Proposal of Multiple Blocking and Its Efficiency in Matrix Operations

Satoshi Tateno and Takaomi Shigehara
Department of Information and Computer Sciences, Saitama University
Shimo-Okubo 255, Saitama, Saitama 338-8570, JAPAN
Phone: +81-48-858-9035, Fax: +81-48-858-3716
E-mail: tateno@me.ics.saitama-u.ac.jp, sigehara@me.ics.saitama-u.ac.jp

Abstract

In this paper, we propose a new blocking method, *multiple blocking*, and examine the efficiency of the method in basic matrix operations. In the best case for the matrix multiplication $C = AB + C$, the multiple blocking improves the performance by more than 10%, compared to the conventional single blocking method.

1 Introduction

Nowadays, the hierarchy structure of the memory plays a crucial role for obtaining high performance especially in a large scale numerical computation in various fields in science and technology. A typical structure of the memory hierarchy in standard modern computers is shown in Fig. 1. As well-known, upper layers have memory access speed far faster than lower layers, whereas the capacity of the upper layers is very limited. Thus, in order to avoid cache misses and keep the performance at a high level, it is important to divide the original data into several pieces of block, the size of which is suitably adjusted to the capacity available in cache. The so-called *blocking* is an absolutely indispensable technique for optimization in this context. Blocking reduces the number of cache misses as much as possible and it ensures a substantial improvement of memory access. In many applications such as the solution of linear systems in linear algebra, the use of a blocked algorithm, if possible, is a standard strategy to keep the performance at a higher level. Indeed, blocked algorithms have been already supported in LAPACK [1, 2].

However, the present status of blocking technique seems to be far from satisfactory for the modern computers with multi-layer caches. The conventional blocking corresponds to a *single* blocking and it does not take into account the difference in the capacity and memory access speed at each cache level. A further improvement in performance is expected if we incorporate with the memory hierarchy with multi-layer caches. This is exactly our purpose. In this paper, we propose a new blocking method, taking into account the multi-layer structure of cashes. Our proposed method, *multiple blocking*, does not treat multi-layer caches as integrated as in the conventional blocking, but it utilizes multi-layer caches in an explicit manner.

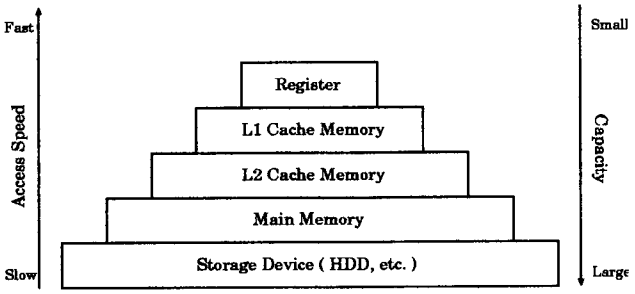

Figure 1: Memory hierarchy

2 Algorithm for multiple blocking

From the viewpoint of programming, multiple blocking is attained by a further blocking of the inner loops of a singly blocked source code. In our approach, the source code is divided into the two levels; the upper level is called FRAME, while the lower level is called CORE. In this section, we describe the algorithms both for FRAME and CORE. Sample programs are also shown.

2.1 FRAME part

FRAME is a routine at the upper level for multiple blocking. All control processes, such as error process, are performed in FRAME. In Fig. 2, we show pseudo-code for a core loop in FRAME for matrix multiplication. The algorithm of FRAME is as follows;

1. Divide data of arraies $A/B/C$ into small square pieces $A_{w0}/B_{w0}/C_{w0}$, and copy $A_{w0}/B_{w0}/C_{w0}$ to $A_w/B_w/C_w$, which are temporary `nb2`×`nb2` arraies. The memory access to $A_w/B_w/C_w$ is kept to be continuous, by setting `nb2` to a suitable size according to the capacity of L2 cache.

```
do j=1,N,nb2
  do i=1,M,nb2
    Cw(1:nb2,1:nb2)=0
    do k=1,K,nb2
      Aw=A(i:i+nb2-1,k:k+nb2-1)
      Bw=B(k:k+nb2-1,j:j+nb2-1)
      CORE(Aw,Bw,Cw)
    end do
    C(i:i+nb2-1,j:j+nb2-1)=alpha*Cw
        + beta*C(i:i+nb2-1,j:j+nb2-1)
  end do
end do
```

Figure 2: Core loop in FRAME for matrix multiplication

2. Call the subroutine CORE from FRAME. Calculate $C_w = A_w B_w + C_w$ in CORE. After calculation is completed in CORE, copy $\alpha C_w + \beta C_{w0}$ to C.

For matrix multiplication, there are two cases in CORE; One is the case that any matrices are not transposed such as $C_w = A_w B_w + C_w$, while the other is the case that one of A_w/B_w is transposed such as $C_w = A_w^T B_w + C_w$. For the latter case, FRAME copies a transposed array data to A_w or B_w in order to keep the memory access continuous in CORE.

2.2 CORE part

CORE is a routine for calculation of matrix operations, using small array data given from FRAME. We have to tune CORE as much as possible in order to improve the total performance. Note that conventional blocking treats multi-layer caches as integrated. As a result, blocking is not considered in CORE. On the other hand, multiple blocking utilizes multi-layer caches in a direct manner to improve the performance. As a result, blocking is indispensable in CORE, as well as in FRAME.

We show typical core loops in CORE in Fig.3. In Fig.3, (a) is a core loop for matrix multiplication $C_w = A_w B_w + C_w$, while (b) is a core loop for matrix multiplication $C_w = A_w^T B_w + C_w$. Blocking in CORE corresponds to *loop stripmining*, which cuts long loops into shorter ones to fit the data capacity in the innermost loops to the capacity of L1 cache. For this purpose, the stride for stripmining is adjusted to the capacity of L1 cache. By setting to a suitable value, we can largely suppress unnecessary memory access. For each loop, stripmining is independently applied and the stride for each loop is also chosen irrespective to the stride of the other loops. So it is possible to find the parameters such that the performance in CORE is kept at a high level. Needless to say, the other tuning techniques than blocking such as loop unrolling, latency hiding and software pipelining are required to improve the performance in CORE to a higher level.

2.3 How to decide parameters

Parameters of multiple blocking (the depth of loop unrolling, the block size of FRAME, the order of stripminings and their strides) are automatically decided by benchmark. Throughout the benchmark, in order to keep the continuous memory access as much as possible, the order of the innermost loops with indices `i,j,k` is fixed at `jki` for $C_w = A_w B_w$, while fixed at `jik` for $C_w = A_w^T B_w$, as shown in Fig. 3.

The benchmark runs according to the following prescription;

1. Select the depth of loop unrolling for the innermost triple loops from the set $S_{LU} = \{1, 2, 3, 4, 8\}$. The number of all combinations is 5^3, from which the best five patterns are selected. At this stage, multiple blocking is not taken into account yet.

2. For each unrolling pattern determined at the stage 1, select the block size `nb2` of FRAME from the set $S_{nb2} = \{32, 48, 64, 96, 128, 256\}$. Here the block size `nb2` is common for the triple loops in FRAME. The number of all combinations is 6×5 at this stage, from which the best four patterns are selected. As at the stage 1, multiple blocking is not taken into account yet at this stage.

3. For each selected at the stage 2, decide the order of stripmining, namely the order of the outer triple loops in CORE. Note that "No Stripminig" is a possible choice in this case. In case that stripmining is applied, the stride is common to all the triple loops and is fixed to `nb1i` = `nb1j` = `nb1k` = 16. The number of the combinations is 16 for stripmining. We select the best four patterns from all combinations (16×4).

4. For each candidates selected at the stage 3, decide the strides `nb1i,nb1j,nb1k` for stripmining from the set $S_{nb1} = \{0, 16, 24, 32, 40, 48, 56\}$, where 0 means "No Stripmining". The number of all combinations is at most $7^3 \times 4$, from which we select the best one.

3 Experimental results

We show machine specifications in Table 1 and compiler options for each machine in Table 2, respectively. We evaluate the performance of multiple/conventional blocking by matrix multiplication $C = \alpha AB + \beta C$. In the experiment, A, B and C are taken as dense square matrices of size $n = 512$, and $\alpha = \beta = 1.0$.

Table 3 shows the results of the performance evaluation on each machine. In Table 3, the column "Unrolling" shows the depth of loop unrolling for `i/j/k`

```
do JB=1,nb2,nb1j
  do KB=1,nb2,nb1k
    do j=JB,JB+nb1j-1
      do k=KB,KB+nb1k-1
        tmp=Bw(k,j)
        do i=1,nb2
           Cw(i,j)=Cw(i,j)+Aw(i,k)*tmp
        end do
      end do
    end do
  end do
end do
```

(a) $C_w = A_w B_w + C_w$

```
do IB=1,nb2,nb1i
  do j=1,nb2
    do i=IB,IB+nb1i-1
      tmp=0
      do k=1,nb2
        tmp=tmp+Aw(k,i)*Bw(k,j)
      end do
      Cw(i,j)=Cw(i,j)+tmp
    end do
  end do
end do
```

(b) $C_w = A_w^T B_w + C_w$

Figure 3: Core loop in CORE for matrix multiplication

core loops in CORE, the order of which is also shown in this column (outer loop - middle loop - inner loop). The column "nb2" shows the block size of a core loop in FRAME. The column "DV" shows the order of core loop stripmining in CORE. In this column "Lxx" indicates that the stride of loop stripmining is xx for L-loop, where L = M or O, and M and O mean the middle and outer loops shown in the column "Unrolling", respectively. The indication "none" in this column corresponds to the case for the conventional single blocking without loop stripmining. The columns of "Speed-Up" and "Performance rel. to Peak" show the performance of multiple blocking, relative to the performance of the conventional single blocking and the peak performance of each machine, respectively.

As seen from Table 3, multiple blocking is indeed efficient for all systems, except AXP18W, for type $C_w = A_w B_w$. In particular, the performance improvement reaches 11.2% for AXP18L. The main reason for speed up by multiple blocking is that, for type $C_w = A_w B_w$, stripmining for the loop with the index **k** in Fig. 3(a) substantially decreases the L1 cache misses for matrix A_w. On the other hand, for type $C_w = A_w^T B_w$, a remarkable performance improvement is not observed. However, we guess that there is much space to be improved for determining the optimal parameters for transposed matrices. In the present method, we take a common block size **nb2** for all indices in FRAME. As a result, the shape of block is square. For type $C_w = A_w^T B_w$, vertical rectangle is preferred to avoid cache misses as much as possible and is expected to bring about the performance improvement.

4 Summary

In this paper, we have proposed a multiple blocking method which makes it possible to utilize multi-layer caches efficiently. The efficiency of the proposed method has been confirmed by the performance evaluation for matrix multiplication. In the best case for the type $C_w = A_w B_w$, the multiple blocking brings about the performance improvement by more than 10%. On the other hand, in case for the type $C_w = A_w^T B_w$, a remarkable change is not observed by using multi-blocking. However, the performance is expected to be improved by extending the parameter space in the benchmark. In particular, it might be crucial for better performance to take into account the vertical rectangle shape in FRAME. The research in this direction is now going on.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney and D. Sorensen, "LAPACK Users' Guide (3rd Ed.)," SIAM, 2000.

[2] J. J. Dongarra, I. S. Duff, D. C. Sorensen and H. A. van der Vorst, "Numerical Linear Algebra for High-Performance Computers," SIAM, 1998.

[3] J. J. Dongarra, J. DuCroz, I. S. Duff and S. Hammarling, " A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Software*, Vol. 16, pp.1-17, 1990.

Table 1: Machine specifications

Abbr. Name	Full Name	Clock (MHz)	Theoretical Peak(Mflops)	L1 Cache (KB)		L2 Cache (KB)	OS
				Inst.	Data		
SPARC	Sun UltraSPARC-I	167	334	16	16	128	Solaris 2.6
ULTRA2	Sun Ultra 2	296	592	16	16	2048	Solaris 2.6
PII333	Intel Pentium II	333	333	16	16	512	Windows95
AXP18L	AMD AthlonXP1800+	1530	3060	64	64	256	Linux
AXP18W							Windows Me

Table 2: Compiler options

System	Compiler	Options
SPRAC	SPARCompiler FORTRAN77 4.0	-fast -xO5 -xtarget=ultra2 -xarch=v8plusa -xcache=16/32/1:2048/64/1-unroll=1 -pad
ULTRA2	SPARCompiler FORTRAN77 4.0	-fast -xO5 -xtarget=ultra -xarch=v8plusa -unroll=1 -pad
PII333	gcc 2.95.3 & g77 2.95.3	-O3 -ffast-math -fomit-frame-pointer -m486
AXP18L	gcc 2.95.3 & g77 2.95.3	-O3 -ffast-math -fomit-frame-pointer -m486
AXP18W	gcc 2.95.3 & g77 2.95.3	-O3 -ffast-math -fomit-frame-pointer -m486

Table 3: Performance of matrix multiplication

System	CORE Type ($A_w B_w$ / $A_w^T B_w$)	Unrolling (Out-Mid.-In)	nb2	DV	Performance (Mflops)	Speed-Up (%)	Performance rel. to Peak(%)
SPARC	$A_w B_w$	j1-k4-i1	96	M24	161.419593	105.5	48.3
				none	153.038389	—	45.8
	$A_w^T B_w$	j2-i4-k1	128	M56	233.467814	99.3	69.9
				none	235.180880	—	70.4
ULTRA2	$A_w B_w$	j1-k8-i1	96	M24	307.641610	110.9	51.9
				none	277.298183	—	46.8
	$A_w^T B_w$	j4-i2-k1	128	M24	366.209814	98.2	61.8
				none	373.016639	—	63.0
PII333	$A_w B_w$	j3-k4-i8	96	M16	136.261653	105.6	40.9
				none	129.055508	—	38.8
	$A_w^T B_w$	j1-i3-k4	32	M16	136.261653	103.4	40.9
				none	132.234215	—	39.7
AXP18L	$A_w B_w$	j1-k4-i8	96	M56	1090.895060	111.2	35.6
				none	981.030516	—	32.0
	$A_w^T B_w$	j4-i1-k2	128	O16-M32	1062.862360	103.8	34.7
				none	1023.617330	—	33.4
AXP18W	$A_w B_w$	j1-k3-i4	64	M16	958.698057	96.4	31.3
				none	994.205393	—	32.4
	$A_w^T B_w$	j4-i1-k3	128	M24	994.205393	100.0	32.4
				none	994.205393	—	32.4