CS:APP2e Web Aside MEM:BLOCKING: Using Blocking to Increase Temporal Locality*

Randal E. Bryant
David R. O'Hallaron

June 5, 2012

Notice

The material in this document is supplementary material to the book Computer Systems, A Programmer's Perspective, Second Edition*, by Randal E. Bryant and David R. O'Hallaron, published by Prentice-Hall and copyrighted 2011. In this document, all references beginning with "CS:APP2e " are to this book. More information about the book is available at* `csapp.cs.cmu.edu`*.*

1 Introduction

There is an interesting technique called *blocking* that can improve the temporal locality of inner loops. The general idea of blocking is to organize the data structures in a program into large chunks called *blocks*. (In this context, "block" refers to an application-level chunk of data, *not* to a cache block.) The program is structured so that it loads a chunk into the L1 cache, does all the reads and writes that it needs to on that chunk, then discards the chunk, loads in the next chunk, and so on.

Unlike the simple loop transformations for improving spatial locality, blocking makes the code harder to read and understand. For this reason, it is best suited for optimizing compilers or frequently executed library routines. Still, the technique is interesting to study and understand because it is a general concept that can produce big performance gains on some systems.

2 A Blocked Version of Matrix Multiply

Blocking a matrix multiply routine works by partitioning the matrices into submatrices and then exploiting the mathematical fact that these submatrices can be manipulated just like scalars. For example, suppose we want to compute $C = AB$, where A, B, and C are each 8×8 matrices. Then we can partition each matrix into four 4×4 submatrices:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Figure 1 shows one version of blocked matrix multiplication, which we call the $bijk$ version. The basic idea behind this code is to partition A and C into $1 \times$ *bsize row slivers* and to partition B into *bsize* $\times$ *bsize* blocks. The innermost (j, k) loop pair multiplies a sliver of A by a block of B and accumulates the result into a sliver of C. The i loop iterates through n row slivers of A and C, using the same block in B.

Figure 2 gives a graphical interpretation of the blocked code from Figure 1. The key idea is that it loads a block of B into the cache, uses it up, and then discards it. References to A enjoy good spatial locality because each sliver is accessed with a stride of 1. There is also good temporal locality because the entire sliver is referenced $bsize$ times in succession. References to B enjoy good temporal locality because the entire $bsize \times bsize$ block is accessed n times in succession. Finally, the references to C have good spatial locality because each element of the sliver is written in succession. Notice that references to C do not have good temporal locality because each sliver is only accessed one time.

Blocking can make code harder to read, but it can also pay big performance dividends. Figure 3 shows the performance of two versions of blocked matrix multiply on a Pentium III Xeon system ($bsize = 25$). Notice that blocking improves the running time by a factor of two over the best nonblocked version, from about 20 cycles per iteration down to about 10 cycles per iteration. The other interesting thing about blocking is that the time per iteration remains nearly constant with increasing array size. For small array sizes, the additional overhead in the blocked version causes it to run slower than the nonblocked versions. There is a crossover point at about $n = 100$, after which the blocked version runs faster.

We caution that blocking matrix multiply does not improve performance on all systems. For example, on a Core i7 system, there exist unblocked versions of matrix multiply that have the same performance as the best blocked version.

code/mem/matmult/bmm.c

```
1  void bijk(array A, array B, array C, int n, int bsize)
2  {
3      int i, j, k, kk, jj;
4      double sum;
5      int en = bsize * (n/bsize); /* Amount that fits evenly into blocks */
6
7      for (i = 0; i < n; i++)
8          for (j = 0; j < n; j++)
9              C[i][j] = 0.0;
10
11     for (kk = 0; kk < en; kk += bsize) {
12         for (jj = 0; jj < en; jj += bsize) {
13             for (i = 0; i < n; i++) {
14                 for (j = jj; j < jj + bsize; j++) {
15                     sum = C[i][j];
16                     for (k = kk; k < kk + bsize; k++) {
17                         sum += A[i][k]*B[k][j];
18                     }
19                     C[i][j] = sum;
20                 }
21             }
22         }
23     }
24 }
```

code/mem/matmult/bmm.c

Figure 1: **Blocked matrix multiply.** A simple version that assumes that the array size (n) is an integral multiple of the block size (`bsize`).

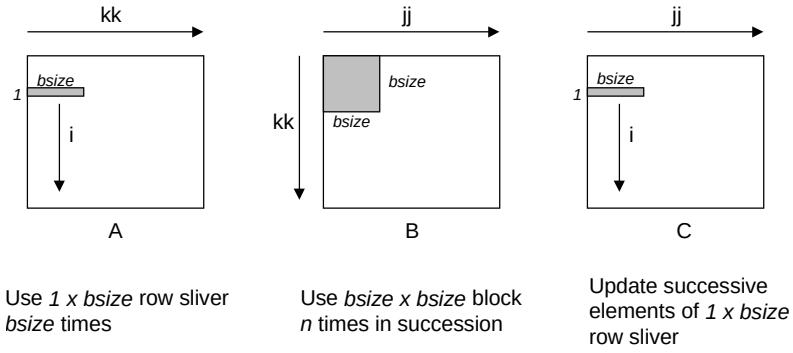

Figure 2: **Graphical interpretation of blocked matrix multiply** The innermost (j, k) loop pair multiplies a $1 \times$ *bsize* sliver of A by a *bsize* $\times$ *bsize* block of B and accumulates into a $1 \times$ *bsize* sliver of C.

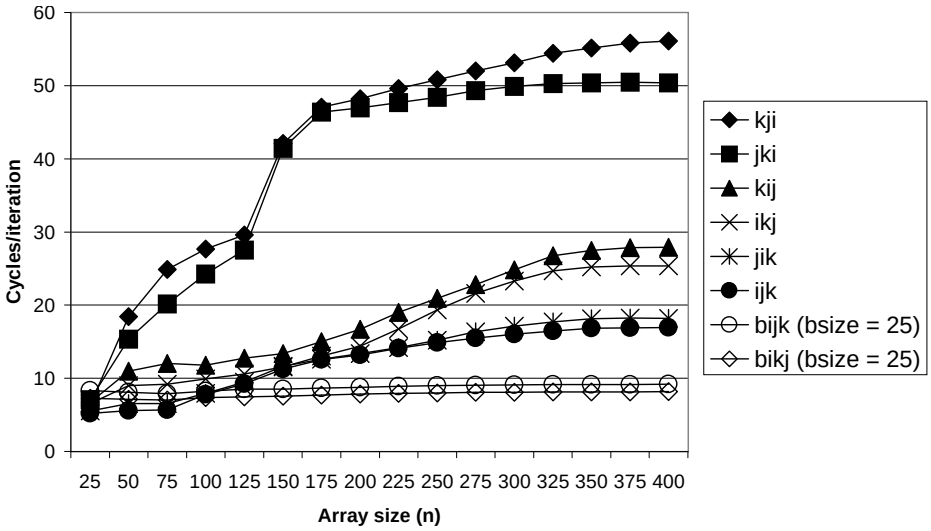

Figure 3: **Pentium III Xeon blocked matrix multiply performance.** Legend: $bijk$ and $bikj$: two different versions of blocked matrix multiply. The performance of different unblocked versions is shown for reference.