ISSN No. 0976-5697

Volume 4, No. 1, January 2013 (Special Issue)

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

An Efficient Algorithm for Tile Size Selection

Mahesh Ubale*
Dept. of Computer Engineering/Information Technology
St. Vincent Pallotti College of Engineering & Technology,
Gavsi Manapur, Wardha Road, Nagpur
maheshubale@gmail.com

Priyanka Joshi
Department of Computer Science Engineering,
Visveswaraya National Institute of Technology,
Nagpur, Maharashtra, Pin 440010 (India)
priyankajoshi230@gmail.com

Abstract: Loop tiling is a well-known compiler transformation that increases data locality exposes parallelism and reduces synchronization costs. Tiling increases the amount of data reuse that can be exploited by reordering the loop iterations so that accesses to the same data are closer together in time [2]. Loop tiling is effective to improve hit ratio of cache. However, while eliminating self interference miss, tiling may produce small tiling factors for the cases of some arrays [3]. This paper presents a new algorithm for choosing problem-size dependent tiles based on the cache size, cache line size that eliminates the self-interference misses and uses 100% of available cache size. This algorithm covers the entire problem array size with the selected tile size. The tile size selection is our choice, so based on the different tile sizes available for A[i][j], we can choose a rectangular or square tile size.

Keywords: Loop tiling, self interference miss, and cache line size.

I. INTRODUCTION

Loop tiling is an effective method to reduce capacity miss and interference miss. However, loop tiling sometimes also may cause too small iterating times of inner loops, which will enlarge the overhead for executing loops and limit other optimizations. This is called the side effect of loop tiling. Especially, eliminating self interference miss makes tiling factors seriously dependent on the data layout of arrays [3]. In order to fully exploit the benefits of caches, compiler transformations have been developed to restructure the computation [2].

Loop tiling reduces capacity misses by enhancing data locality. However, tiling can introduce misses due to cache mapping conflicts, namely interference misses, since it causes non-contiguous regions of memory to be referenced within a single tile. For a given cache size and configuration, the amount of interference in a tiled loop nest varies significantly with both array size and tile size, such that a tile that works for one array size may cause a large amount of interference for other array sizes [4].

This paper is an improvement on few of the previously proposed algorithms in tile size selection. In this paper we present an efficient algorithm for tile size selection that will eliminate self-interference misses within iteration space. We have considered cache line size to be an essential factor in deciding the tile size. The available tile sizes avoid the side effect of a loop tiling caused due to selection of too small tiles. We can make a choice to use either rectangular or square tile once we get the tile sizes for an array space. The selected tiles uses the 100% of available cache and also covers (overlaps) the entire array space

II. BACKGROUND AND RELATED WORK

A. *Background*

Tiling can be applied to registers, the TLB, or any other level of memory hierarchy. We concentrate on tiling for the first level of cache memory [1].

Interference misses occur when a cache line that contains data that will be reused is replaced by another cache line. An interference miss is distinguished from a capacity miss because not all the data in the cache at the point of the miss on the displaced data will be reused. Intuitively, interference misses occur when there is enough room for all the data that will be reused, but because of the cache replacement policy data maps to the same location. Self-interference misses result when an element of the same array causes the interference miss. Cross interference misses result when an element of a different array causes the interference miss [1].

Tiling reduces the volume of data accessed between reuses of an element, allowing a reusable element to remain in the cache until the next time it is accessed.

```
(a) Matrix Multiply
   DO I = 1, N
     DO K = 1, N
       R = X(K, I)
       Do J = 1, N
          Z(J,I) = Z(J,I) + R* Y(J,K)
(b) Tiled Matrix Multiply
   DO KK = 1, N, TK
     DO JJ = 1, N, TJ
       DO I = 1, N
         DO K = KK, MIN(KK+TK-1,N)
           R = X(K,I)
           DO J = JJ, MIN(JJ + TJ-1,N)
              Z(J,I) = Z(J,I) + R * Y(J,K)
```

Figure 1. Matrix Multiplication as Tiling Example [1]

CONFERENCE PAPER
National Level Conference on
"Trends in Advanced Computing and Information Technology (TACIT - 2012)"
on 2nd September 2012
Organized by
St. Vincent Pallotti College of Engineering and Technology, Nagpur, India

B. Related Work [2] :

Several researchers have addressed the problem of minimizing cache interference [6, 7, 8, 9, 10, 11, 12, 13, 14]. Some of this work targets tiled loop nests, which are known to exhibit significant interference depending on the problem size. Lam, Rothberg and Wolf [l1], Coleman and McKinley [6] and Esseghir [7] have proposed tile size selection schemes for avoiding self interference in tiled loop nests.

Lam, Rothberg and Wolf [l1] present a model for Research Paper

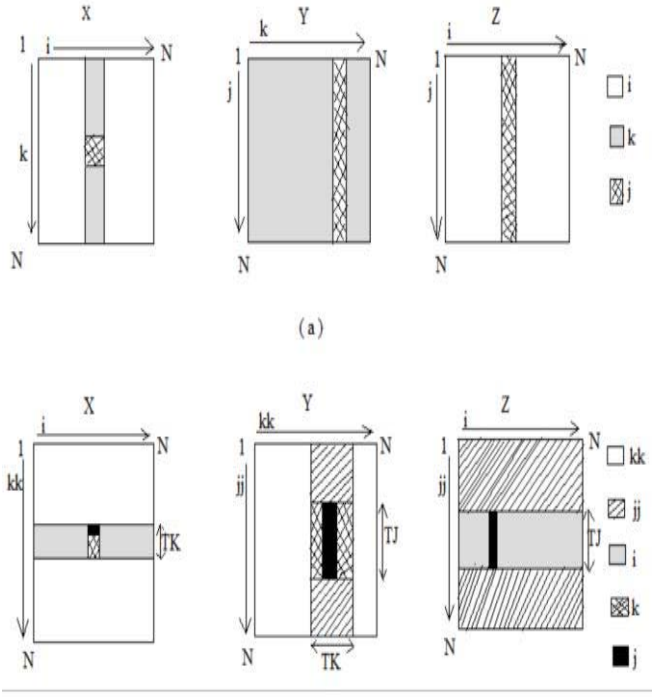

Figure 2. Iteration space traversal in (a) untiled and (b) tiled matrix multiply [1]

Esseghir's approach (ESS) [7] consists of selecting the maximum number of rows (assuming row-wise allocation) that fit in the cache. His approach eliminates self interference, but it can result in low cache utilization by leaving potentially large portions of the cache unused. Furthermore, it may result in low temporal locality, if the temporal reuse of outer tiled loops is not exploited. The result of not fully exploiting cache utilization and temporal reuse is an increase in capacity misses and loop overheads.

A third tile selection technique (TSS) was proposed by Coleman and McKinley [6]. TSS selects rectangular tiles that eliminate self interference and minimize cross-interference misses, achieving better performance than ESS. However, for some matrix sizes, TSS selects tiles that do not fully exploit temporal reuse, resulting in a high number of capacity misses.

Ghosh, Martonosi and Malik [9,10] proposed a technique for selecting tile sizes, based on their Cache Miss Equations framework for quantifying both capacity and interference misses. Their algorithm determines the largest rectangular tile without self interference that fits in an iteration subspace bounded by integer solutions to the cache miss equations.

predicting self interference and use it to evaluate self temporal interference in tiled matrix multiply. They also propose an algorithm (that we call LRW, as in [6]) for selecting the largest square tile that does not generate self interference for a particular array size. Although LRW improves the average cache performance for a range of array sizes, it chooses very small tiles for some array sizes, resulting in only a fraction of the cache being utilized and large loop overheads due to tiling.

Minimizing self interference independently of capacity and cross interference does not always result in the lowest miss rates or execution times. To minimize the total number of cache misses it is necessary to minimize the sum of the contributions of the capacity, cross and self-interference misses. Since self interference is often the main cause of interference misses, and self interference is costly to estimate [5, 10, 14], our work focuses on finding tiles that do not introduce self interference based on cache line size, cache size and array size.

III. TILE SELECTION ALGORITHM

A. Concept of Tiling:

Tiling reduces the volume of data accessed between reuses of an element, allowing a reusable element to remain in the cache until the next time it is accessed. Consider the code for matrix multiply in Figure l(a) and its corresponding reuse patterns illustrated in Figure 2(a). The reference Y(J,K) is loop-invariant with respect to the I loop. Each iteration of the loop also accesses one row each of X and Z. Therefore, 2*N + N2 elements are accessed per iteration of the I loop. Between each reuse of an element of Y there are N distinct elements of Z accessed on the J loop, N elements of the X array on the K loop, and N2 - 1 elements of Y array. If the cache is not large enough to hold this many elements, then the reusable Y data will be knocked out of the cache, and the program will have to repeat a costly memory access [1].

The largest tile with the most reuse on the I loop is the access to Y. We therefore target this reference to fit and stay in cache. We want to choose TK and TJ such that the TK x TJ sub matrix of Y will still be in the cache after each iteration of the I loop and there is enough room in the cache for the working set size of the I loop, TK * TJ + TK + TJ[1].

Define abbreviations and acronyms the first time they are used in the text, even after they have been defined in the abstract. Abbreviations such as IEEE, SI, MKS, CGS, sc, dc, and rms do not have to be defined. Do not use abbreviations in the title or heads unless they are unavoidable.

B. Inefficient cache usage:

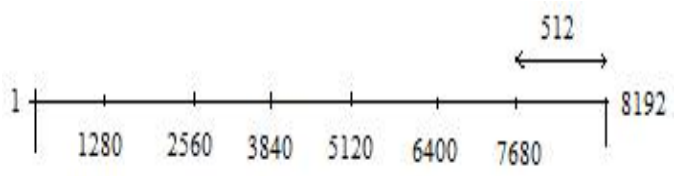

Figure 3. Column Layout for a 1280*1280 array in an 8192 element cache

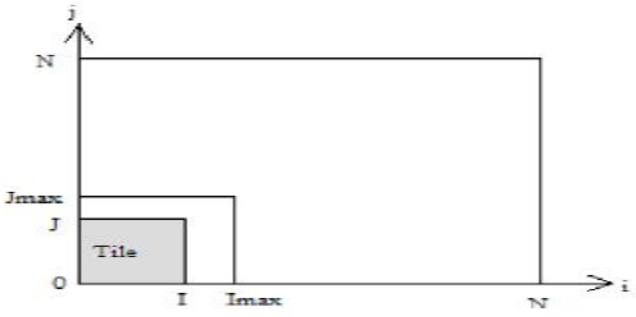

Figure 4. Limits of tile size as per theorem for A[N][N]

Consider the layout of 1280x1280 array Y in a direct mapped cache that can hold 8192 elements of Y as illustrated in Figure 3. We assume that the first element of array Y falls in the first position of the cache.

The number of columns (Col) that can fit in cache is

⌊Cache Size / N⌋

Where N is the column dimension (the consecutively stored dimension). For Figure 3, Col = 6 for a tile of 1280 x 6 Esseghir selects this tile [15]. A 1280 x 6 tile uses 93% percent of an 8192 element cache, but leaves a single contiguous gap of 512 elements that can't fit into the tile perfectly. If N evenly divides CS, we also select this tile size. Otherwise, we look for a smaller column dimension with a larger row size that does not incur interference which combines to use a higher percentage of the cache [1].

C. Tile size selection approach:

Given a target array size, cache line size, cache size and size of each array element, we now show how to select a desired square or rectangular tile.

a. Theorem [3]:

Let CS the size of cache, CLS the size of cache line. Let $CS=2^n$, $CLS=2^k$, base $_A$ be the base address of the array A $[N_1][N_2]$. We have the following theorem:

a) Theorem:

If $N_2=d*2^m$, where d is an odd number, m is the step of A, and k<=m<n, then the reference of A, A[i][j], won't cause self-interference misses within the range: $i0<=i<i0+2^{n-m}$, $j_0<=j<j_0+2^m-2^k+1$. (i_0 can be any value between 0 and N_1-1, j_0 can be any value between 0 and N_2- 1).

According to theorem 1, one can select suitable value of m, and adjust the N_2 as odd time of 2^m, it will eliminate the self-interference miss within iteration space **[**16].

In our approach we have considered array of size A[N][N] instead of $A[N_1][N_2]$ as considered in the above theorem. The theorem gives us the upper limit for the tile size as Imax and Jmax for which self-interference misses will be eliminated. We have the freedom to choose either a square tile or a rectangular tile with these constraints on tile size.

Our main intension is to utilize the cache size to the fullest i.e. 100% cache usage with the selected tile.

One more limitation for tile selection that we have considered is, the set of all possible tiles (square or rectangular) for a given array size and cache size is given by,

{ T = < I , J > | I*J ≤ min(N2, CS) }

Where CS is the maximum number of array elements that can fit in cache, I and J are respectively the size of the sub-column and sub-row of tile T and 1≤I, J≤CS. [2]

Sometimes it may happen that, for T < I, J >, (multiple of I) ≠ N or/and (multiple of J) ≠ N i.e. the selected tile does not completely overlap or covers the entire array space. We are finding tile that should cover up the array entirely. All the goals stated above are fulfilled by finding GCD of array size, cache size and cache line size.

The main idea behind our tile selection algorithm is finding a tile that will eliminate self-interference misses, the tile should use the cache fully to 100%, it should overlap the entire array and finally it should avoid side effect of a loop tiling.

Once we find the gcd of the three essential parameters in tile size selection, we get a set of values that can be chosen as tile on i and j axis for the array. We have to check these values for the two constraints that we have imposed on tiles and then we can select a square or rectangular tile as desired. Among the total tile resulted from algorithm, we have to avoid the small tile size and try for the larger tiles. This will give us less choice for tiles but will avoid side effect. The actual algorithm for which the results are tabulated in TABLE-I is given in Figure 5 below.

```
Algorithm tileSizeSelection (CS, CLS, array_size)
{
        CS= 2n CLS
        = 2k
        x = array_size / 2k
        for i = k, n
                for j = 1, x
                {
                if ( array_size = j * 2i )
                        {
                                m = i d = j
                        }
                                j++
                }
        Imax = 2n-m
        Jmax = 2m – 2k + 1
        gcd = Algo_gcd (CLS, CS, array_size) tile =
        Algo_factor (gcd)
}
```

Figure 5. Tile size selection algorithm

IV. RESULT ANALYSIS AND CONCLUSION

The result in the below table gives us the needed tile size as per our choice i.e. square or rectangular tile.

Consider second column, where L1 cache size is 16k. The corresponding row for Imax shows 32 and Jmax shows 385 and the last row shows three values 64, 32, 16. It means we are free to choose a square tile of size 16×16, 32×32 likewise we can choose rectangular tile of sizes 16×32, 16×64, 32×16, 32×64. But we can't choose a square tile of size 64×64 or a rectangular tile of size 64×32 or 64×16, this is mainly because of the maximum limit Imax as 32. This limitation says that we can't choose a tile with I greater than 32, because beyond this size the self-interference misses will occur.

The work done in this paper has a limited set of values for

CONFERENCE PAPER
National Level Conference on
"Trends in Advanced Computing and Information Technology (TACIT - 2012)"
on 2nd September 2012
Organized by
St. Vincent Pallotti College of Engineering and Technology, Nagpur, India

testing because of the gcd computation of the three parameters i.e. cache size, matrix size and cache line size. We have considered cache line size as an essential parameter in the experimentation with the implementation of the algorithm, so we got very limited choices for tile. If we give a bit relaxation by considering the gcd of cache size and matrix size only then the set of values will have a larger set to choose for tile size.

Also, the choice for m and d based on the theorem has made the choices for array size to be very specific in the experimentation. If we considered 2n to be the cache size in Kbytes and 2n as cache line size then some of the choices for matrix size for which the theorem satisfies with some integer values of m and d are as 640, 768, 896, 1280, 1536, 1792, 2560, 3584.

Table: 1 Possible Tile Sizes for Different Combination of CS, CLS and Array Size

Sr. No.	*L1 Cache size (bytes)*	*Cache Line Size (bytes)*	*Array Size (N)*	*Array Element Size (bytes)*	*No. of Elements in Single Cache Line*	*No. of Elements in Cache*	*No. of Elements in Array*	*n*	*k*	*m*	*d*	*Imax*	*Jmax*	*Possible Tile Size*
1	32k	64	1280	4	16	8192	1638400	15	6	8	5	128	193	16,8,4
2	16k	128	3584	2	64	8192	12845056	14	7	9	7	32	385	64,32,16
3	128k	256	1536	4	64	32768	2359296	17	8	9	3	256	257	64,32,16,8
4	64k	128	1792	4	32	16384	3211264	16	7	8	7	256	129	32,16,8
5	128k	128	1024	4	32	32768	1048576	17	7	10	1	128	897	32,16,8

V. REFERENCES

[1] Stephanie Coleman and Kathryn S. McKinley, Tile Size Selection Using Cache Organization and Data Layout, Conference on Programming Language Design and Implementation. Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation.

[2] Jacqueline Chame and Sungdo Moon, A tile selection algorithm for data locality and cache interference, International Conference on Supercomputing. Proceedings of the 13th international conference on Supercomputing.

[3] Xinda Lu, Jie Chen, "Eliminate Self-Interference Misses of Cache through Modifying Data Layout of Arrays," hpc, vol. 1, pp.259, The Fourth International Conference on High-Performance Computing in the Asia-Pacific Region-Volume 1, 2000.

[4] Michael E. Wolf. Improving Locality and Parallelism in Nested Loops. PhD thesis, Dept. of Computer Science, Stanford University, August 1992.

[5] Jacqueline Chame. Compiler Analysis of Cache Interference and its Applications to Compiler Optimizations. PhD thesis, Dept. of Computer Engineering, University of Southern California, 1997.

[6] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In Proceedings of the ACM SIGPLA N '95 Conference on Programming Language Design and Implementation, pages 279- 290, La Jolla, California, June 1995.

[7] Karim Esseghir. Improving data locality for caches. Master's thesis, Dept. of Computer Science, Rice University, September 1993.

[8] Christine Fricker, Olivier Temam, and William Jalby. Influence of cross-interferences on blocked loops: A case study with matrix-vector multiply. ACM ZYansaction on Programming Languages and Systems, 17(4):561-575, July 1995.

[9] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In Proceedings of the 1997 ACM International Conference on Supercomputing, Vienna, Austria, July 1997.

[10] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 228-239, San Jose, California, October 1998.

[11] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimization of blocked algorithms. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 63-74, Santa Clara, California, April 1991.

[12] Oliver Temam, Christine Fricker, Elana Granston, and William Jalby. Cache performance analysis. APPARC deliverables PMEP, Leiden University, July 1993.

[13] Oliver Temam, Elana Granston, and William Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In Proceedings of Supercomputing '99, pages 401-419, Portland, Oregon, November 1993.

[14] Michael E. Wolf. Improving Locality and Parallelism in Nested Loops. PhD thesis, Dept. of Computer Science, Stanford University, August 1992.

[15] K. Esseghir. Improving data locality for caches. Master's thesis Dept. of Computer Science, Rice University, Sepetember 1993

[16] J. Chen, Improving data localities through loop transformation, Ph.D thesis Dept. of Computer Science and Engineering, Shanghai Jiao TongUniversity, 1998.

CONFERENCE PAPER
National Level Conference on
"Trends in Advanced Computing and Information Technology (TACIT - 2012)"
on 2nd September 2012
Organized by
St. Vincent Pallotti College of Engineering and Technology, Nagpur, India