k(l) = \prod_{r=1}^{k} MY_1(l_r) \tag{8}$$

```
begin
   for l from 0 to 7^k − 1
   begin        l=l_1 l_2 ... l_k base 7
      compute MX_k(l)
      compute MY_k(l)
      compute m_l
   end
   for i from 0 to 2^k − 1
      for j from 0 to 2^k − 1
         i=i_1 i_2 ... i_k base 2
         j=j_1 j_2 ... j_k base 2
         compute MQ_k(i, j)
         compute q_i j
      end
end
```

Fig. 1. Schema of an iterative version of Winograd's matrix multiplication algorithm

The process of calculating q_{ij} is similar. We have to determine of which multiplication q_{ij} is the sum , and what is the coefficient. Similarly we have:

$$MQ_k(ij) = \prod_{r=1}^{k} MQ_1(i_r, j_r) \tag{9}$$

where element of $MQ_k(i,j)$ are pairs of the number of the multiplication and its coefficient in q_{ij}. (3) is converted to

$$\begin{aligned} m_l &= \sum_{(i,j,p)\in MX_k(l)} x_{ij}p \\ &\times \sum_{(i,j,p)\in MY_k(l)} y_{ij}p,\ l = 0...7^k - 1, \\ &\text{and then} \\ q_{ij} &= \sum_{(l,p)\in MC_k(ij)} m_l p \end{aligned} \tag{10}$$

From (10) we can build an iterative algorithm as in figure (1)

As we have seen, in the iterative algorithm, we added operations as computations MQ_k, MX_k, MY_k, which are all operations of integers and operations over 1,0, and -1. These operations are very low cost, so they don't affect the performance of the algorithm. It is easy to see now that we have achieved the theoretic performance of the original Winograd algorithm, without using any stacks. The use of the stack was replaced by using the number l, which is a string or a word that recognises the path of the tree. We can consider it as a method to avoid using stacks. The results are based on that all computing are linear, except the scalar multiplications. So a natural idea is to extend the results for a class of algorithms containing the algorithm discussed above.

III. Generalization for bilinear algorithms

Many algorithms like the one we have studied were discovered for fast MM. A lot of matrix operations are realized by recursive algorithms. We try to find a method to transform them to iterative forms without use of stacks. As help here comes Victor Pan [Pan1984] with his general forms of bilinear formulas. The model is formalized as followed.
Lets X, Y are 2 matrices of size $m \times n$ and $n \times p$.
Lets compute the product XY at the following :

$$\begin{aligned} m_l &= \sum_{i=0,m-1\, j=0,n-1} x_{ij}.SX(l,i,j) \\ &\times \sum_{i=0,n-1\, j=0,p-1} y_{ij}.SY(l,i,j), \\ &l = 0...M-1 \\ q_{ij} &= \sum_{l=0}^{l=M-1} m_l.SQ(l,i,j) \end{aligned} \tag{11}$$

Here, M multiplications m_l's play the role of the 7 multiplications of Winograd's algorithm, when SX, SY, SQ are arrays of constants from the set $\{1, 0, -1\}$, which plays the roles of the matrices of factors of components of m_l. M is the number of multiplication used for the algorithm, and called rank of the algorithm. The minimum rank of bilinear algorithm is called the rank of the problem of $m \times n$ by $n \times p$ MM. Recursive algorithms based on this algorithm for arbitrary matrices have the complexity of $M \log m.n.p.$ This model was used for researching new algorithms for MM with better complexity. Once we have an algorithm with rank M for the problem of (m, n, p) matrix multiplication, we can have a matrix multiplication algorithm with complexity $n^{3 \log M / \log mnp}$ for matrices with arbitrary large size. Winograd's formula is a particular case of this model. Similar to Winograd's algorithm, we can construct an iterative algorithm based on each algorithm from this class. We have noticed that in the formulas, the values of the elements of the matrices, which describe Winograd's algorithm have no influence on the iterative algorithm. So we can do the same for the general form (11).

The idea stays the same. We change the number 7 of the multiplication with M, the sizes of the matrices are respectively $m \times n$ and $n \times p$. As (11) replaces (3) and (6), (7), (8), (9) stay the same then (10) can be rewritten:

$$\begin{aligned} m_l &= \sum_{(i,j,r)\in MX_k(l)} x_{ij}r \times \\ &\sum_{(i,j,r)\in MY_k(l)} y_{ij}r, \quad l = 0...M^k - 1, \\ &\text{and then} \\ q_{ij} &= \sum_{(l,r)\in MQ_k(ij)} m_l r \end{aligned} \tag{12}$$

Based on (12) we have the same algorithm as in figure (2)

IV. Conclusions

We have presented an iterative schema for Winograd's matrix multiplication algorithm. The schema is extended for all bilinear algorithms of MM. It is easy to see that an algorithm described by (11) is not necessary to be for MM. So the schema can be used for every algorithm for matrix operations, provided it can be expressed by (11). Moreover (11) can be extended for 2,3,4 and more matrices, and the extended schema is still valid. Then the algorithm can be used for all bilinear matrix operation, including multiplication of 2, 3 and

```
begin
   for l from 0 to M^k − 1
   begin        l=l_1 l_2 ... l_k base M
      compute MX_k(l)
      compute MY_k(l)
      compute m_l
   end
   for i from 0 to m^k − 1
      for j from 0 to p^k − 1
         i=i_1 i_2 ... i_k base m
         j=j_1 j_2 ... j_k base p
         compute MQ_k(i, j)
         compute q_i j
      end
end
```

Fig. 2. Schema of an iterative version of general matrix multiplication algorithm

more matrices, or 1 matrix (by example in the computation of determinant), ...and so on. It is easy to see that the order of execution is not important. So we can divide the M^k multiplications as multi sessions and then execute each session separately as needed. That means we can execute the algorithm in parallel medium without changing many code (We have to change only the range of iteration for each process). Avoiding the recursion does not change the theoretical complexity, but in practice, it improves the performance significantly. For those who are interested in the implementation with an particular language, a package of set operations and a package of counting grand numbers are needed.

Based on the model of bilinear algorithms, many algorithms were discovered with better complexity. Using this method of conversion from recursive to non-recursive algorithm that avoids stack operations, we can extend the range of application of those theoretical results.

REFERENCES

[CopWin1987] **[CopWin1987]** 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

[Fid71] **[Fiduccia1971]** Charles M. Fiduccia. Fast matrix multiplication *Conference record of the third annual ACM symposium on Theory of computing* 45–49,Shaker Heights, Ohio, United States,1971

[LaBa2000] **[LavBaâ2000]** Lavallée Ivan, Baâla Hichem Version itérative de la multiplication matricielle de Strassen. *C. R. Acad. Sci. Paris, t. 333, Série 1, p. 383-388*, 2001.

[Lav1982] **[Lav1982]** Lavallée Ivan, Note sur le problème des tours de Hanoï. *Acta Vietnamica*, 1982.

[LeAb1991] **[Lee D.H. Aboelaze M.A.]** Lee D.H. Aboelaze M.A. Linear Speedup of Winograd's matrix multiplication algorithm using an array processor. *In 6th IEEE Distributed Memory Computing Confference*427-430, 1991

[LuoDrake1995] **[LuoDrake1995]** Qingshan Luo and John xB. 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

[Pan1978] **[Pan1978]** Pan V. Strassen's Algorithme is not optimal-trinlinear technique of aggreating, uniting and canceling for constructing fast algorithms for matrix operations. *19th Annuel Symposium on foundations of computer science* 166-176, 1978

[Pan1984] **[Pan1984]** Pan V. How to multiply matrix faster *Springer-Verlag, L.N.C.S, Vol 179*, 1984

[Stev] **[Stev]** Steven Huss-Lederman. Implementation of Strassen's Algorithm for Matrix Multiplication

[Str1969] **[Strassen1969]** Strassen V. Gaussian elimination is not optimal *Numerische Mathemathik, 13*354-356, 1969

[Win1973] **[Winograd1973]** Winograd S. Some remarks on fast multiplication of polynomial *Traub*181-196, 1973

[Wirth1976] **[Wirth1976]** Wirth Niklaus Algorithms + Data Structures = Programs *Prentice Hall* 1976