# Efficient Matrix Multiplication Using Cache Conscious Data Layouts

Neungsoo Park[*], Wenheng Liu[†], Viktor K. Prasanna[*], Cauligi Raghavendra
Department of Electrical Engineering–Systems
University of Southern California
Los Angeles, CA 90089-2562
{neungsoo, prasanna, raghu}@halcyon.usc.edu

**Abstract**

This paper demonstrates performance improvements for matrix multiplication and mesh generation for Finite Element Method (FEM) by optimizing the memory hierarchy of traditional processors. The theory developed earlier is used to perform such optimizations. Our work provides a uniform methodology across multiple HPC platforms for optimizing the performance of the kernel codes (such as matrix transposition and matrix multiplication) commonly used in DoD applications.

For "standard" matrix multiplication, we develop a novel data layout that reduces cache pollution, data cache misses and Translation Look-Aside Buffer (TLB) misses. Prior to computation, we reorganize the matrix data layout by transposing and partitioning it into blocks. The proposed method avoids cache pollution, conflict cache misses, and TLB misses.

We also employ our techniques to improve memory system performance in order to perform mesh generation for a suite of FEMs. We apply an algorithmic technique of separating rows of a data matrix based on their access patterns. Applied in conjunction with blocking, our approach minimizes memory requirements and optimizes cache performance.

We have implemented our proposed approach on UltraSPARC II, Alpha 21264, and Pentium III based machines. Experimental results show our approach to be effective in improving the overall performance.

## 1 Introduction

HPC platforms are being employed for a wide variety of applications including scientific computations. In such applications, the data is stored in different levels of the memory hierarchy with different data access costs. Although HPC platforms are already able to provide large computational power, the memory access costs have not improved commensurately. Thus memory access can become the bottleneck in achieving the full potential of HPC platforms for such applications.

In this paper, we focus on algorithmic techniques for efficient memory access. By applying the theory developed in a related project[1], we show the effectiveness of our approach by demonstrating improved

[*]Work funded wholly by the DoD High Performance Modernization Program ERDC Major Shared Resource Center through Programming Environment and Training (PET) supported by Contract Number DAHC 94-96-C0002, and Subcontract Number NRC-CR-98-0002.

[†]Current address: Department of Electrical and Computer Engineering, California State University, Los Angles, email: cliu@calstatela.edu

performance for a kernel operation, matrix multiplication, and the mesh generation operation for generic finite element methods on state-of-the-art machines. The efficiency of such operations depends heavily on the data access cost. Traditionally, matrices are stored in either column or row major order (data layout) in the memory. Mismatch between the data layout and the data access pattern can lead to severe performance degradation due to cache misses, cache pollution, and TLB misses. In this paper, an algorithmic technique for cache conscious data layout is proposed for tuning the performance of matrix operations. Our proposed approach provides a uniform methodology across multiple HPC platforms for performance optimization of a suite of kernel codes (e.g., matrix transposition[8] and matrix multiplication) used by many DoD applications.

A novel data layout is proposed to reduce cache pollution and data cache and TLB misses in performing the "standard" matrix multiplication. To perform matrix multiplication $C = A \times B$, the elements in Matrix $B$ are accessed in column major order. If Matrix $B$ is stored in a row-major order, in performing the inner product, each access to $B$ will result in a cache miss since they belong to different cache blocks. In a large-scale matrix multiplication, only small portions of such cache blocks are accessed before they get replaced due to conflicts. Blocking (tiling) is widely used to reorder the computation sequence to reduce cache pollution. This technique exploits temporal locality to reduce cache misses. However, cache conflict misses still occur in tiled matrix multiplication. Also, for large matrices, elements in a column in the same block will be located in several different pages. Thus, column accesses cause TLB thrashing since the size of TLB is usually small. Therefore, blocking alone is not sufficient to alleviate all the memory access penalties.

We reorganize the layout of matrix data stored in the main memory such that it is cache friendly. This reorganization is performed prior to the computation. In the proposed approach, we transpose matrix $B$ so that the data layout matches the data access pattern. This reduces cache pollution. Then we partition each matrix into square sub-matrices, which are denoted as blocks. In the proposed data layout, matrix elements belonging to the same block are stored in consecutive memory locations in row major order. This avoids conflict misses among the elements in the same block. The block size is chosen to be equal to the virtual page size. This ensures that computations within a block do not result in a TLB miss. Therefore, the total number of TLB misses is reduced significantly with our layout compared to the standard row major layout. The above data layout transformation (i.e., matrix transpose and block data layout) is performed only once. Thus, no additional overhead is incurred during the computation. Our proposed layout reduces cache pollution, cache misses, and TLB misses without excessive overheads in reorganizing the data in memory as well as in index computations. Experimental results on UltraSPARC II, Alpha 21264, and Pentium III based machines show improved performance.

Our algorithmic technique for cache conscious block data layout has other applications. With this approach, we develop a data layout for the mesh generation for an FEM using serendipity elements. FEM is widely used for analysis and simulation of a large variety of scientific problems. The first step of a typical finite element method is to generate an appropriate mesh for the problem domain. In grand challenge applications, a typical FEM has 10,000 to 1,000,000 elements. Such applications require large number of memory access. In our technique, rows of a data matrix are clustered based on their access patterns. This technique can avoid the cache pollution problem. Then, blocking and block layout are used to further reduce cache misses and TLB misses. This data layout results in efficient data access and minimization of the memory space allocated for the corresponding mesh.

The remainder of this paper is organized as follows. In Section 2, we present our techniques for matrix multiplication. In Section 3, we discuss our research on the use of efficient data layouts for FEM applications. In Section 4, we give our concluding remarks on this paper.

## 2 Matrix Multiplication

In this section, we describe cache conscious data layouts for efficient matrix multiplication. For standard matrix multiplication, we show how to reduce cache pollution, data cache misses, and TLB misses. We start with a description of data access costs of the memory hierarchy and issues relating to data layout design.

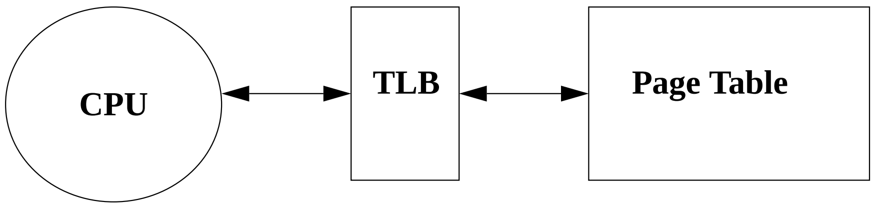


Figure 1: Using TLB for fast page number translation

Then we present an efficient data layout for the standard matrix multiplication operation. As seen from the experimental results obtained from three different architectures, our approach compares favorably with the three others that we have considered.

## 2.1 Data Access in the Memory Hierarchy

In state-of-the-art computer systems, a processor communicates with its cache memory (fast memory), which in turn communicates with the main memory. If the data requested by the processor is not available in the cache, a cache miss occurs. This cache miss causes processor to stall until data is fetched from the main memory into the cache. Typically, a main memory access is about 10 times slower than a cache memory access. For example, on a DEC Alpha 21154 platform, the on-chip cache access latency is around 10 nsec, while the main memory access latency is 253 nsec. Therefore, cache misses cause the performance to degrade. To improve performance, it is critical to reduce the number of cache misses during a computation.

To fetch data from memory to cache, the virtual address of data needs to be translated to physical address. Physical memory is organized into equal sized pages (say, 4k bytes). All addresses within a physical memory page lie in consecutive memory locations. The virtual memory is also organized into pages of the same size. Each virtual page is assigned a page number. When the main memory runs out of space, one page is brought at a time from the disk. Before the processor can execute a memory access instruction, the virtual address has to be translated. Translation determines the physical page number corresponding to a given virtual page number. This mapping information is provided by the page tables which are the data structures stored in the main memory.

It is very expensive to look up the page table for each address. In order to reduce the cost of translation, a special high-speed cache is provided to buffer the page table entries. This cache is called the TLB. In Figure 1, the CPU looks up the TLB for each address translation. Usually, the TLB size is limited to only a few entries. For each TLB miss, the missing translation entry is loaded into it from the page table. A TLB miss significantly increases the translation time.

## 2.2 Issues in Design of Data Layout

A data layout is the scheme in which data elements are assigned addresses in the memory. In a row major layout (see Figure 2), elements in one row are assigned consecutive memory locations. In a column major layout, elements in one column are assigned consecutive memory locations.

Mismatch between data access patterns and data layout patterns can increase the number of cache misses. Consider the example shown in Figure 3, where a cache with one 4 word sized block has data laid out in row major order. A row major access pattern causes only 4 cache misses since each cache miss loads the entire row into a block. In the case of column major access, the number of cache misses is as high as 16. This happens because each cache miss loads one useful element and three unused elements.

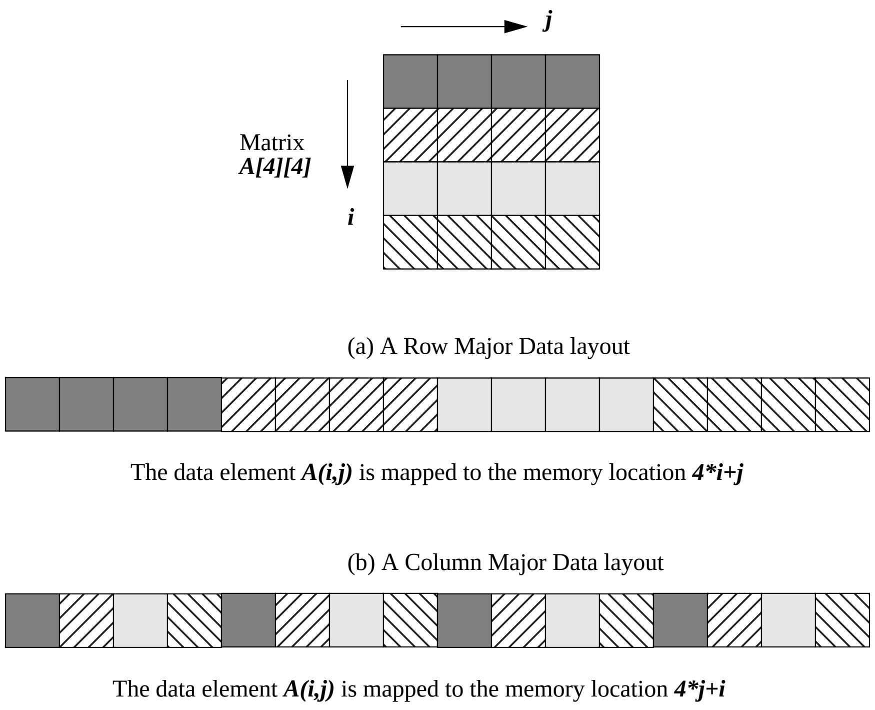

j
Matrix
A[4][4]
i
(a) A Row Major Data layout
The data element A(i,j) is mapped to the memory location 4*i+j
(b) A Column Major Data layout
The data element A(i,j) is mapped to the memory location 4*j+i

Figure 2: Example data layouts

## 2.3 Data Layout for Standard Matrix Multiplication

Large scale matrix multiplication deals with matrices which do not fit into the cache. Assume that both $A$ and $B$ are stored in row major order. Conventional methods of matrix multiplication are not cache-friendly. In the matrix multiplication $C = A \times B$, elements in Matrix $B$ are accessed in column major order. Each access to $B$ results in a cache miss since the consecutively accessed elements are located far apart in memory. Elements of $B$ are repeatedly accessed when computing different elements of $C$, but they do not remain in the cache for reuse as the cache capacity is small. Besides, only small portions of the fetched cache blocks are accessed before they get replaced due to conflicts. The net result is a large number of cache misses.

Blocking (tiling) is widely used to reduce the size of the working set in order to fit it into the cache. Blocking exploits temporal locality to reduce cache misses. However, cache conflict misses still occur in tiled matrix multiplication. TLB thrashing is another problem which occurs in computations with large matrices. For large matrices, elements in a column in the same block will be located in several different pages. Thus, column accesses may cause a spate of TLB misses because TLB can keep only small number of entries. Therefore, blocking does not offer a complete solution for cache-optimized matrix multiplication.

To improve the performance of blocking, the copying technique [3] is combined with blocking in order to remove the conflict misses in blocking computation. Before blocking computation, the tiled data are copied into temporary storage like a buffer. Therefore, during the computation, the self-interference in block data can be removed. However, the copying incurs large overhead and causes performance degradation. Also, it cannot solve the TLB miss problem discussed above.

One of the key ideas of our approach is to reorganize the layout of matrix data stored in the main memory such that it is cache friendly. We perform this reorganization prior to computation. In the proposed data layout, we transpose matrix B such that the data layout matches the data access pattern. This reduces cache pollution to a considerable extent. We partition each matrix into square sub-matrices, which are denoted

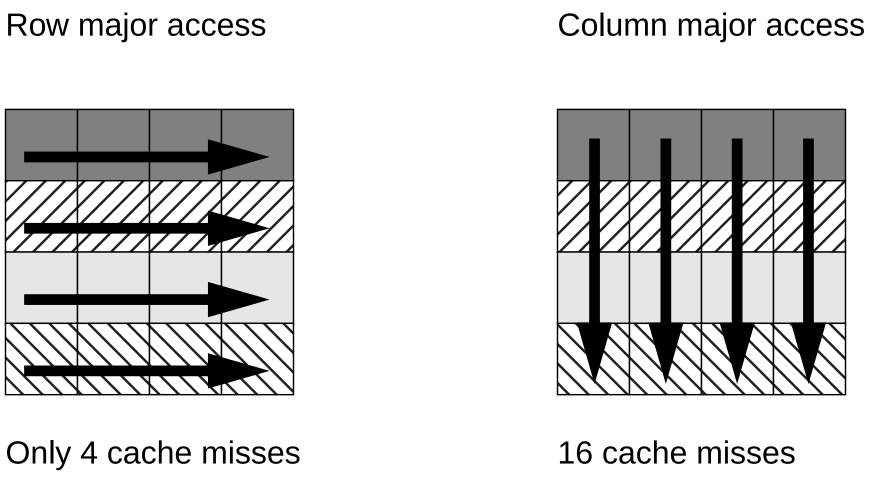


Figure 3: Access of page numbers using translation look-aside buffer

as blocks. Elements belonging to the same block are stored in consecutive memory locations in row major order. The resulting data layout is as shown in Figure 4 (b). There are no conflict misses among the elements in the same block. The block size is chosen to be equal to the virtual page size so that computations within a block do not result in a TLB miss. Therefore, the total number of TLB misses is reduced significantly with our layout compared to the standard row major layout. The above data layout transformation (i.e., matrix transpose and block data layout) is performed only once. Thus, no additional overhead is incurred during the computation. Our proposed layout reduces cache pollution, cache misses, and TLB misses without excessive overheads like reorganizing the data in memory or computing indices.

### 2.4 Experimental Results

We have implemented our scheme on UltraSPARC II, Alpha 21264, and Pentium III based machines for matrix sizes ranging from 1024x1024 to 1536x1536. Tables 1, 2, and 3 compare the performance of our scheme with the naive CBLAS (without blocking), CBLAS (with blocking), and CBLAS (with blocking and copying) algorithms. All the experiments were conducted on the above machines. The reported execution times are wall clock times. On UltraSPARC II, Alpha 21264, and Pentium III machines, we used the gcc compiler with "-O3" optimization option. As the experimental results show, our scheme is up to 15 times faster than naive CBLAS, 2 times faster than blocking based CBLAS, and is superior to blocking and copying based CBLAS implementations on UltraSPARC II. On Alpha 21264, our scheme performs up to 5 times faster than the naive CBLAS, up to 3 times faster than blocking based CBLAS, and is faster than blocking and copying based CBLAS implementation. On Pentium III, our scheme again outperforms the other three algorithms.

## 3 Finite Element Method

In this section, we present an efficient data layout for mesh generation for generic two-dimensional FEMs. FEMs are widely used for numerical analysis and simulation of a large variety of scientific problems. The first step of a typical FEM is to generate an appropriate mesh of the problem domain. FEMs for grand challenge applications can have 10,000 to 1,000,000 elements. Often, very large number of memory access is required. We describe a data layout of the mesh generated for an FEM using serendipity elements. This

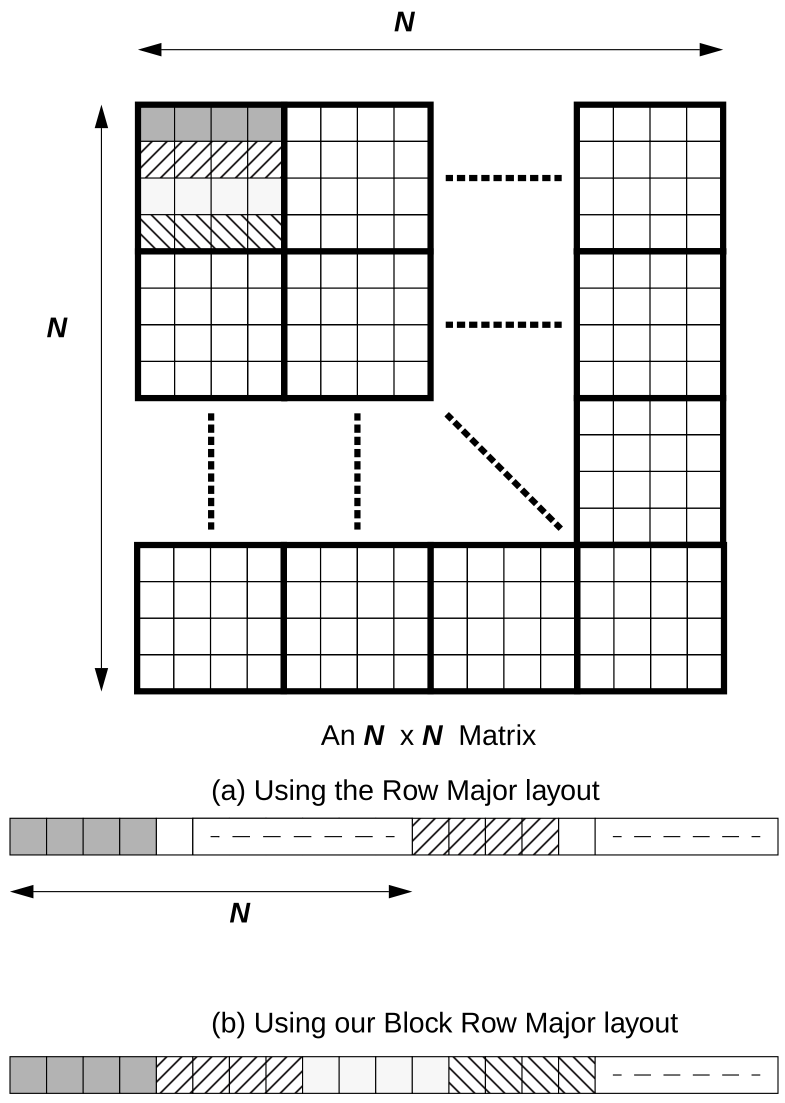

N
N
An N x N Matrix
(a) Using the Row Major layout
N
(b) Using our Block Row Major layout

Figure 4: Illustration of our proposed block data layout for matrix multiplication

Table 1: Execution time on UltraSPARC II (400 MHz, 2MByte L2 cache)

| Matrix size | CBLAS (Native) | CBLAS (Blocking) | CBLAS (Blocking +copying) | Our Algorithm |
|---|---|---|---|---|
| $1024 \times 1024$ | 243.418 | 34.147 | 22.271 | 17.240 |
| $1200 \times 1200$ | 370.387 | 40.663 | 34.478 | 29.920 |
| $1280 \times 1280$ | 455.795 | 65.952 | 42.262 | 33.842 |
| $1400 \times 1400$ | 592.934 | 66.675 | 53.522 | 45.192 |
| $1536 \times 1536$ | 810.280 | 124.489 | 74.740 | 60.865 |

Table 2: Execution time on DEC Alpha 21264 (500MHz, 4MByte L2 cache)

| Matrix size | CBLAS (Native) | CBLAS (Blocking) | CBLAS (Blocking +copying) | Our Algorithm |
|---|---|---|---|---|
| $1024 \times 1024$ | 125.237 | 23.214 | 16.427 | 13.283 |
| $1200 \times 1200$ | 194.556 | 28.330 | 28.465 | 22.383 |
| $1280 \times 1280$ | 238.613 | 31.115 | 29.984 | 26.503 |
| $1400 \times 1400$ | 310.947 | 44.730 | 45.311 | 35.248 |
| $1536 \times 1536$ | 415.870 | 79.907 | 54.045 | 45.816 |

layout results in efficient data access and minimization of the memory space allocated for the corresponding mesh.

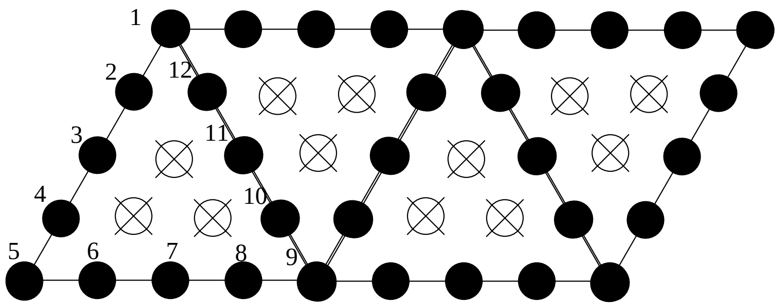
### 3.1 Mesh Generation for two-Dimensional FEMs

In finite element methods, the problem domain is first discretized into a collection of pre-selected finite elements. Based on discretization, a finite-element mesh of pre-selected elements is generated. The mesh generation method has a strong influence on the quality of the numerical results. In two-dimensional FEMs, the problem domain is usually discretized into triangular or rectangular elements. Some examples are shown in Figure 5.

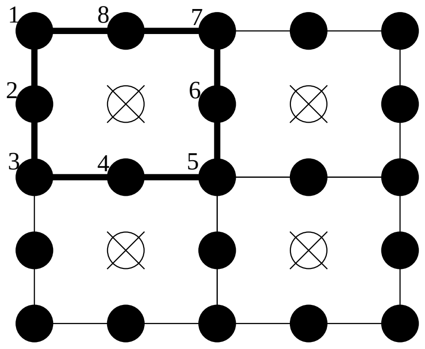
Different types of elements have different number of pre-selected nodes. After discretization, the geometric properties (e.g., coordinates, cross-section areas, etc.) of each node are generated. Based on these properties, element equations are derived either directly or iteratively. For applications using the iterative approach, the mesh may need to be regenerated for each iteration based on changes in geometric properties.

### 3.2 Efficient Data Layout for FEMs using Serendipity Elements

In mesh generation for two-dimensional FEMs, internal nodes of the higher-order triangular or rectangular elements can be condensed at the element level. Condensation is justified as these nodes do not contribute to the inter-element connectivity. Figure 5 (a) shows a mesh with triangular elements, each having 12 boundary nodes and 3 internal nodes. Another mesh with rectangular elements is shown in Figure 5 (b). Here each element has 8 boundary nodes and 1 internal node. We can use the serendipity elements to avoid internal nodes. Serendipity elements are those triangular or rectangular elements which have no interior nodes [6]. Techniques to condense internal nodes can reduce the size of the element matrices, which in turn are derived from nodal properties. A mesh with serendipity elements can be realized by storing the geometric properties of all nodes in row-major order. However, this straight-forward approach has its problems. First, it wastes memory space. For instance, the FEM shown in Figure 5 (b) uses 8-node rectangular elements, and about a quarter of the nodes are not accessed. Second, this approach causes cache pollution when the even rows of the node matrix are accessed, as only half of the nodes are required.



(a) 12-node triangular elements



(b) 8-node rectangular elements

Figure 5: Illustration of two FEM meshes using serendipity elements and their node numbering schemes

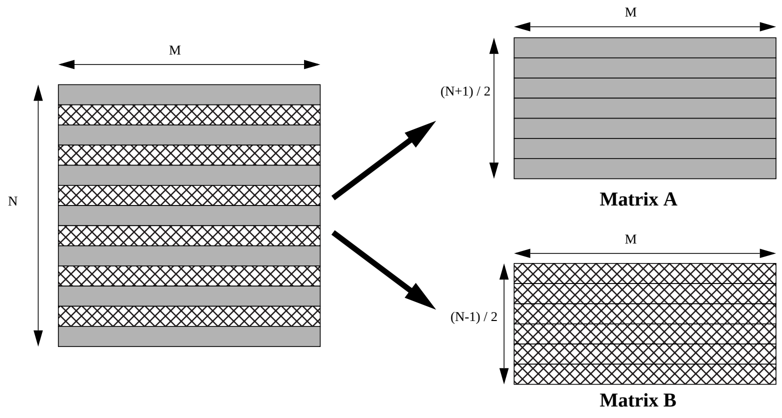


Figure 6: Separation of the odd rows and even rows of a mesh with 12-node rectangular elements

Table 3: Execution time on Pentium III (450MHz, 512KByte L2 cache)

| Matrix size | CBLAS (Native) | CBLAS (Blocking) | CBLAS (Blocking +copying) | Our Algorithm |
|---|---|---|---|---|
| $1024 \times 1024$ | 92.566 | 27.136 | 22.030 | 17.335 |
| $1200 \times 1200$ | 152.311 | 30.107 | 33.390 | 28.050 |
| $1280 \times 1280$ | 184.973 | 52.325 | 43.117 | 34.184 |
| $1400 \times 1400$ | 244.652 | 48.215 | 55.644 | 45.756 |
| $1536 \times 1536$ | 325.241 | 90.306 | 74.345 | 59.137 |

In our work, we have focussed on developing a data layout that addresses the above problems effectively. The goals are to minimize the memory requirements, avoid cache conflict misses, and prevent cache pollution. We suggest an approach in which rows are stored with the same access pattern as a sub-matrix. Various sub-matrices require different indexing schemes based on their corresponding access patterns. Using the technique of separating rows with different access patterns, only the required nodes of each row are stored. In Figure 5 (b), the FEM requires all nodes in the odd rows, and only every other node in the even rows. Alternate rows from the original matrix are stored in matrices $A$ and $B$ as shown in Figure 6. As depicted in Figure 7 , we condense matrix $B$ so that only the nodes that have to be accessed are stored. Thus, memory requirements are minimized.

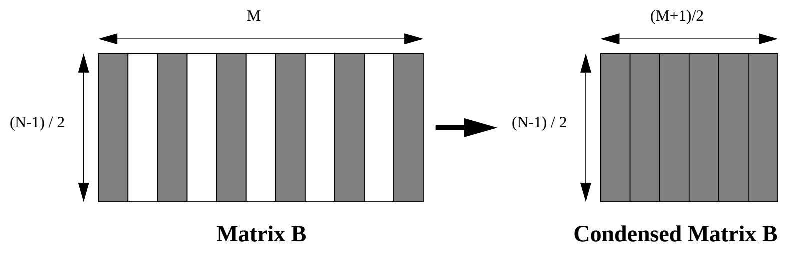


Figure 7: Matrix B is condensed to reduce the overall memory requirement

Table 4: Sub-matrices used for the 12-node triangular FEM

| Sub-matrix Name | Sub-matrix Content | matrix size | access pattern |
|---|---|---|---|
| $A$ | Rows $4 \times k + 1$ | $(\frac{n}{4})m$ | all nodes |
| $B$ | Rows $4 \times k + 2$ | $(\frac{n}{4})(\frac{m}{2})$ | 1,2,5,6... |
| $C$ | Rows $4 \times k + 3$ | $(\frac{n}{4})(\frac{m}{2})$ | 1,3,5,7... |
| $D$ | Rows $4 \times k + 4$ | $(\frac{n}{4})(\frac{m}{2})$ | 1,4,5,8... |

Using our data layout, both the sub-matrices are needed to access the nodes of a single element. For instance, in Figure 5 (b), Matrix A is used to access nodes 1, 3, 4, 5, 7, and 8, while Matrix B is used to access nodes 2 and 6.

Similarly, in the other example shown in Figure 5 (a), the FEM requires all of the nodes in rows 1, 5, 9, ..., $4 \times k + 1$ ($k$ is a non-negative integer). However, it only requires nodes 1,2,5,6,... in rows 2, 6, 10,..., $4 \times k + 2$. Using our approach, we alternately store the rows in four sub-matrices. Table 4 summarizes the sizes and the access patterns of these sub-matrices. Note that we assume an $n \times m$ node matrix, where $n$, $m$ are both an integer multiple of 4. Also, we assume that $k$ is a non-negative integer number.

In addition to the proposed technique to separate the node matrix into sub-matrices, we further apply the block data layout as described in the previous section. The use of block data layout can further reduce the cache conflict misses as explained in Section 2.

### 3.3 Experimental Results

We have implemented an FEM benchmark using the 8-node rectangular model to examine the efficiency of memory access on an UltraSPARC II machine. The matrix in our experiment contains 512 nodes. Table 5 compares the performance of three approaches: a straightforward (without blocking) approach, an approach with blocking, and a scheme that combines separate sub-matrices, blocking, and block data layout. The reported execution times are wall clock times. On UltraSPARC II, we used the gcc compiler with the "-O3" optimization option. For implementing our proposed scheme, we used block layouts and in-line functions to access each array element. The block size was chosen to be 32 in order to avoid TLB misses.

Table 5: Memory Access time on UltraSPARC II (400 MHz, 2MByte L2 cache)

| FEM Memory Access Benchmark | Memory Access Time (milli-seconds) | Techniques applied |
|---|---|---|
| Straightforward Approach | 95.313 | None |
| Blocking-Based Approach | 74.190 | Blocking |
| Our Proposed Approach | 43.718 | Separate Sub-matrices<br>Blocking + Block Layout |

In spite of the greater complexity in index computation, our scheme is about 2.2 times faster than the straightforward approach and approximately 1.7 times faster than the blocking based approach, as evident from the experimental results shown in Table 5.

## 4 Conclusions

In this paper, we have proposed techniques for data placement to support efficient memory access for large scale scientific applications.

For large-scale matrix transposition (out-of-core matrix transpose), our algorithm reduced both the number of I/O operations and the index computation time. Our results show that our algorithm reduces the execution time by up to 50%.

For large-scale standard matrix multiplication, our approach using the block data layout has shown significant performance improvement. Our approach reduces cache pollution, conflict cache misses, and TLB misses. Our scheme is up to 15 times faster than naive CBLAS, 2 times faster than blocking based CBLAS, and 22% faster than blocking and copying based CBLAS, which were implemented on UltraSPARC II.

For mesh generation of Finite element methods (FEMs), our approach avoids cache pollution. Using blocking and block layout, we achieved further reductions in cache misses and TLB misses. Experimental results indicate speedup by a factor of 2.2 over normal layout and by 1.7 over blocking with normal layout.

Our results encourage the use of data placement/data layout techniques to improve the memory access for scientific applications. We believe that the data layout designs can be further employed in other ERDC applications with complex data access patterns. The work reported here is adapted from the theory developed in the ADVISOR project(`http://advisor.usc.edu`)[1]. It is applied to two kernel problems in large scale scientific computing which are of interest to HPCMO. In the ADVISOR project, algorithmic technique for advanced architectures (such as smart memory and PIM) with a particular emphasis on irregular and streaming applications is being developed.

## 5 Acknowledgment

We would like to thank Mr. Bhaskar Srinivasan for his assistance in preparing the paper.

## References

[1] ADVISOR project, `http://advisor.usc.edu`.

[2] A. Choudhary, W. K. Liao, P. Varshney, D. Weiner, R. Linderman and M. Linderman "Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers," 12th International Parallel Processing Symposium, Orlando, Florida, 1998.

[3] M. S. Lam and E. E. Rothberg and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," ASPLOS IV, April 1991.

[4] D. E. Dudgen and R. M. Mersereau, Multidimensional Signal Processing, Prentice-Hall, 1984.

[5] V. Kumar, A. Grama, A. Gupta, and G. Karypis, Introduction to Parallel Computing, The Benjamin/Cummings Publishing Company, Inc., 1994.

[6] J. N. Reddy, "An Introduction to the Finite Element Method," published by McGraw-Hill Inc., 1984.

[7] J. Suh and V. K. Prasanna, "Portable Implementation of Real Time Signal Processing Benchmarks on HPC Platforms," International Workshop on Applied Parallel Computing in Large Scale Scientific and Industrial Problems '98, Umea, Sweden, June 1998.

[8] J. Suh, S. Narayanan and V. K. Prasanna, "An Efficient Algorithm for Large-Scale Matrix Transposition," International Conference on Parallel Processing, Aug. 2000.