Research Journal of Information Technology

ISSN 1815-7432

Research Journal of Information Technology 3 (1): 61-67, 2011
ISSN 1815-7432 / DOI: 10.3923/rjit.2011.61.67
© 2011 Academic Journals Inc.

Cache Oblivious Matrix Multiplication Algorithm using Sequential Access Processing

[1]P.S. Korde and [2]P.B. Khanale
[1]Department of Computer Science, Shri Shivaji College, Parbhani (M.S.), India
[2]Department of Computer Science, Dnyanopasak College, Parbhani (M.S.), India

ABSTRACT

Hardware implements cache as a block of memory for temporary storage of data likely to be used again. Cache Oblivious Algorithms are typically analyzed using an idealized model of cache, but it is much easier to analyze than a real cache memory. Researchers have used cache oblivious algorithms for element ordering, matrix multiplication, matrix transposition and fast Fourier transform. In this study an efficient technique is proposed to manage cache memory. The new technique uses block recursive structure of two types only. The algorithm is tested on famous problem of matrix multiplication. It avoids jumps and cache misses are reduced to the order of $N^3/L\sqrt{M}$.

Key words: Cache memory, Cache oblivious, Cache hit, Cache miss, tag, sequential processing

INTRODUCTION

Cache memory is a high-speed, relatively small memory that represents a critical point in computer systems, since it eliminates the gab between a Central Processing Unit (CPU) and main memory speed. It has an access time smaller than that for main memory. So, the average access time is decreased in a great deal. Every time the CPU generates a reference for a specific word, the cache memory will be accessed to get that word if it is there; otherwise, the main memory is accessed to retrieve the line containing that word.

A Cache Oblivious Algorithm is designed to exploit the CPU cache without having the size for cache or the length of the cache lines as an explicit parameter. Cache Oblivious was conceived by Charles E. Leiserson as early as 1996 and first published by Harald Prokop in his master thesis at Massachusetts Institute of Technology in 1999 (Prokop, 1999).

The Cache Oblivious Algorithm is a simple and elegant model to design algorithm that perform well in hierarchical memory models ubiquitous on present hardware platforms. This model was first formulated in 1999 (Charles *et al.*, 1999) and since then is the topic of research.

Optimal cache oblivious algorithm is widely used in the Cooley-Tukey Fast Fourier Transform (FFT) algorithm, matrix multiplication, sorting, matrix transposition and several other problems (Bader *et al.*, 2002). Typically a cache oblivious algorithm works by a recursive divide and conquer methodology, where the problems are divided into smaller and smaller sub-problems. The resource uses can be optimally used by using recursive block structure algorithms. They are simple and portable. They use cache effectively because once the sub-problem fits into cache, its smaller sub problems can be solved into cache and with no cache misses (Frigo, 1999).

Matrix multiplication is one of the most studied computation problem (Coppersmith and Winograd, 1990). Consider the two matrices of size N × N, Matrix X = ($X_{i,} X_j$) and Y = ($Y_{i,} Y_j$). The product Z of these two matrices is given by:

$$Z_{ij} = \sum X_{ik} Y_{kj} \tag{1}$$

We can also assume that 99% of matrix multiplication resulting algorithms are similar to algorithm 1. But this algorithm involves a lot of memory jumps and makes bad use of cache memory. The cache misses in this algorithm are of high order.

Algorithm 1: Multiplication of two n-by-n matrices

```
for i from 1 to N do
for j from 1 to N do
C[ i, j ] := 0
for k from 1 to N to
C[ i, j ] := C[ i, j ] + A[ i, k ] * B [ k,j] ;
enddo;
enddo;
enddo;
```

To improve cache performance, the temporal and spatial locality of the access to the linearized matrix elements has to be improved. Most linear algebra libraries, like implementations of ATLAS (Whaley *et al.*, 2001), therefore use techniques like loop blocking and loop unrolling (Goto and van de Geijn, 2004). A lot of fine tuning is required to reach optimal cache efficiency on a given hardware and very often the tuning has to be repeated from scratch for a new machine. Recently, techniques have become popular that are based on a recursive block matrix multiplication (Gunther *et al.*, 1999). They automatically achieve the desired blocking of the main loop and the tedious fine tuning is restricted to the basic block matrix operations. Such algorithms are called cache oblivious (Demaine, 2002), emphasizing that they are inherently able to exploit a present cache hierarchy, but do not need to know about the exact structure of the cache.

Several approaches have been presented that use an element ordering based on the peano curve (Bader *et al.*, 2002). The peano curve results from a recursive construction idea. It totally avoids jump in the access to all matrices involved and shows optimal spatial locality but still this problem is open for research and modifications can be performed.

SEQUENTIAL ACCESS PROCESSING

In this study, we present an approach that uses a sequential access processing. The sequential access (Fig. 1) technique that converts the two dimensions into single. It is very easy to manipulate different matrix transformation operations.

However, our presented scheme totally avoids jumps in access to all matrices involved and reduces cache miss. It also uses only two types of recursive blocks.

We will demonstrate the general idea of sequential access based algorithm for 3-by-3 matrices. Initially we will convert a matrix of two dimensions into a single one. Algorithm 2 gives conversion algorithm.

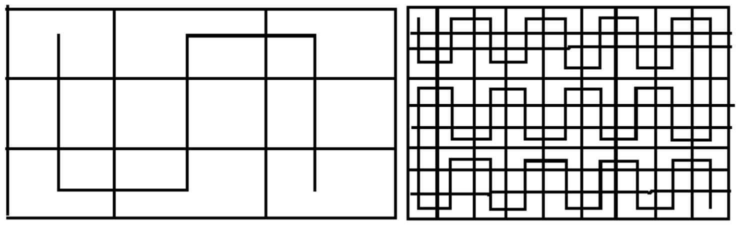

Fig. 1: Sequential access processing

Algorithm 2: Conversion of two dimensions into single dimension

```
counter =0;
for ( i = 0; i < n; i++)
for ( j = 0; j < n; j++)
cin>>a[ i, j];
    d[counter] = a[ i , j]
```

Now it is easy to access the matrix element levels. It gives better element access from the cache memory.

Lets consider the multiplication of 3-by-3 matrices. The elements matrices are combined as single matrices:

$$\underset{a}{\begin{pmatrix} a_0 & a_5 & a_6 \\ a_1 & a_4 & a_7 \\ a_2 & a_3 & a_8 \end{pmatrix}} \underset{b}{\begin{pmatrix} b_0 & b_5 & b_6 \\ b_1 & b_4 & b_7 \\ b_2 & b_3 & b_8 \end{pmatrix}} \underset{c}{\begin{pmatrix} c_0 & c_5 & c_6 \\ c_1 & c_4 & c_7 \\ c_2 & c_3 & c_8 \end{pmatrix}} \tag{2}$$

The elements C_k of matrix C are computed as sum of products of a and b:

$$C_k = \Sigma a_{i,} \, . \, b_{,j} \tag{3}$$

Where each set C_k contains multiplication of both pair elements. Figure 2 gives the graph representation of operations of 3-by-3 matrix multiplication. The nodes of graph are triples (i, j, k). In Fig. 2, two nodes are connected if the difference between two indices of nodes is not larger than one. Observe that all the nodes are connected and there are no jumps at all. It can be traversed in forward or backward direction.

The multiplications scheme presented can be easily extended to multiplication of 5-by-5 , 7-by-7 and so on. It can be used on any matrix multiplication as long as the matrix dimensions are odd numbers. It is necessary to use a block recursive approach. In case of a large matrix, the matrix can be divided into 3-by-3 recursive blocks as shown in Fig. 3.

Recursive blocks for 9-by-9 matrices are shown in Fig. 4. Observe that we have used only two types P and Q recursive blocks for the complete 9-by-9 matrix. This is the best possible way to divide the 9-by-9 matrix.

Also, observe that the range of indices within a matrix block is contiguous. So, it fulfills the basic requirement of recursive block and avoids the jumping of blocks.

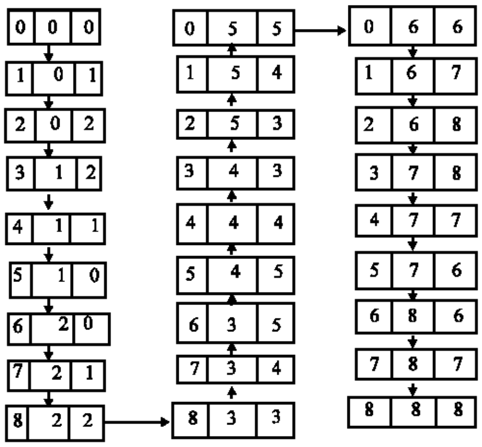

Fig. 2: Graph representation of operations of 3-by-3 matrix multiplication. For example, the element $C_0 = \{(a_0, b_0), (a_5, b_1), (a_6, b_2)\}$ and that of $C_1 = \{(a_1, b_0), (a_4, b_1), (a_7, b_2)\}$

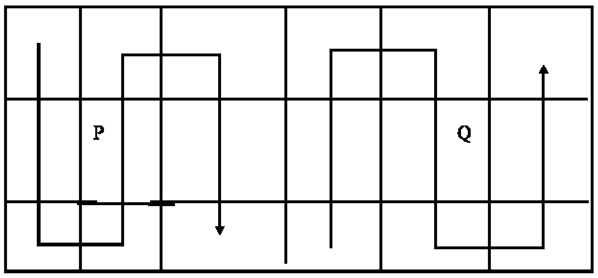

Fig. 3: 3-by-3 recursive blocks

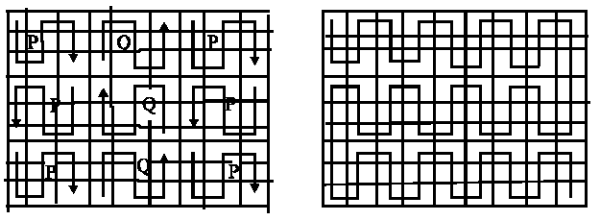

Fig. 4: Recursive blocks for 9-by-9 matrix

RECURSIVE SEQUENTIAL MULTIPLICATION

Now we will show the use of recursive blocks P and Q in case of 9-by-9 matrix multiplication. The two 9-by-9 matrices and their resultant matrix is given by Eq. 4:

$$\begin{pmatrix} Pa_0 & Qa_5 & Pa_6 \\ Pa_1 & Qa_4 & Pa_7 \\ Pa_2 & Qa_3 & Pa_8 \end{pmatrix} \begin{pmatrix} Pb_0 & Qb_5 & Pb_6 \\ Pb_1 & Qb_4 & Pb_7 \\ Pb_2 & Qb_3 & Pb_8 \end{pmatrix} \begin{pmatrix} Pc_0 & Qc_5 & Pc_6 \\ Pc_1 & Qc_4 & Pc_7 \\ Pc_2 & Qc_3 & Pc_8 \end{pmatrix} \tag{4}$$

We can write the equations for elements of resultant matrix as given in Eq. 5:

$$\begin{aligned} Pc_0 &= Pa_0\ Pb_0 + Qa_5\ Pb_1 + Pa_6\ Pb_2 \\ Qc_5 &= Pa_0\ Qb_5 + Qa_5\ Qb_4 + Pa_6\ Qb_3 \\ Qc_3 &= Pa_2\ Qb_5 + Qa_3\ Qb_4 + Pa_8\ Qb_3 \end{aligned} \tag{5}$$

Similarly we can write equations for Pc_1, Pc_2, Qc_4, Pc_6, Pc_7, Pc_8. If we consider the ordering of the matrix blocks , there are exactly four types of block multiplications as given in Eq. 6:

$$\begin{aligned} P &\xleftarrow{+} P\,.\,P \\ Q &\xleftarrow{+} P\,.\,Q \\ P &\xleftarrow{+} Q.\,P \\ Q &\xleftarrow{+} Q.\,Q \end{aligned} \qquad (6)$$

$$P \xleftarrow{+} P\,.\,P \text{ means } P = P + P.P$$

Thus we have a closed system of four multiplication schemes. Observe that we have got only four multiplication schemes which are much less than other works (Bader *et al.*, 2002).

We need to compute matrix operations for all five multiplication schemes. The matrix operations for P = P+P.P can be same as that given in Fig. 2. The matrix operations for Q = Q+Q.Q is given in Fig. 5.

After carefully examining all matrix operations we can prepare the table for sequential access operations for all matrix multiplication schemes as given in Table 1.

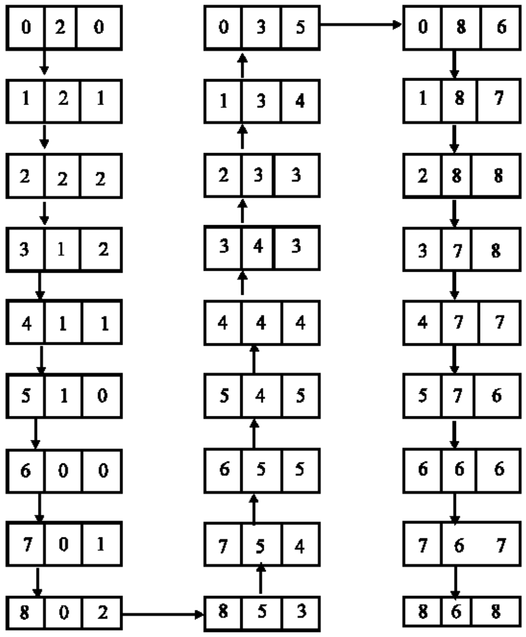

Fig. 5: Matrix operations for Q = Q + Q.Q

Table 1: Sequential operations of all matrix multiplications

$C \xleftarrow{+} A\,.\,B$	C	A	B
$P \xleftarrow{+} P\,.\,P$	+	+	+
$Q \xleftarrow{+} P\,.\,Q$	+	+	+
$P \xleftarrow{+} Q\,.\,P$	+	+	-
$Q \xleftarrow{+} Q\,.\,Q$	+	+	-

+ sign indicates that block scheme is executed in forward direction and - sign indicates that the block scheme is executed in reverse direction

Algorithm 3: Implementation of sequential access recursive scheme

```
Mult(int SA, int SB, int SC, int dim)
{
If (dim==1)
{
C[c] = C[c]+A[a].B[b]
}
else
{ Mult (SA, SB, SC, dim/3); a += SA; c += SC
Mult (SA, SB, SC, dim/3); a += SA; c += SC
Mult (SA, SB, SC, dim/3); a += SA; b +=SB

Mult (SA, -SB, SC, dim/3); a += SA; c - = SC
Mult (SA,-SB, SC, dim/3); a += SA; c - = SC
Mult (SA, -SB, SC, dim/3); a += SA; b += SB

Mult (SA, SB, SC, dim/3); a += SA; c += SC
Mult (SA, SB, SC, dim/3); a += SA; c += SC
Mult (SA, SB, SC, dim/3); c += SC; b += SB

Mult (SA, SB, SC, dim/3); a -= SA; c += SC
Mult (SA SB, SC, dim/3); a -= SA; c += SC
Mult (SA, SB, SC, dim/3); a -= SA; b += SB

Mult (SA, -SB, SC, dim/3); a -= SA; c- = SC
Mult (SA,-SB, SC, dim/3); a -= SA; c - =SC
Mult (SA, -SB, SC, dim/3); a-= SA; b += SB

Mult (SA, SB, SC, dim/3); a -= SA; c += S C
Mult (SA, SB, SC, dim/3); a -= SA; c+= SC
Mult (SA, SB, SC, dim/3);; b+-= SB; c += SC

Mult (SA, SB, SC, dim/3); a += SA; c += SC
Mult (SA ,SB, SC, dim/3); a += SA; c += SC
Mult (SA, SB, SC, dim/3); a += SA; b += SB

Mult (SA, -SB,SC, dim/3); a += SA; c - =SC
Mult (SA,-SB, SC, dim/3); a += SA; c - =SC
Mult (SA, -SB, -SC, dim/3); a += SA; b += SB

Mult (SA, SB, SC, dim/3); a += SA; c += SC
Mult (SA, SB, SC, dim/3); a += SA; c += SC
Mult (SA, SB, SC, dim/3);
}

}
```

IMPLEMENTATION

The sequential access recursive scheme can be implemented as given in algorithm 3. This algorithm takes four parameters SA, SB, SC and dim. SA, SB and SC indicates sequential access scheme. These parameters can take only +1 or -1 value. Parameter dim specifies the current matrix size. A, B and C are actual matrices and their indices are a, b and c. Matrix C is the resultant matrix. The name of the recursive function is Mult. The algorithm can be easily implemented by writing appropriate program in C or C++.

Data access locality and cache efficiency: During 3-by-3 block multiplication, the algorithm will perform 27 operations. Normally, for multiplication we require access of n^2 elements for n^3 operations (Bader *et al.*, 2002). But in this algorithm we need to access only 9 elements of all the three matrices. That is, we are accessing only k^2 elements during k^3 operations. So, we are getting k^2/k^3, that is, $k^{2/3}$ ratio. Hence, data access locality is of the order of $k^{2/3}$.

The ideal cache model assumes a computer consisting of a local cache of limited size and unlimited external memory. The cache consists of M words that are organized as cache lines of L words each. The replacement strategy is assumed to be ideal in the sense that the cache can foresee the future. Hence, if a cache line has to be removed from the cache, it will always be the one that is used farthest away in the future.

For a matrix multiplication of two N×N matrices, N being a power of three, out of N×N block, only n×n block is processed and will remain in cache. This block is also reused in next multiplication. Hence, only two blocks will be required to be transferred. Therefore, the number of cache lines transfers are given by $T(n) = 2\ n^2/L$.

Therefore, for N × N, $T(N) = (N/n)^3\ T(n)$ which is of the order of $\left(N^3/L\sqrt{M}\right)$.

CONCLUSION

We present here the problem of optimization of cache memory done by implementation of optimal oblivious matrix multiplication. Our algorithm uses only two types of recursive blocks. All the elements are accessed sequentially and there are no jumps at all. The number of cache miss are of the order of $\grave{O}\left(N^3/L\sqrt{M}\right)$.

REFERENCES

Bader, M.A., Z. Duan, J. Iacono and J. Wu, 2002. A locality-preserving cache-oblivious dynamic dictionary. Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Jan. 6-8, San Francisco, California, pp: 29-38.

Charles, M.F., E. Leiserson, H. Prokop and S. Ramachandaran, 1999. Cache oblivious algorithm. Proceedings of the 40th Annual Symposium on Foundation of Computer Science, (FOCS'99), New York, pp: 32-40.

Coppersmith, D. and S. Winograd, 1990. Matrix multiplication via arithmetic progression. J. Symbolic Computation, 9: 251-280.

Demaine, E.D., 2002. Cache-Oblivious Algorithms and Data Structures. University of Aarhus, Denmark.

Frigo, M., 1999. A fast fourier transform compiler 1999. ACM SIGPLAN Notices, 34: 169-180.

Goto, K. and R. van de Geijn, 2004. On reducing TLB misses in matrix multiplication. FLAME Working Note, University of Texas at Austin.

Gunther, F., M. Mehl, M. Pogl and C. Zenger, 1999. A cache-aware algorithm for PDEson hierarchical data structures based on space-filling curves. SIAM J. Sci. Comput., 28: 1634-1650.

Prokop, H., 1999. Cache-oblivious algorithms. Master's Thesis, Massachusetts Institute of Technology.

Whaley, R.C., A. Petitet and J.J. Dongarra, 2001. Automated empirical optimization of software and the ATLAS project. Parallel Computing, 27: 3-35.