Automatic Generation of Block-Recursive Codes

Nawaaz Ahmed and Keshav Pingali

Department of Computer Science,
Cornell University, Ithaca, NY 14853

Abstract. Block-recursive codes for dense numerical linear algebra computations appear to be well-suited for execution on machines with deep memory hierarchies because they are effectively blocked for all levels of the hierarchy. In this paper, we describe compiler technology to translate iterative versions of a number of numerical kernels into block-recursive form. We also study the cache behavior and performance of these compiler generated block-recursive codes.

1 Introduction

Locality of reference is important for achieving good performance on modern computers with multiple levels of memory hierarchy. Traditionally, compilers have attempted to enhance locality of reference by *tiling* loop-nests for each level of the hierarchy [4, 10, 5]. In the dense numerical linear algebra community, there is growing interest in the use of block-recursive versions of numerical kernels such as matrix multiply and Cholesky factorization to address the same problem. Block-recursive algorithms partition the original problem recursively into problems with smaller working sets until a base problem size whose working set fits into the highest level of the memory hierarchy is reached. This recursion has the effect of blocking the data at many different levels at the same time. Experiments by Gustavson [8] and others have shown that these algorithms can perform better than tiled versions of these codes.

To understand the idea behind block-recursive algorithms, consider the iterative version of Cholesky factorization shown in Figure 1. It factorizes a symmetric positive definite matrix `A` into the product $A = L \cdot L^T$ where L is a lower triangular matrix, overwriting A with L. A block-recursive version of the algorithm can be obtained by sub-dividing the arrays A and L into 2×2 blocks, as shown in Figure 2. Here, $chol(X)$ computes the Cholesky factorization of array X. The recursive version factorizes the A_{00} block, performs a division on the A_{10} block, and finally factorizes the updated A_{11} block. The termination condition for the recursion can be either a single element of A (in which case a square root operation is performed) or a $b \times b$ block of A which is factored by the iterative code.

[0] This work was supported by NSF grants CCR-9720211, EIA-9726388, ACI-9870687, EIA-9972853.

```
    for j = 1, n
        for k = 1, j-1
            for i = j, n
S1:             A(i,j) -= A(i,k) * A(j,k)
S2:     A(j,j) = dsqrt(A(j,j))
        for i = j+1, n
S3:         A(i,j) = A(i,j) / A(j,j)
```

Fig. 1. Cholesky Factorization

$$
\begin{aligned}
\begin{bmatrix} A_{00} & A_{10}^T \\ A_{10} & A_{11} \end{bmatrix} &= \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix} \begin{bmatrix} L_{00}^T & L_{10}^T \\ 0 & L_{11}^T \end{bmatrix} \\
&= \begin{bmatrix} L_{00}L_{00}^T & L_{00}L_{10}^T \\ L_{10}L_{00}^T & L_{10}L_{10}^T + L_{11}L_{11}^T \end{bmatrix} \\
L_{00} &= chol(A_{00}) \\
L_{10} &= A_{10}L_{00}^{-T} \\
L_{11} &= chol(A_{11} - L_{10}L_{10}^T)
\end{aligned}
$$

Fig. 2. Block recursive Cholesky

A block-recursive version of matrix multiplication $C = AB$ can also be derived in a similar manner. Subdividing the arrays into 2×2 blocks results in the following block matrix formulation —

$$
\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{bmatrix}
$$

Each matrix multiplication results in eight recursive matrix multiplications on sub-blocks. The natural order of traversal of a space in this recursive manner is called a *block-recursive* order and is shown in Figure 4 for a two-dimensional space. Since there are no dependences, the eight recursive calls to matrix multiplication can be performed in any order. Another way of ordering these calls is to make sure that one of the operands is reused between adjacent calls[1]. One such ordering corresponds to traversing the sub-blocks in the *gray-code* order. A gray-code order on the set of numbers $(1 \ldots m)$ arranges the numbers so that adjacent numbers differ by exactly 1 bit in their binary representation. A gray-code order of traversing a 2-dimensional space is shown in Figure 5. Such an order is called *space-filling* since the order traces a complete path through all the points, always moving from one point to an adjacent point. There are other space-filling orders; some of them are described in the references [6]. Note that lexicographic order, shown in Figure 3, is not a space-filling order.

In this paper, we describe compiler technology that can automatically convert iterative versions of array programs into their recursive versions. In these programs, arrays are referenced by affine functions of the loop-index variables. As a result, partitioning the iterations of a loop will result in the partitioning of data as well. We use affine mapping functions to map all the statement instances of the program to a space we call the *program iteration space*. This mapping effectively converts the program into a perfectly-nested loop-nest in which all statements are nested in the innermost loop. We develop legality conditions under which the iteration space can be recursively divided. Code is then generated to traverse the space in a block-recursive or space-filling manner, and when each point in this space is visited, the statements mapped to it are executed. This strategy effectively converts the iterative versions of codes into their recursive

[1] Not more than one can be reused, in any case.

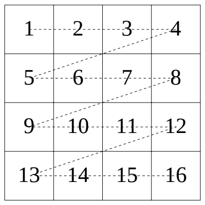

Fig. 3. Lexicographic

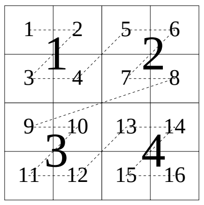

Fig. 4. Block Recursive

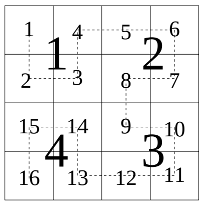

Fig. 5. Space-Filling

ones. The mapping functions that enable this conversion can be automatically derived when they exist. This approach does not require that the original program be in any specific form – any sequence of perfectly or imperfectly nested loops can be transformed in this way.

The rest of this paper is organized as follows. Section 2 gives an overview of our approach to program transformation (details are in [2]); in particular, in Section 2.1, we derive legality conditions for recursively traversing the program iteration space. Section 3 describes code generation, and Section 4 presents experimental results. Finally, Section 5 describes future work.

2 The Program-Space Formulation

A program consists of statements contained within loops. All loop bounds and array access functions are assumed to be affine functions of surrounding loop indices. We will use S_1, S_2, $\ldots$, S_n to name the statements of the program in syntactic order.

A *dynamic instance* of a statement S_k refers to a particular execution of that statement for a given value of index variables i_k of the loops surrounding it, and is represented by $S_k(i_k)$. The execution order of these instances can be represented by a *statement iteration space* of $|i_k|$ dimensions, where each dynamic instance $S_k(i_k)$ is mapped to the point i_k. For the iterative Cholesky code shown in Figure 1, the statement iteration spaces for the three statements `S1`, `S2` and `S3` are $j_1 \times k_1 \times i_1$, j_2, and $j_3 \times i_3$ respectively. The program execution order of a code fragment can be modeled in a similar manner by a *program iteration space*, defined as follows.

1. Let $\mathcal{P}$ be the Cartesian product of the individual statement iteration spaces of the statements in that program. The order in which this product is formed is the syntactic order in which the statements appear in the program. If p is the sum of the number of dimensions in all statement iteration spaces, then $\mathcal{P}$ is p-dimensional. $\mathcal{P}$ is also called the *product space* of the program.
2. Embed all statement iteration spaces $\mathcal{S}_k$ into $\mathcal{P}$ using embedding functions $\tilde{F}_k$ which satisfy the following constraints:
   (a) Each $\tilde{F}_k$ must be one-to-one.

(b) If the points in space $\mathcal{P}$ are traversed in lexicographic order, and all statement instances mapped to a point are executed in original program order when that point is visited, the program execution order is reproduced.

The program execution order can thus be modeled by the pair $(\mathcal{P}, \tilde{\mathcal{F}} = \{\tilde{F_1}, \tilde{F_2}, \ldots, \tilde{F_n}\})$. We will refer to the program execution order as the *original* execution order. For the Cholesky example, the program iteration space is a 6-dimensional space $\mathcal{P} = j_1 \times k_1 \times i_1 \times j_2 \times j_3 \times i_3$. One possible set of embedding functions $\tilde{\mathcal{F}}$ for this code is shown below –

$$\tilde{F_1}(\begin{bmatrix} j_1 \\ k_1 \\ i_1 \end{bmatrix}) = \begin{bmatrix} j_1 \\ k_1 \\ i_1 \\ j_1 \\ j_1 \\ i_1 \end{bmatrix} \quad \tilde{F_2}(\begin{bmatrix} j_2 \end{bmatrix}) = \begin{bmatrix} j_2 \\ j_2 \\ j_2 \\ j_2 \\ j_2 \\ j_2 \end{bmatrix} \quad \tilde{F_3}(\begin{bmatrix} j_3 \\ i_3 \end{bmatrix}) = \begin{bmatrix} j_3 \\ j_3 \\ i_3 \\ j_3 \\ j_3 \\ i_3 \end{bmatrix}$$

Note that all the six dimensions are not necessary for our Cholesky example. Examining the mappings shows us that the last three dimensions are redundant and the program iteration space could as well be 3-dimensional. For simplicity, we will drop the redundant dimensions when discussing the Cholesky example. The redundant dimensions can be eliminated in a systematic manner by retaining only those dimensions whose mappings are linearly independent.

In a similar manner, any other execution order of the program can be represented by an appropriate pair $(\mathcal{P}, \mathcal{F})$. Code for executing the program in this new order can be generated as follows. We traverse the entire product space lexicographically, and at each point of $\mathcal{P}$ we execute the original program with all statements protected by guards. These guards ensure that only statement instances mapped to the current point are executed. For the Cholesky example, naive code which implements the execution order $(\mathcal{P}, \tilde{\mathcal{F}})$[2] is shown in Figure 6. This naive code can be optimized by using standard polyhedral techniques [9] to remove the redundant loops and to find the bounds of loops which are not redundant. An optimized version of the code is shown in Figure 7. The conditionals in the innermost loop can be removed by index-set splitting the outer loops.

2.1 Traversing the Program Iteration Space

Not all execution orders $(\mathcal{P}, \mathcal{F})$ respect the semantics of the original program. A legal execution order must respect the *dependences* present in the original program. A dependence is said to exist from instance i_s of statement S_s to instance i_d of statement S_d if both statement instances reference the same array location, at least one of them writes to that location, and instance i_s occurs before instance i_d in original execution order. Since we traverse the product space lexicographically, we require that the vector $v = F_d(i_d) - F_s(i_s)$ be lexicographically positive for every pair (i_s, i_d) between which a dependence exists. We refer to v as the *difference vector*.

[2] We have dropped the last three redundant dimensions for clarity.

```
for j1 = -inf to +inf
for k1 = -inf to +inf
for i1 = -inf to +inf
  for j = 1, n
    for k = 1, j-1
      for i = j, n
        if (j1==j && k1==k && i1==i)
S1:       A(i,j) -= A(i,k) * A(j,k)

    if (j1==j && k1==j && i1==j)
S2:   A(j,j) = dsqrt(A(j,j))

    for i = j+1, n
      if (j1==j && k1==j && i1==i)
S3:     A(i,j) = A(i,j) / A(j,j)
```

Fig. 6. Naive code for Cholesky

```
for j1 = 1,n
for k1 = 1,j1
for i1 = j1,n
    if (k1 < j1)
S1:     A(i1,j1) -= A(i1,k1) * A(j1,k1)

    if (k1==j1 && i1==j1)
S2:     A(j,j) = dsqrt(A(j1,j1))

    if (k1==j1 && i1 > j1)
S3:     A(i1,j1) = A(i1,1j) / A(j1,j1)
```

Fig. 7. Optimized code for Cholesky

For a given embedding $\mathcal{F}$, there may be many legal traversal orders of the product space other than lexicographic order. The following traversal orders are important in practice.

1. Any order of walking the product space represented by a unimodular transformation matrix T is legal if $T \cdot v$ is lexicographically positive for every difference vector v associated with the code.
2. If the entries of all difference vectors corresponding to a set of dimensions of the product space are **non-negative**, then those dimensions can be blocked. This partitions the product space into blocks with planes parallel to the axes of the dimensions. These blocks are visited in lexicographic order. This order of traversal for a two-dimensional product space divided into equal-sized blocks is shown in Figure 3.
   When a particular block is visited, all points within that block can be visited in lexicographic order. Other possibilities exist. Any set of dimensions that can be blocked can be *recursively* blocked. If we choose to block the program iteration space by bisecting blocks recursively, we obtain the block-recursive order shown in Figure 4.
3. If the entries of all difference vectors corresponding to a dimension of the product space are **zero**, then that dimension can be traversed in any order. If a set of dimensions exhibit this property, then those dimensions can not only be blocked, but the blocks themselves do not have to be visited in a lexicographic order. In particular, these blocks can be traversed in a *space-filling* order. This principle can be applied recursively within each block, to obtain space-filling orders of traversing the entire sub-space (Figure 5).

Given an execution order $(\mathcal{P}, \mathcal{F})$, and the dependences in the program, it is easy to check if the difference vectors exhibit the above properties using standard dependence analysis [11]. If we limit our embedding functions $\mathcal{F}$ to be affine functions of the loop-index variables and symbolic constants, we can determine functions that allow us to block dimensions (and hence also recursively block them) or to traverse a set of dimensions in a space-filling order. The condition that entries corresponding to a particular dimension of all difference vectors must be non-negative (for recursive-blocking) or zero (for space-filling orders) can be converted into a system of linear inequalities on the unknown coefficients of $\mathcal{F}$ by an application of *Farkas' Lemma* as discussed in [2]. If this system has solutions, then any solution satisfying the linear inequalities would give the required embedding functions. The embedding functions for the Cholesky example were determined by this technology.

3 Code Generation

Consider an execution order of a program represented by the pair $(\mathcal{P}, \mathcal{F})$. We wish to block the program iteration space recursively, terminating when blocks of size $B \times B \ldots \times B$ are reached.

Let p represent the number of dimensions in the product space. To keep the presentation simple, we assume that redundant dimensions have been removed and that all dimensions can be blocked. We also assume that all points in the program iteration space that have statement instances mapped to them are positive and that they are contained in the bounding box $(1 \cdots B \times 2^{k_1}, \ldots, 1 \cdots B \times 2^{k_p})$.

Code to traverse the product space recursively is shown in Figure 8. The parameter to the procedure `Recurse` is the current block to be traversed, its coordinates given by `(lb[1]:ub[1], ..., lb[p]:ub[p])`. The function `HasPoints` prevents the code from recursing into blocks that have no statement instances mapped to them. If there are points in the block and the block is not a base block, `GenerateRecursiveCalls` subdivides the block into 2^p sub-blocks by bisecting each dimension and calls `Recurse` recursively in a lexicographic order[3]. The parameter `q` of the procedure `GenerateRecursiveCalls` specifies the dimension to be bisected. On the other hand, if the parameter to `Recurse` is a base block, code for that block of the iteration space is executed in procedure `BaseBlockCode`.

For the initial call to `Recurse`, the lower and upper bounds are set to the bounding box.

Naive code for `BaseBlockCode(lb,ub)` is similar to the naive code for executing the entire program. Instead of traversing the entire product space, we only need to traverse the points in the current block lexicographically, and execute statement instances mapped to them. Redundant loops and conditionals can be hoisted out by employing polyhedral techniques.

Blocks which contain points with statement instances mapped to them can be identified by creating a linear system of inequalities with variables lb_i, ub_i corresponding to each entry of `lb[1..p]`, `ub[1..p]` and variables x_i corresponding to each dimension of the product-space. Constraints are added to ensure that the point $(x_1, x_2, \ldots, x_p)$ has a statement instance mapped to it and that it lies within the block $(lb_1 : ub_1, \ldots, lb_p : ub_p)$. From the above system, we obtain the condition to be tested in `HasPoints(lb,ub)` by projecting out (in the Fourier-Motzkin sense) the variables x_i.

[3] This must be changed appropriately if space-filling orders are required

```
Recurse(lb[1..p], ub[1..p])
if (HasPoints(lb,ub)) then
   if (∀i ub[i] == lb[i]+B-1) then
      BaseBlockCode(lb)
   else
      GenerateRecursiveCalls(lb,ub,1)
   endif
endif
end


GenerateRecursiveCalls(lb[1..p], ub[1..p], q)
if (q > p)
   Recurse(lb, ub)
else
  for i = 1,p
     lb'[i] = lb[i]
     ub'[i] = (i==q) ? (lb[i]+ub[i])/2
                     : ub[i]
  GenerateRecursiveCalls(lb',ub',q+1)

  for i = 1, p
     lb'[i] = (i==q) ? (lb[i]+ub[i])/2 + 1
                     : lb[i]
     ub'[i] = ub[i]
  GenerateRecursiveCalls(lb',ub',q+1)
endif
end
```

Fig. 8. Recursive code generation

```
BaseBlockCode(lb[1..3])
for j1 = lb[1], lb[1]+B-1
for k1 = lb[2], lb[2]+B-1
for i1 = lb[3], lb[3]+B-1
  for j = 1, n
    for k = 1, j-1
      for i = j, n
        if (j1==j && k1==k && i1==i)
S1:        A(i,j) -= A(i,k) * A(j,k)

    if (j1==j && k1==j && i1==j)
S2:   A(j,j) = dsqrt(A(j,j))

    for i = j+1, n
      if (j1==j && k1==j && i1==i)
S3:      A(i,j) = A(i,j) / A(j,j)
end


HasPoints(lb[1..3], ub[1..3])
if (lb[1]<=n && lb[2]<=n && lb[3]<=n
    && lb[1]<=ub[3]
    && lb[2]<=ub[1]
    && lb[2]<=ub[3])
    return true
else
    return false
end
```

Fig. 9. Recursive code for Cholesky

For our Cholesky example, the embedding functions shown in Section 2 enable all dimensions to be blocked. Since there are difference vectors with non-zero entries, the program iteration space cannot be walked in a space-filling manner, though it can be recursively blocked. The portion of the product-space that has statement instances mapped to it is $[j,k,i] : 1 \leq k \leq j \leq i \leq n$. This is used to obtain the condition in `HasPoints()`. Naive code for executing the code in each block is shown in Figure 9. As mentioned earlier, the redundant loops must be removed and the conditionals hoisted out for good performance.

4 Experimental Results

In this section, we discuss the performance of block-recursive and space-filling codes produced using the technology described in this paper. All experiments were run on an SGI R12K machine running at 300Mhz with a 32Kb primary-data cache (L1), 2Mb second-level cache (L2) and 64 TLB entries.

The legality conditions discussed in Section 2.1 permit us to conclude matrix multiply (MMM) can be blocked both recursively and in a space-filling manner. The Cholesky code can only be blocked recursively. We generated four versions of block-recursive code with different base block sizes (16, 32, 64, 128) for both programs. These codes were compiled with the "-O3 -LNO:blocking=off" option of the SGI compiler. At this level of optimization, the SGI compiler performs tiling for registers and software-pipelining.

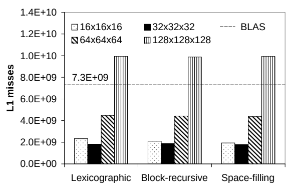

Fig. 10. MMM : L1 cache

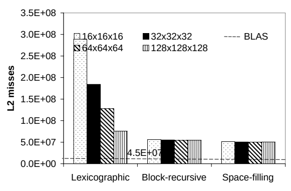

Fig. 11. MMM : L2 cache

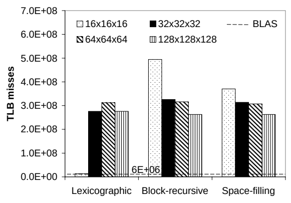

Fig. 12. MMM : TLB

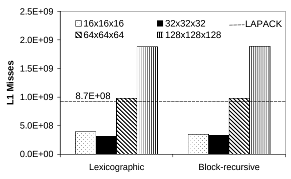

Fig. 13. CHOL : L1 cache

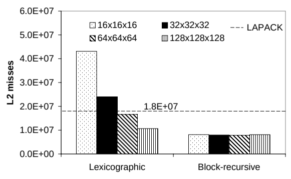

Fig. 14. CHOL : L2 cache

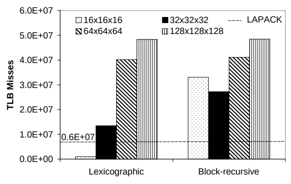

Fig. 15. CHOL : TLB

For each program, we ran the recursive (and if legal, the space-filling) versions of the code for a variety of matrix sizes. For lack of space, we only present results for a matrix size of 4000×4000. Results for other matrix sizes are similar. In the graphs, the results marked `Lexicographic` correspond to executing the code in `BaseBlockCode(lb)` by visiting the base blocks in a lexicographic order. For comparison, we also show results of executing vendor-supplied, hand-tuned implementations of matrix multiply (`BLAS`) and Cholesky (`LAPACK` [3]).

Figures 10 and 13 show the number of L1 data cache misses for the two programs. For the larger block sizes (64, 128), the data touched by a base-block does not fit in cache (32K) and hence both the recursive and lexicographic versions suffer the same penalty. For smaller block sizes (16, 32), the data does fit into cache resulting in much fewer misses. Figures 11 and 14 show the L2 cache misses. The lexicographic versions for block sizes of 16 and 32 exhibit much higher miss numbers than the corresponding recursive versions since these block sizes are too small to fully utilize the 2M cache. In the recursive versions, however, even the small block sizes succeed in full utilization of the cache due to the recursive doubling effect. These recursive versions will have a similar effect on any further levels of caches. Of the two recursive orders, the space-filling orders show slightly better cache performance for both programs.

Figures 12 and 15 show the number of TLB misses for the two programs. The R12K TLB has only 64 entries, hence large block sizes (more than 64) will exhibit high miss rates in both the lexicographic and recursive cases. Small block sizes could work well in the lexicographic case if the loop order is chosen well. In our case, the `jki` order is the best order for both the programs. There are very few TLB misses when the block size is 16 because fewer than 48 TLB entries are required at a time for this block size. In the recursive case, the recursive doubling does cause significantly more TLB misses for small block sizes, although the recursive walks are largely immune to the effect of reordering the loops. By

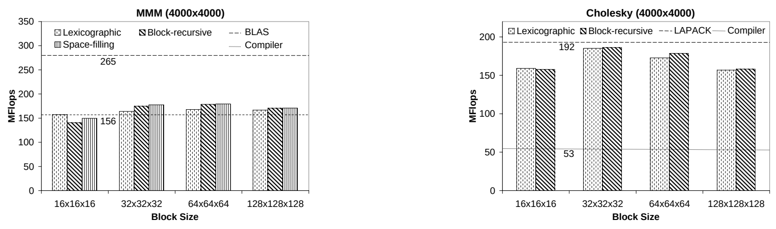

Fig. 16. Performance

comparison, in the `jik`-order (not shown here), the code with block size of 16 suffers a 100-fold increase in the number of TLB misses for the lexicographic case but the number of misses remains roughly the same in the recursive cases.

Figure 16 shows the performance of the two programs in MFlops. As a sanity check, the lines marked `Compiler` show the performance obtained with compiling the original code with the "-O3" flag of the SGI compiler which attempts to tile for cache and registers and then software-pipeline the resulting code. For both programs, the recursive codes with block size of 32 are the best among all the generated code. For most block sizes, the recursive codes are better than their lexicographic counter-parts by a small percentage (2-5%). For a block size of 16, the recursive cases are worse due to an increase in the number of TLB misses.

For matrix multiply, the best recursive code generated by the compiler is still substantially worse than the hand-tuned versions of the programs even though the recursive overhead is less than 1% in all cases. This difference could be due to the high number of TLB misses suffered by the recursive versions. Copying data from base blocks into contiguous locations as is done in the hand-tuned code might help improve performance. It is interesting to note that although the hand-tuned version suffers higher primary cache miss rates, the impact on performance is small. This is not surprising in an out-of-order issue processor like the R12K where the latency of primary cache misses (10 cycles) can be hidden by scheduling and software-pipelining. These misses will be more important in an in-order issue processor like the Merced. For Cholesky factorization, on the other hand, the best block-recursive version is comparable in performance to LAPACK code.

5 Related Work and Conclusions

Hand-coded versions of block recursive algorithms have been studied for a long time [1, 7, 6]; some of them are implemented in the IBM's Engineering and Scientific Subroutine Library (ESSL) for example.

In this paper, we developed program restructuring technology to convert iterative numerical programs into block-recursive versions. Our experiments show that the block-recursive versions of matrix multiply and Cholesky are effectively blocked for all memory hierarchy levels. However, base block sizes must be chosen with care – the data accessed in a base-block must fit into the lowest level of the cache hierarchy, the blocks must be large enough so that the recursive overhead is negligible, and the back-end compiler must be able to schedule the instructions in a base-block efficiently. Unfortunately, our experiments also show that the block-recursive algorithms do not interact well with the TLB. In spite of this, the best compiler-generated code for the two applications was nevertheless a recursive version. We conjecture that better interaction with the TLB requires either (i) copying data from column-major order into recursive data layouts as suggested by Chatterjee [6] or (ii) copying the data used by a base block into contiguous locations as suggested by Gustavson [1].

The work in this paper can be extended in a number of ways. More experiments are needed to assess the importance of block-recursive codes for other applications such as relaxation methods. Non-square base-blocks may be useful to eliminate conflict misses in some codes. Finally, it would be interesting to study the effect of copying data into layouts that are matched to block-recursive traversals.

References

1. Ramesh C. Agarwal, Fred G. Gustavson, Joan McComb, and Stanley Schmidt. Engineering and Scientific Subroutine Library Release 3 for IBM ES/3090 Vector Multiprocessors. *IBM Systems Journal*, 28(2):345–350, 1989.
2. Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proc. International Conference on Supercomputing*, Santa Fe, New Mexico, May 2000.
3. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, editors. *LAPACK Users' Guide. Second Edition.* SIAM, Philadelphia, 1995.
4. Steve Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing*, 1992.
5. L. Carter, J. Ferrante, and S. Flynn Hummel. Hierarchical tiling for improved superscalar performance. In *International Parallel Processing Symposium*, April 1995.
6. S. Chatterjee, V. Jain, A. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *International Conference on Supercomputing (ICS'99)*, June 1999.
7. Matteo Frigo, C.L.Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science.* IEEE Press, 1999.
8. F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.
9. Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, February 1995.
10. Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Programming Languages, Design and Implementation.* ACM SIGPLAN, June 1997.
11. William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Communications of the ACM*, pages 102–114, August 1992.