Multithreaded Transposition of Square Matrices with Common Code for Intel Xeon Processors and Intel Xeon Phi Coprocessors

Andrey Vladimirov
for Colfax Research

August 12, 2013

Abstract

In-place matrix transposition, a standard operation in linear algebra, is a memory bandwidth-bound operation. The theoretical maximum performance of transposition is the memory copy bandwidth. However, due to non-contiguous memory access in the transposition operation, practical performance is usually lower. The ratio of the transposition rate to the memory copy bandwidth is a measure of the transposition algorithm efficiency.

This paper demonstrates and discusses an efficient C language implementation of parallel in-place square matrix transposition. For large matrices, it achieves a transposition rate of 49 GB/s (82% efficiency) on Intel Xeon CPUs and 113 GB/s (67% efficiency) on Intel Xeon Phi coprocessors. The code is tuned with pragma-based compiler hints and compiler arguments. Thread parallelism in the code is handled by OpenMP, and vectorization is automatically implemented by the Intel compiler. This approach allows to use the same C code for a CPU and for a MIC architecture executable, both demonstrating high efficiency. For benchmarks, an Intel Xeon Phi 7110P coprocessor is used.

Contents

1 Why Matrix Transposition is Difficult **2**

2 Implementation **4**
2.1 Praire Schooner Speed: How *not* to Do Matrix Transposition 4
2.2 Team Penning: Regularizing the Vectorization Pattern 5
2.3 Square Dance: Matrix Traversal Algorithm 7
2.4 Flying Gallop to RAM: Prefetching and Non-temporal Stores 9
2.5 Mustang Taming: Thread Affinity 10
2.6 "The Good, the Bad and the Ugly" Matrix Sizes 10

3 Benchmarks **11**
3.1 Transposition Rate 11
3.2 Harware System Configuration 11
3.3 STREAM Benchmark 11
3.4 Compilation and Execution 12
3.5 Results: "Good" Matrix Sizes 14
3.6 Results: "Good" versus "Bad" and "Ugly" Matrix Sizes 15
3.7 Comparison with Intel MKL 16

4 Discussion **17**

A Source code **18**

1 Why Matrix Transposition is Difficult

In computer memory, two-dimensional arrays are typically laid out in row-major or column-major format. With row-major layout, adjacent elements within each row are contiguous in memory, and one row follows another. Traversing a row-major matrix along columns is much slower than traversing it along rows, because in the latter case, memory is accessed with a stride equal to the row length. Therefore, for algorithms that must traverse the matrix data in the sub-optimal direction, it may be cheaper to transpose the data and modify the algorithm to access memory in the more efficient row-wise direction.

Transposition looks simple on paper: this operation swaps matrix rows with columns, as shown in Equation 1. However, when transposition is performed in computer memory, the order in which matrix elements are swapped tremendously impacts performance. The theoretical maximum performance of shared-memory matrix transposition is equal to the memory copy bandwidth, because the matrix data must be read once and written once to a different location. Yet, an architecture-unaware algorithm may perform several times to an order of magnitude slower than that.

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & \dots & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}, \qquad A^T = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n2} \\ & \dots & & \\ a_{1n} & a_{2n} & \dots & a_{nn} \end{pmatrix}. \tag{1}$$

Most computer architectures, including the Intel MIC architecture, achieve high memory bandwidth when data in memory are accessed contiguously. However, whether the matrix is laid out in a row-major or column-major format, in the course of transposition, non-contiguous memory accesses must occur somewhere. For instance, for the row-major format, if the algorithm reads the original matrix along a row, then the read access to memory is contiguous, but when it writes the transposed data, it must write into a column, where adjacent elements are separated in memory by the stride equal to the length of the matrix row. Conversely, if the algorithm writes contiguously (along a row), it must read with a stride (along a column). Figure 1 illustrates that difficulty.

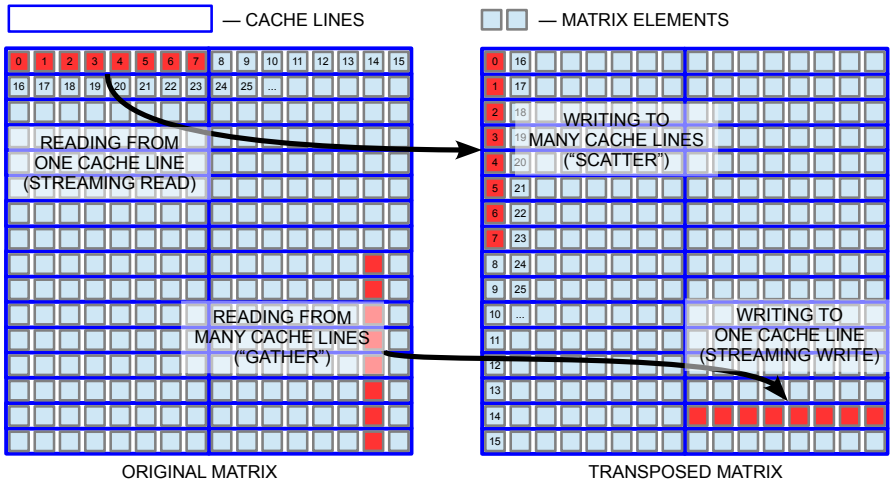

Figure 1: Transposition of a square matrix in a row-major layout on a computer architecture where a cache line fits 8 matrix elements. If the algorithm performs streaming read, it must do scattered write, and vice-versa.

The need for contiguous memory access is rooted in the memory architecture. In order to achieve the best bandwidth, memory in most computer architectures, including Intel Xeon Phi coprocessors, is accessed with a granularity of a cache line. When a core reads or writes a single data element in memory, the memory controller transfers a whole *cache line* containing that element from RAM to cache, or to the core. Cache lines in Intel Xeon CPUs and Xeon Phi coprocessors are 64 bytes long. In other words, after a double-precision (8 bytes long) element a_{11} of a matrix is fetched from memory, it costs nothing to access a_{12}

through a_{18}; however, accessing a_{19} or a_{21} will require an additional memory access[1]. As Figure 1 shows, the nature of matrix transposition causes the algorithm to either read from one and write to many cache lines, or vice-versa. In addition to this, the virtual memory in Linux is mapped into virtual memory pages, and when too many large-stride accesses occur, the lookup of the page mapping may incur additional delays.

In order to overcome the difficulty with data access being non-local in memory space, scattered memory accesses must be issued close to each other in time. This way, even though the algorithm will scatter to, or gather from multiple cache lines, it will be done while these cache lines are still in the core vector registers or in a high-level cache. As a consequence, the data will be shuffled in fast memory, and the main memory access can be streamlined. Such optimization of data locality in time can be achieved with the help of loop tiling (also known as loop blocking) or recursive divide-and-conquer (also known as a cache-oblivious algorithm). These approaches have been developed by multiple authors (e.g., [1], [2], [3]); see also [4] for a summary.

In addition to the memory traffic optimization, efficient matrix transposition requires parallelism. Utilizing all memory controllers and core-local caches in the Intel MIC architecture is possible only when multiple cores join forces. However, work must be shared between cores in such a way that

a) There is enough work to balance the load across all cores (i.e., parallelism must be sufficiently fine-grained), and

b) Only one core must access any given cache line at a time so that false sharing does not occur (i.e., parallelism must be sufficiently coarse-grained),

c) Little or no synchronization between cores is necessary.

A number of existing libraries provide efficient functions for distributed-memory transpositions, i.e., when matrix data are stored in multiple compute nodes that do not share memory (e.g., [5], [6]). While distributed-memory (also called "out-of-core") transposition is an important and, in many aspects, more difficult problem, it does not fulfill the need of applications that require efficient multithreaded transposition in shared memory. The Intel MKL library [7] has a multithreaded implementation of in-place matrix transposition and scaling (functions `mkl_?imatcopy`). While `mkl_?imatcopy` supports non-square matrices, its performance is highly tuned only for specific regimes of multithreading and matrix sizes[2]. In Section 3.7 we will see that for a broader range of matrix sizes, it is possible to perform in-place square matrix transposition up to 1.5x faster than with `mkl_?imatcopy`. Transposition on GPUs has also been studied (e.g., [8], [9]). However, efficient shared-memory transposition of large (i.e., large than cache) matrices on multi- and many-core architectures seems to be under-represented in software, which motivates the development of such an application.

In addition to the practical importance of square matrix transposition, this problem is a good educational challenge on which general methods of optimization for multi-core CPUs and many-core coprocessors can be studied. Indeed, cache traffic optimization simultaneously with instrumenting thread parallelism is typical of a general class of problems that involves the processing of multidimensional arrays.

Considering these aspects, by implementing an efficient matrix transposition algorithm for the Intel MIC architecture, I pursue two goals:

1) to produce a multi-purpose tool for matrix transposition, as this operation is universal in linear algebraic applications, and

2) to learn and share general optimization methods for memory traffic in general-purpose CPUs and MIC architecture coprocessors.

[1] This example assumes that a_{11} is located at the beginning of a cache line, and that the matrix is laid out in a row-major order.
[2] As of MKL version 11.0.5, build date 20130612.

2 Implementation

In a previous Colfax Research publication [4], I described my initial experiences with the implementation of the transposition algorithm for the Intel MIC architecture. Two methods of optimization were investigated: nested loop tiling and cache-oblivious recursion, and two parallel frameworks were tested for each method: Intel Cilk Plus and OpenMP. The results reported in [4] were quantitatively unsatisfactory, but provided a roadmap for future work.

A new, improved implementation of the in-place square matrix transposition algorithm is discussed below. This implementation corrects the issues observed in [4] and achieves a better performance for all matrix sizes. The rest of this section describes the details of this implementation and discusses the optimization process that has led to this result. The code of the new implementation is provided in Figure 18, and the full benchmark code is available at the Colfax Research web site[3].

2.1 Praire Schooner Speed: How *not* to Do Matrix Transposition

The simple transposition algorithm shown in Figure 2 is impeded by sub-optimal memory access pattern, as described in Section 1. On the host system with Intel Xeon CPUs, this algorithm achieves a transposition rate equal to 60-70% of the performance of the optimized algorithm discussed below. On an Intel Xeon Phi coprocessor, this algorithm is only 25-30% as fast as the optimized implementation that we are discussing.

Figure 2: Unoptimized parallel transposition algorithm yields unsatisfactory performance. `FTYPE` is `float` for single precision or `double` for double precision.

```
1  #pragma omp parallel for
2  for (int i = 0; i < n; i++)
3    for (int j = 0; j<i; j++) {
4      const FTYPE c = A[i*n + j];
5      A[i*n + j] = A[j*n + i];
6      A[j*n + i] = c;
7    }
```

The key to improving the transposition rate is modifying this algorithm with tiling or recursion. With this optimization, the matrix is split into many sub-matrices (tiles), and a parallel algorithm for traversing the set of tiles is chosen. When a tile is visited, it is transposed serially (by one thread) with a piece of code hereafter referred to as the transposition microkernel. This approach is illustrated in Figure 3.

Figure 3: Schematic parallel transposition algorithm with data locality improvements via tiling.

```
1  #pragma omp parallel for
2  for (int tile = 0; tile < nTiles; tile++) {
3    // Traversal of the set of tiles:
4    const int ii = // ... choose the x-location of the tile
5    const int jj = // ... choose the y-location of the tile
6
7    // Tile transposition microkernel:
8    for (int i = ii; i < ii+TILE; ii++)
9      for (int j = jj; j < jj+TILE; jj++) {
10       // ... swap A_ij with A_ji
11       const FTYPE c = A[i*n + j];
12       A[i*n + j] = A[j*n + i];
13       A[j*n + i] = c;
14     }
15 }
```

The choice of the traversal algorithm and the optimization of the transposition microkernel are discussed in this Section.

[3] http://research.colfaxinternational.com/post/2013/08/12/Trans-7110.aspx

2.2 Team Penning: Regularizing the Vectorization Pattern

In my previous publication [4], the tile traversal algorithm visited all tiles in the matrix, including the tiles that contain the main diagonal of the matrix, and the tiles at the edges of the matrix. The implementation of a single tile transposition microkernel from [4] is shown in Figure 4.

Figure 4: Sub-optimal transposition microkernel from [4] with a check for reaching the main diagonal or the matrix edge.

```
1  const int iMax = (n < ii+TILE ? n : ii+TILE);
2  for (int i = ii; i < iMax; i++) {
3    const int jMax = (i < jj+TILE ? i : jj+TILE);
4  #pragma loop count avg(TILE)
5  #pragma simd
6    for (int j = jj; j<jMax; j++) {
7      const FTYPE c = A[i*n + j];
8      A[i*n + j] = A[j*n + i];
9      A[j*n + i] = c;
10   }
11 }
```

Here `TILE` is a constant known at compile time, empirically chosen as `TILE=32`. The inner loop in `j` has a variable upper bound `jMax`. This is to control that the inner loop index `j` is never greater than the outer loop index `i`. Otherwise, the corresponding matrix element would be transposed twice. More often than not, `jMax==jj+TILE` — this is the case for all tiles except those that contain the main diagonal of the matrix. In addition, the outer loop in `i` has a variable upper bound `iMax`, which makes sure that if the tile is at the edge of the matrix, the microkernel does not access the matrix out of bounds. Again, for most tiles, `iMax==ii+TILE` — unless the tile is at the edge of the matrix.

This microkernel is general, but sub-optimal, because the iteration count of the inner loop in `j` is not known at compile time. As a consequence, the compiler must implement multiple executable codes for this loop, and produce runtime checks which take different code paths, depending on whether the runtime value of the loop count is a multiple of the SIMD vector length or not. The compiler hint "`#pragma loop count avg(TILE)`" helps somewhat by advising the compiler that the most frequent case is `jMax==jj+TILE` (for tiles that are away from the main diagonal and from the matrix edges). However, runtime checks and the choice of the execution path still have to be performed for each transposed tile.

It is possible to improve the transposition performance by modifying the microkernel to have constant loop bounds. This would regularize the vectorization pattern and enable the compiler to produce a more efficient executable code. In order to do that, the tile traversal algorithm must be modified so that the microkernel is applied only to tiles "well-behaved" tiles, i.e., tiles that do not contain the main diagonal, and are not at the matrix edges.

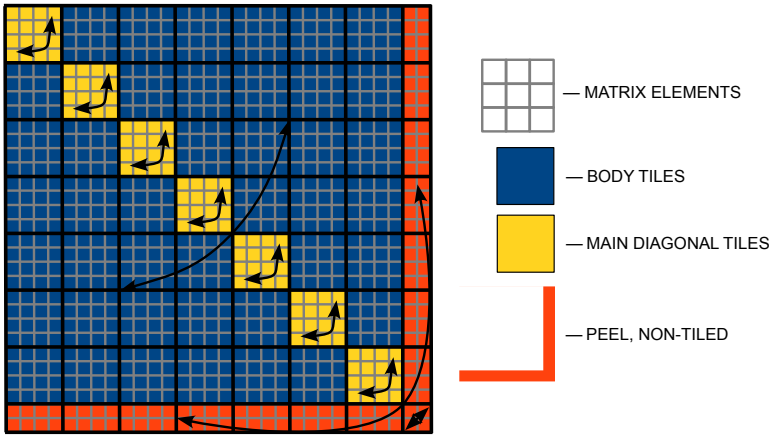

Figure 5: In order to simplify the microkernel for the transposition of a single tile, the tile traversal algorithm is modified. Matrix body tiles (shown in blue) that do not contain the main diagonal and do not touch the matrix edges are processed using an optimized microkernel (Figure 6). The rest of the tiles (the main diagonal tiles and the "peel" around the matrix edges) are processed separately.

As Figure 5 shows, in the present implementation we partition the matrix into three regions.

1. The matrix body tiles (shown in blue) comprise the bulk of the matrix data. For these tiles, a simplified transposition microkernel is used (see Figure 6, top panel), which does not have to check for proximity to the main diagonal and the matrix edges.
2. The tiles that contain the main diagonal (shown in yellow) are transposed in a separate loop, using a different microkernel (see Figure 6, bottom left panel), in which the inner vector loop in `i` has a termination condition `i>=j`. This renders the vector form of this loop less efficient, however, it is required for producing the correct result of transposition.
3. Finally, the tiles around the edges of the matrix (the "peel") are transposed in a third loop using a microkernel shown in Figure 6 bottom right panel, which performs checks for proximity to the edges to avoid out-of-bounds memory accesses. In this case, `#pragma simd` is not used, and therefore automatic vectorization is not mandatory. Due to the short length of the inner loop, vectorization may, in fact, be counter-productive.

Because the tile size does not depend on the matrix size `n`, the number of tiles in the matrix body scales as $O(n^2)$. The number of the main diagonal tiles and the "peel" tiles scales as $O(n)$. Therefore, for large `n`, the fraction of work in the body tiles asymptotically approaches 100%.

```
// Tile transposition microkernel
// for main body tiles
for (int j = jj; j < jj + TILE; j++) {
#pragma simd
  for (int i = ii; i < ii + TILE; i++) {
    const FTYPE c = A[i*n + j];
    A[i*n + j] = A[j*n + i];
    A[j*n + i] = c;
  }
}
```

```
// Tile transposition microkernel
// for tiles on the main diagonal
for (int j = jj; j < jj+TILE; j++) {
#pragma simd
  for (int i = ii; i < j; i++) {
    const FTYPE c = A[i*n + j];
    A[i*n + j] = A[j*n + i];
    A[j*n + i] = c;
  }
}
```

```
// Transposition algorithm
// for elements at the edges of the matrix
const int nEven = n - n%TILE;
for (int j = 0; j < nEven; j++) {
  for (int i = nEven; i < n; i++) {
    const FTYPE c = A[i*n + j];
    A[i*n + j] = A[j*n + i];
    A[j*n + i] = c;
  }
}
```

Figure 6: Optimized transposition microkernels. Top: for the bulk of the tiles, with constant loop bounds known at compile time. Bottom left: for tiles containing the main diagonal. Bottom right: for elements at the edges of the matrix (the "peel").

As opposed to transposing all of the tiles with one general microkernel (as done in [4]), the partitioning of the matrix into three regions and using an optimized microkernel for the "well-behaved" regions increases the amount of code and makes it less abstract. However, it improves the performance by approximately 20%.

2.3 Square Dance: Matrix Traversal Algorithm

The indexes `ii` and `jj` in the codes in Figure 6 are the column and row of the top left element of a single tile. The algorithm for choosing the values of `ii` and `jj` determines in what order the set of tiles is traversed. An essential part of this algorithm is the parallelization of the work across processor cores.

In the previous work [4], two methods of matrix traversal were implemented:

1. A method with two nested loops, which sequentially increment the tile row (outer loop) and column (inner loop). This method had the disadvantage that only the outer loop was parallelized, while the inner loop ran serially in each thread, which caused insufficient parallelism for small matrix sizes. In this case then the number of parallel work items (rows of tiles) is ≈`n/TILE`. The empirical optimal value of `TILE` is 32, and the Intel Xeon Phi coprocessor supports 240 or 244 threads (depending on the model). For matrix sizes less than 8000×8000, the value `n/TILE` is smaller than 240, so there is not enough work to occupy all threads. Furthermore, small values of `ii` have less work than large values of `ii`, so load imbalance may occur. This approach is illustrated in the left-hand side panel of Figure 7.
2. A recursive cache-oblivious divide-and-conquer method, in which the matrix is recursively split horizontally or vertically into sub-matrices, until the sub-matrix size is small enough. These smallest sub-matrices (tiles) are then transposed individually. The recursive method did not have the insufficient parallelism issue: the granularity of parallelism with recursion is a single tile, as shown in the right-hand side panel of Figure 7. This makes the number of parallel work-items (tiles) approximately equal to `(n*n)/(2*TILE)`. For `TILE=32`, and for matrices greater than 700×700, there are at least 240 work-items, which means that workload can be balanced across the 240 logical cores of the coprocessor. However, the parallel scheduling overhead of the recursive method was too high because of a large number of parallel tasks that needed to be scheduled.

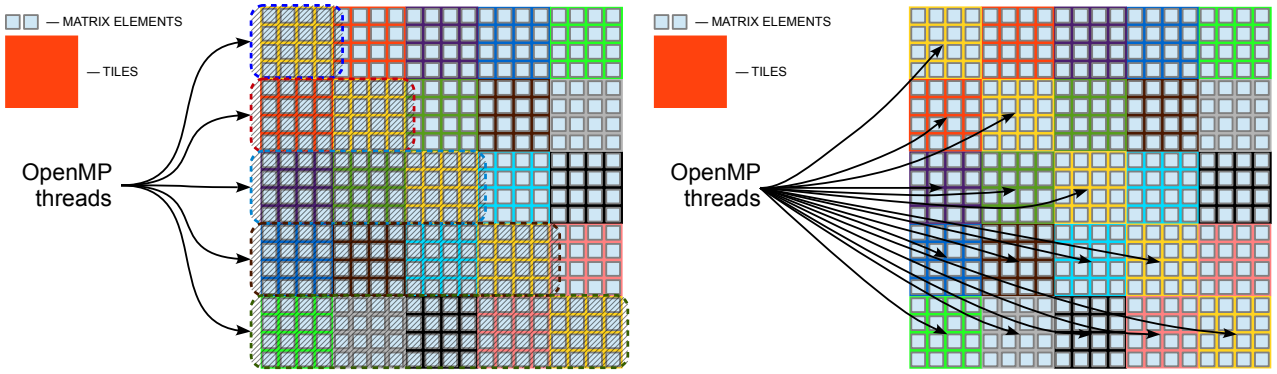

Figure 7: *Left*: matrix traversal with nested loops in previous work [4]: rows of tiles are distributed across threads (insufficient parallelism, load imbalance). *Right*: matrix traversal with the tile-size granularity of parallelism (more parallelism, better opportunities for load balancing).

In the new implementation discussed here, the nested loop and the recursive tile traversal algorithms are implemented again. However, in order to alleviate the scheduling difficulties encountered in [4], the new implementation

a) parallelizes the nested loop algorithm with a tile-size granularity, and

b) uses a planning stage to generate the order of tile traversal prior to executing the transposition, which reduces the parallel overhead.

The planning routine must be called for a given matrix size prior to calling the transposition method. This routine calculates the locations of all tiles in the main body of the matrix (blue region in Figure 5) and writes them into a pre-allocated array in the required order. Then the transposition routine reads the tile locations from this array (see, e.g., lines 5 and 6 in Figure 3). An example of the planning routine, the nested

loop traversal algorithm, is shown in Figure 8. The planning routine for the recursive algorithm is not shown due to its length. However, the complete benchmark code can be downloaded along with this PDF file from the Colfax Research web site.

```
1  void NestedLoopTranspositionPlan(int* const plan, const int n) {
2
3    // Number of complete tiles in each dimension
4    const int wTiles = n / TILE;
5    int i = 0;
6
7    // Tiled plan
8    for (int j = 1; j < wTiles; j++)
9      for (int k = 0; k < j; k++) {
10       plan[2*i + 0] = j*TILE; // Value of ii
11       plan[2*i + 1] = k*TILE; // Value of jj
12       i++;
13   }
14 }
```

Figure 8: Planning routine for the nested loop tile traversal algorithm. Only those tiles that do not touch the main diagonal or the matrix edges are traversed.

Figure 9 illustrates the nested and recursive tile traversal plans. In the nested loop algorithm (left-hand side panel), the algorithm starts in the top left corner of the main matrix body and then traverses each row of tiles from the left to the main diagonal. The chosen tile is transposed and swapped with the corresponding tile symmetrically located above the main diagonal. The recursive algorithm also starts in the top left corner, but traverses the tiles in a more complex fashion, striving to choose the next tile close in the horizontal as well as in the vertical direction to the previous tile.

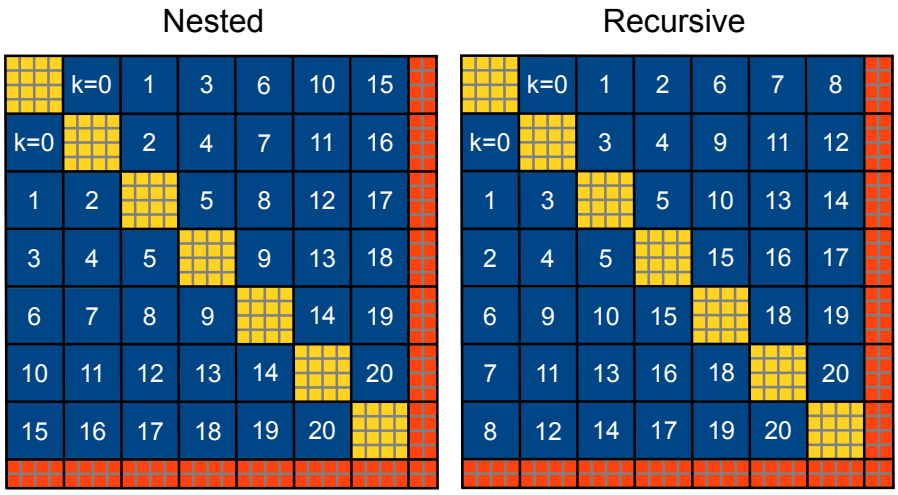

Figure 9: Left: plan of matrix transposition with the nested loop tile traversal algorithm. Right: plan with the recursive cache-oblivious method. Numbers in the tiles indicate the order in which the tiles are visited and the transposition microkernel (Figure 6, top) is applied to each of them.

Both algorithms traverse only the tiles that do not touch the main diagonal or the edges of the matrix, as justified in Section 2.2. Both algorithms strive to maintain data locality by choosing the next tile close to the previous. According to [3], the recursive algorithm has the best asymptotic cache hit ratio. However, in practice, we have found on the Intel Xeon Phi coprocessor, in many cases, the nested loop algorithm is the winner (see Section 3).

2.4 Flying Gallop to RAM: Prefetching and Non-temporal Stores

With the tile traversal algorithm chosen, and the transposition microkernel optimized, the remaining optimizations are done in the compiler arguments and environment variables. This section focuses on methods that improve the efficiency of reading and writing data from/to memory.

In order to optimize the read performance, prefetching can be used. Prefetching is a function of caches available in CPUs and in the MIC architecture, which allows to request the movement of a cache line from the RAM into a cache ahead of the time when these data are used by the core. This allows to mask the memory access latency behind calculations. Prefetch instructions can be issued by dedicated hardware units (hardware prefetching) or by the application itself (software prefetching). Intel Xeon CPUs have Level-1 and Level-2 cache hardware prefetchers, and Intel Xeon Phi coprocessors have only Level-2 cache hardware prefetchers. Therefore, for Intel Xeon Phi applications, software prefetch instructions are crucial for performance, and they are usually inserted into the executable code by the compiler automatically. However, it is possible to override the prefetch distance chosen by the compiler (i.e., how many vector loop iterations ahead a prefetch instruction is issued). We have found that using the compiler argument "`-opt-prefetch-distance=8`" improves the performance by an additional 1-2%.

In order to optimize the writing performance, the transposition routine can use non-temporal stores. With non-temporal stores, written data are flushed to RAM, bypassing caches. This avoids cache contamination with unnecessary data and makes a greater cache capacity available to the read operation. Non-temporal stores are suitable for the transposition operation, because once the transposed tile is written, its data are not re-used in the course of transposition. This type of write operations can be requested by using the compiler argument "`-opt-streaming-stores always`". Alternatively, the Intel C++ compiler can implement non-temporal stores for a single loop if "`#pragma vector nontemporal`" is placed before this loop. In the present work, the former method (the compiler argument) is used. Non-temporal stores improve the transposition performance by approximately 2%.

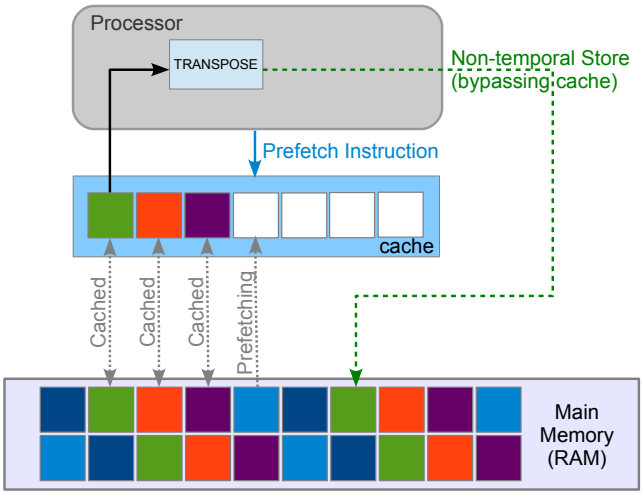

Figure 10: Software prefetching: processor requests the cache to fetch data from the main memory ahead of time. Prefetch distance is estimated by the compiler heuristically, but can be overridden by the compiler argument "`-opt-prefetch-distance=n`". Non-temporal stores: processor writes data directly to the main memory, bypassing the cache. Compiler argument "`-opt-streaming-stores always`" enforces non-temporal write operations for the compiled object file.

2.5 Mustang Taming: Thread Affinity

By default, OpenMP threads are allowed to migrate from one logical and physical core to another in the course of application execution. This often leads to decreased performance, because when a thread (i.e., an instance of a parallel code) moves from one physical core to another, it loses access to the data in the local caches of that core. It is possible to improve the application performance and reduce the performance fluctuations by binding OpenMP threads to logical or physical cores. The mapping of OpenMP threads to logical and physical cores is called *thread affinity*.

In the present implementation, the thread affinity is set at runtime by using the environment variable "`KMP_AFFINITY=compact`". This type of affinity places OpenMP threads with adjacent numbers as close to each other as possible: if the current physical core is not full, then the thread is placed on that physical core; otherwise, it is placed on the next physical core. For multi-CPU solutions, one CPU socket is filled first, and only then are the threads placed on the next CPU. By default (unless parameter "`granularity`" is specified in the affinity mask), the granularity of "compact" OpenMP affinity is a single physical core, i.e. threads are allowed to migrate between the logical cores of a single physical core. This has no appreciable effect on the performance of the matrix transposition application.

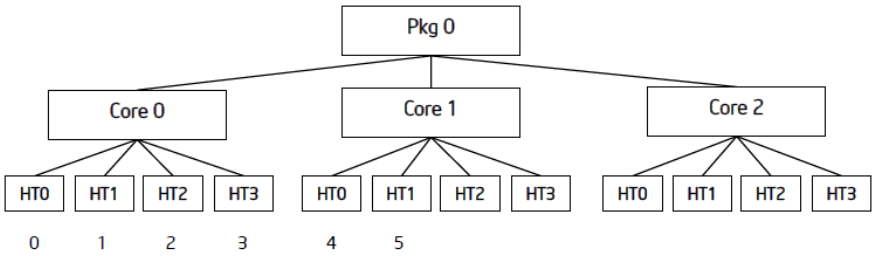

Figure 11: OpenMP thread affinity of type "`compact`" places consecutive OpenMP threads as close to each other as possible and prevents thread migration across physical cores. Image credit: Intel.

2.6 "The Good, the Bad and the Ugly" Matrix Sizes

Best results are achieved when the matrix row length is a multiple of the cache line size, which is 64 bytes in the Intel Xeon architecture as well as in the Intel Xeon Phi architecture. Therefore, the matrix size `n` must be a multiple of 64/4=16 in single precision or 64/8=8 in double precision. Matrix sizes that satisfy this condition are hereafter referred to as "good" sizes, unless they also satisfy the criterion of "ugly" (see below). Examples of "good" matrix sizes are `n`=960, 8000, 22000.

A "bad" matrix size is when the matrix row length in bytes is not a multiple of the cache line length. The problem with this size is that the first element of a matrix row is not always mapped to the beginning of a cache line. Therefore, when the transposition microkernel reads a block of elements equal in size to the cache line, it sometimes has to load/store data from/to two cache lines instead of one. In addition, false sharing may occur when a cache line is shared between two tiles, and two threads processing the tiles simultaneously modify the contents of that cache line. Examples of "bad" matrix sizes: `n`=962, 8051, 22004.

Finally, "ugly" matrix sizes are those for which the length of a matrix row in bytes is a multiple of the cache set conflict length, or close to it. Cache set conflicts occur in associative caches, where any given memory address can be mapped to only a small region of the cache, called a set. The mapping between memory addresses and allowable cache sets repeats with a periodicity called the cache set conflict length. For Intel Xeon Phi architecture, the cache set conflict size is 4 kilobytes. This means that data elements 4 kilobytes apart in memory are mapped to the same cache set. When an application traverses data in memory with a stride that is a multiple of 4 kilobytes, only a fraction of the cache may be used, which decreases the performance. Examples of "ugly" matrix sizes are `n`=1024, 8192, 21504.

The implementation presented here produces correct results for any matrix size. However, the highest performance is achieved for "good" sizes. In practical applications that require transposition, it is usually possible to choose matrix sizes so that they fall into the "good" category. If necessary, a "good" matrix size can be achieved by padding the matrix with unused rows and columns.

3 Benchmarks

3.1 Transposition Rate

In this paper, the performance of transposition is reported in the units of bandwidth (GB/s). The transposition rate is calculated by dividing the matrix size, in gigabytes, by the transposition time, in seconds, and multiplying the result by 2:

$$\text{Transposition Rate} = \frac{N \times N \times \text{sizeof(TYPE)}}{2^{30} \times \text{Time}} \times 2, \tag{2}$$

where N is the matrix size. This is the conventional way to report the transposition performance. The factor of 2 accounts for the fact that in order to transpose the matrix, the data has to be read and then written, thus resulting in two memory accesses per matrix element. Multiplying the size to time ratio by 2 also allows to directly compare the results to the STREAM benchmark [10] (the "copy" test), which also multiplies the ratio of size to time by a factor of 2. Specifically, the theoretical maximum transposition rate in the sense of Equation (2) is equal to the STREAM "copy" bandwidth.

When comparing the present results to the previous white paper [4], note that in the latter publication, the factor of 2 was not included in the reported transposition rate, however, it is included in the present report.

3.2 Harware System Configuration

In this paper, all tests are run on a CX2265i-XP5 server[4] with the following specifications:

1) Host system: two eight-core Intel Xeon E5-2680 processors with two-way hyper-threading, 64 GB of DDR3 RAM at 1,333 MHz, running Cent OS 6.4, using the Intel C++ Compiler version 13.1.3.192 (Build 20130607) to compile the code;
2) Coprocessor: one 61-core Intel Xeon Phi coprocessor SKU B1QS-7110P with 16 GB of GDDR5 RAM, running the MPSS (MIC Platform Software Stack, the driver suite for Intel Xeon Phi coprocessors) version 2.1.6720-13.

The coprocessor and the host CPU have the ECC (error-correcting code) functionality for runtime protection against single-bit corruption of data in RAM. This functionality was enabled for all tests in this paper. It is a known fact that disabling the ECC functionality may increase the performance of bandwidth-bound benchmarks. However, in practical applications, the ECC mode is usually employed, and therefore for these tests we keep ECC on.

3.3 STREAM Benchmark

In order to put the transposition performance in perspective, the STREAM benchmark [10] was run on the host CPUs as well as on the coprocessor. The result of the STREAM "copy" test provides an upper bound on the performance of matrix transposition. The STREAM benchmark is compiled for the host and for the coprocessor using the following commands, as recommended in [11]:

```
andrey@dublin$ # Compile CPU version:
andrey@dublin$ icpc -O3 -openmp -DSTREAM_ARRAY_SIZE=64000000 -o stream stream.c
andrey@dublin$ #
andrey@dublin$ # Compile MIC version:
andrey@dublin$ icpc -O3 -openmp -DSTREAM_ARRAY_SIZE=64000000 \
               -mmic -opt-prefetch-distance=64,8 -opt-streaming-cache-evict=0 -ffreestanding \
               -o stream-MIC stream.c
```

Figure 12: Compilation of the STREAM benchmark.

[4] http://www.colfax-intl.com/xeonphi/CX2265i-XP5.html

The STREAM benchmark was executed on the host as usual, and on the coprocessor in the native execution mode. For native execution, the executable code and dependent libraries are copied to the virtual filesystem of the coprocessor, and then the application is started directly on the device via an SSH session. The host processor does not participate in the native application execution. See also Section 3.4, where native execution is illustrated for the transposition benchmark.

In order to achieve optimal results, the number of threads on the host was chosen as to 16, and on the coprocessor as 60, by setting the environment variable `OMP_NUM_THREADS` to the respective value. This setting corresponds to using one thread per physical core on each device[5]. In addition, the OpenMP thread affinity was set to type "scatter" by assigning the environment variable `KMP_AFFINITY=scatter`.

The STREAM benchmark results are summarized in Table 1.

Test	Two Intel Xeon E5-2680	Intel Xeon Phi 7110P
`KMP_AFFINITY`	scatter	scatter
`OMP_NUM_THREADS`	16	60
ECC	on	on
Copy	60.4 GB/s	169.2 GB/s
Scale	66.0 GB/s	166.5 GB/s
Add	65.8 GB/s	174.3 GB/s
Triad	66.3 GB/s	174.1 GB/s
Theoretical Peak Bandwidth	102.4 GB/s	352 GB/s

Table 1: Results of the STREAM benchmark.

The theoretical peak bandwidth of the host system with two Intel Xeon E5-2860 CPUs is 102.4 GB/s [12], and so the STREAM benchmark achieves 60-65% efficiency. For the Intel Xeon Phi 7110P coprocessor, the theoretical peak bandwidth is 352 GB/s [13], and STREAM tests are 48-50% efficient.

3.4 Compilation and Execution

Figure 13 shows the command lines used to compile the double precision version of the matrix transposition application. These commands include the compiler flags discussed in Section 2.4:

```
andrey@dublin$ # CPU version:
andrey@dublin$ icpc -c Transpose.cc -o Transpose-dp-CPU.o \
               -DDOUBLE -O3 -openmp -opt-prefetch-distance=8
andrey@dublin$ icpc -c Main.cc -o Main-dp-COU.o -DDOUBLE -O3 -openmp
andrey@dublin$ icpc -o runme-dp-CPU Main-dp-CPU.o Transpose-dp-CPU.o -O3 -openmp
andrey@dublin$ #
andrey@dublin$ # MIC version:
andrey@dublin$ icpc -c Transpose.cc -o Transpose-dp-MIC.o \
               -DDOUBLE -O3 -openmp -mmic -opt-prefetch-distance=8 -opt-streaming-stores always
andrey@dublin$ icpc -c Main.cc -o Main-dp-MIC.o -DDOUBLE -O3 -openmp -mmic
andrey@dublin$ icpc -o runme-dp-MIC Main-dp-MIC.o Transpose-dp-MIC.o -O3 -openmp -mmic
```

Figure 13: Compilation of the double precision version of the transposition benchmark for the CPU (executable file `runme-dp`) and for the coprocessor (executable file `runme-dp-MIC`).

The resulting executable `runme-dp` is compiled for the Intel Xeon architecture, and `runme-dp-MIC` is compiled for native execution on Intel Xeon Phi coprocessors. For running the code (see Figure 14),

[5]Even though the 7110P coprocessor contains 61 cores, STREAM performs better with 60 threads. The extra core can be beneficial in offload applications, where this core is dedicated to offload task management.

the environment variable `KMP_AFFINITY=compact` was used, as discussed in Section 2.5. The value of `OMP_NUM_THREADS` was chosen as 32 on the host (16 cores with 2-way hyper-threading), and 244 on the coprocessor (61 cores with 4-way hyper-threading). For native execution, the executable file and the libraries used by it are transferred to the coprocessor's virtual file system using the command `scp`. Then the user can obtain a shell using the `ssh` client, set up the environment variables, and launch the application directly on the Intel Xeon Phi coprocessor.

```
andrey@dublin$ # Benchmark on the CPU-based host
andrey@dublin$ export KMP_AFFINITY=compact
andrey@dublin$ export OMP_NUM_THREADS=32
andrey@dublin$ ./runme-dp-CPU
...
andrey@dublin$ #
andrey@dublin$ # Benchmark on the MIC-based coprocessor
andrey@dublin$ # Copy required libraries to the coprocessor:
andrey@dublin$ scp /opt/intel/composer_xe_2013.5.192/compiler/lib/mic/libiomp5.so mic0:~/
andrey@dublin$ # Copy the code to the coprocessor:
andrey@dublin$ scp runme-dp-MIC runme-sp-MIC mic0:~/
andrey@dublin$ # Log in to the coprocessor to run the code:
andrey@dublin$ ssh mic0
andrey@dublin-mic0$ export KMP_AFFINITY=compact
andrey@dublin-mic0$ export OMP_NUM_THREADS=244
andrey@dublin-mic0$ export LD_LIBRARY_PATH=.
andrey@dublin-mic0$ ./runme-dp-MIC
...
andrey@dublin-mic0$ exit
andrey@dublin$ #
```

Figure 14: Execution of the transposition benchmark on the host and on the coprocessor in the native execution mode.

The text output of the benchmark was collected, parsed and presented in plots and tables in Section 3.5 and 3.6. Like with the STREAM benchmark, the ECC functionality of the host and the coprocessor was enabled for all tests.

3.5 Results: "Good" Matrix Sizes

Transposition benchmarks for "good" matrix sizes (see Section 2.6 for definition) are summarized in Figure 15. Note that for the results on the host CPUs, matrices under 30 MB in size fit in the L3 cache of each Xeon E5-2680 processor. In order to simulate a realistic situation where the matrix is not in the processor cache at the start of transposition, the benchmark evicts the contents of the cache between benchmark trials by performing read/write operations on a large dummy array.

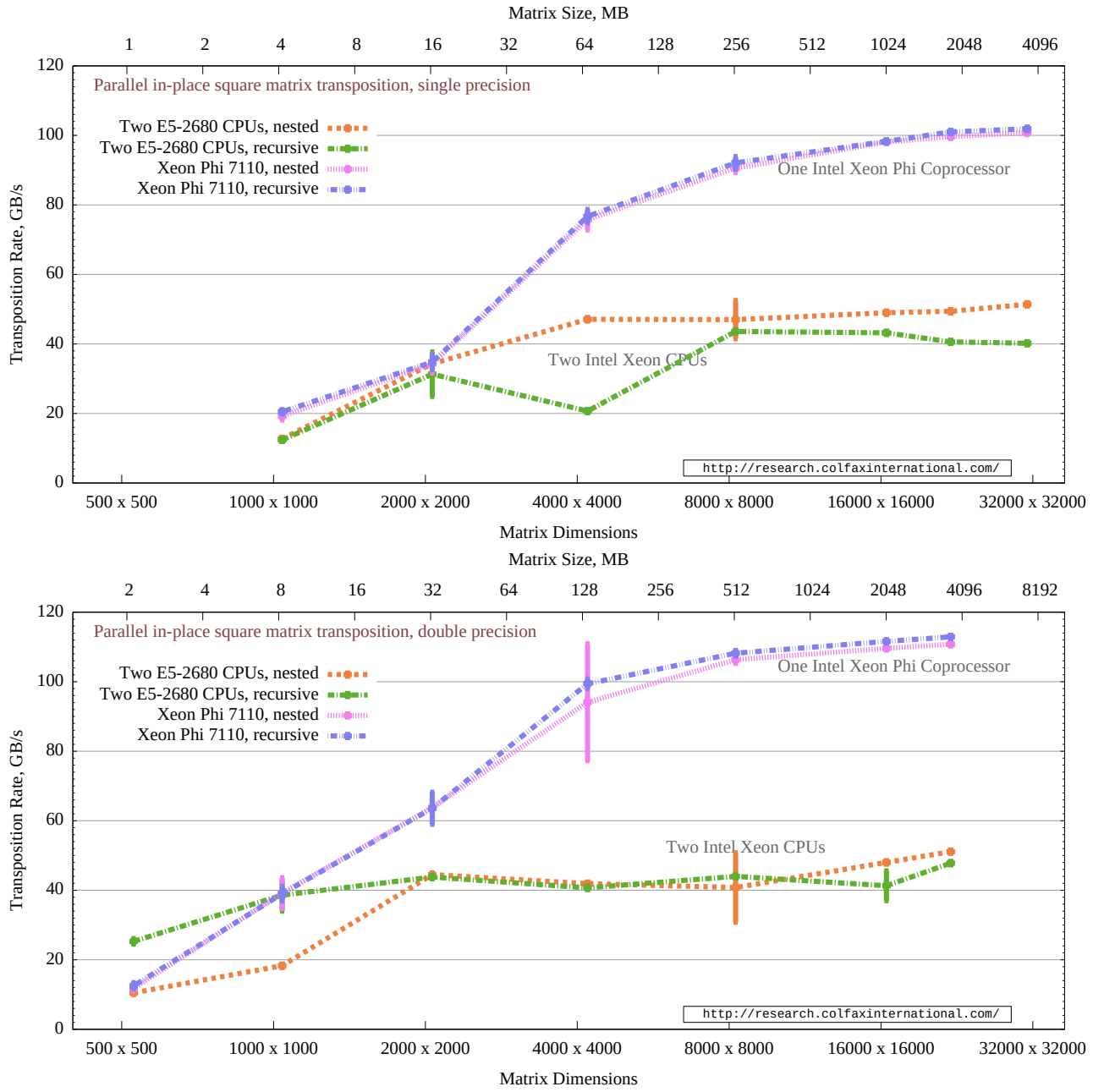

Figure 15: Parallel in-place square matrix transposition rate for "good" matrix sizes. Error bars are one mean square deviation of 20 trials. See text for details.

3.6 Results: "Good" versus "Bad" and "Ugly" Matrix Sizes

		Performance in Single Precision					
		Host			Coprocessor		
#		"Good"	"Bad"	"Ugly"	"Good"	"Bad"	"Ugly"
1	Size	1040×1040	1030×1030	1024×1024	1040×1040	1030×1030	1024×1024
	Nested	42.3 ± 1.9	36.4 ± 2.8	29.8 ± 1.2	23.2 ± 3.2	21.2 ± 0.2	18.0 ± 0.4
	Recursive	44.0 ± 1.0	33.0 ± 1.2	35.1 ± 0.9	21.9 ± 0.6	18.6 ± 2.9	18.3 ± 0.2
2	Size	2064×2064	2060×2060	2048×2048	2064×2064	2060×2060	2048×2048
	Nested	55.8 ± 2.6	33.1 ± 8.4	22.0 ± 6.5	37.5 ± 0.6	32.5 ± 0.5	26.7 ± 0.6
	Recursive	55.5 ± 2.7	31.7 ± 8.5	38.1 ± 0.5	36.0 ± 3.0	28.4 ± 0.2	22.9 ± 1.4
3	Size	4160×4160	4100×4100	4096×4096	4160×4160	4100×4100	4096×4096
	Nested	22.2 ± 0.2	21.7 ± 0.3	15.1 ± 0.3	77.9 ± 5.2	60.2 ± 1.6	20.3 ± 1.3
	Recursive	26.6 ± 4.5	24.6 ± 2.3	14.9 ± 0.0	77.6 ± 2.1	57.5 ± 0.9	20.4 ± 2.7
4	Size	8240×8240	8210×8210	8192×8192	8240×8240	8210×8210	8192×8192
	Nested	33.4 ± 0.7	33.7 ± 0.3	27.2 ± 0.8	93.1 ± 1.1	71.4 ± 1.2	8.1 ± 0.0
	Recursive	36.1 ± 0.6	46.1 ± 0.3	28.2 ± 0.7	90.8 ± 0.7	69.0 ± 0.7	7.5 ± 0.0
5	Size	16400×16400	16390×16390	16384×16384	16400×16400	16390×16390	16384×16384
	Nested	44.9 ± 0.2	37.3 ± 0.2	29.1 ± 0.1	98.0 ± 0.2	71.2 ± 0.3	4.9 ± 0.0
	Recursive	50.4 ± 0.4	48.6 ± 0.2	30.2 ± 0.0	98.1 ± 0.1	68.3 ± 0.2	4.8 ± 0.0
6	Size	22000×22000	22004×22004	21504×21504	22000×22000	22004×22004	21504×21504
	Nested	41.9 ± 0.2	36.6 ± 0.1	36.0 ± 0.1	100.9 ± 0.3	78.9 ± 0.1	47.2 ± 0.1
	Recursive	52.7 ± 0.2	50.8 ± 0.2	40.2 ± 0.0	99.5 ± 0.3	79.5 ± 0.2	46.5 ± 0.1
7	Size	31200×31200	31100×31100	30720×30720	31200×31200	31100×31100	30720×30720
	Nested	40.5 ± 0.2	34.7 ± 0.1	35.1 ± 0.1	102.0 ± 0.1	78.5 ± 0.0	46.5 ± 0.0
	Recursive	51.4 ± 0.2	48.7 ± 0.1	38.9 ± 0.1	100.8 ± 0.1	79.3 ± 0.1	45.9 ± 0.0

		Performance in Double Precision					
		Host			Coprocessor		
#		"Good"	"Bad"	"Ugly"	"Good"	"Bad"	"Ugly"
1	Size	528×528	524×524	512×512	528×528	524×524	512×512
	Nested	21.3 ± 1.4	13.2 ± 0.6	10.9 ± 0.5	14.7 ± 0.2	16.8 ± 0.4	14.5 ± 0.4
	Recursive	18.7 ± 6.9	18.0 ± 0.6	12.2 ± 0.5	14.1 ± 0.3	15.2 ± 5.4	16.1 ± 0.4
2	Size	1040×1040	1030×1030	1024×1024	1040×1040	1030×1030	1024×1024
	Nested	26.4 ± 1.5	24.7 ± 1.1	20.8 ± 0.6	40.6 ± 0.6	38.0 ± 4.0	32.3 ± 3.3
	Recursive	26.8 ± 1.5	23.8 ± 1.0	21.5 ± 0.6	38.4 ± 0.6	39.2 ± 0.9	27.8 ± 3.6
3	Size	2064×2064	2060×2060	2048×2048	2064×2064	2060×2060	2048×2048
	Nested	40.1 ± 1.5	80.7 ± 5.0	51.5 ± 1.8	66.4 ± 1.8	60.5 ± 1.1	39.3 ± 3.0
	Recursive	39.6 ± 1.1	62.0 ± 13.8	56.4 ± 1.5	65.7 ± 2.0	58.5 ± 1.5	34.3 ± 0.8
4	Size	4160×4160	4100×4100	4096×4096	4160×4160	4100×4100	4096×4096
	Nested	42.3 ± 0.1	24.7 ± 0.1	39.3 ± 0.2	98.5 ± 2.3	87.9 ± 2.2	16.6 ± 0.3
	Recursive	44.4 ± 0.1	49.3 ± 0.5	29.1 ± 9.2	97.3 ± 1.7	85.0 ± 1.0	15.2 ± 0.1
5	Size	8240×8240	8210×8210	8192×8192	8240×8240	8210×8210	8192×8192
	Nested	44.4 ± 0.2	45.9 ± 0.6	36.5 ± 2.3	108.0 ± 1.1	92.3 ± 0.4	16.0 ± 0.0
	Recursive	48.1 ± 0.1	49.4 ± 0.7	29.3 ± 7.8	106.5 ± 0.2	90.4 ± 0.1	15.5 ± 0.0
6	Size	16400×16400	16390×16390	16384×16384	16400×16400	16390×16390	16384×16384
	Nested	45.0 ± 0.2	41.9 ± 0.1	33.1 ± 0.2	111.5 ± 0.1	92.8 ± 0.2	16.2 ± 0.0
	Recursive	48.4 ± 0.2	47.0 ± 0.1	34.9 ± 0.1	109.6 ± 0.1	90.1 ± 0.2	16.1 ± 0.0
7	Size	22000×22000	22004×22004	21504×21504	22000×22000	22004×22004	21504×21504
	Nested	47.5 ± 0.4	46.8 ± 0.2	43.9 ± 0.3	112.9 ± 0.1	100.4 ± 0.1	83.0 ± 0.1
	Recursive	49.3 ± 0.2	50.3 ± 0.6	47.5 ± 0.2	110.8 ± 0.1	99.6 ± 0.1	81.7 ± 0.0

Table 2: Performance results. Top table: single precision, bottom table: double precision. In each cell, the first line is the matrix size. The second and third lines are the transposition rates and one mean square deviation of 20 trials, in GB/s. See Section 2.6 for more information.

3.7 Comparison with Intel MKL

The Intel Math Kernel Library (Intel MKL) [7] has functions `mkl_?imatcopy`, which can perform the same in-place matrix transposition operation as we implemented in this paper [14]. `mkl_?imatcopy` has more functionality than the routine developed here: it can transpose non-square matrices and perform data scaling simultaneously with transposition. In addition, it is highly tuned for matrix sizes proportional to powers of 2. The implementation of `mkl_?imatcopy` found in MKL version 11.0.5 performs best when the number of threads is coordinated with the matrix size: n must be a multiple of the number of threads times the number of data elements in a cache line.

Figure 16 shows a comparison of the performance of our transposition routine with `mkl_dimatcopy` on an Intel Xeon Phi coprocessor. For this comparison, two sets of matrix sizes were chosen:

1. For the plot in the left panel, sizes favorable for both our routine and the MKL function were chosen: 1952, 3904, 5856, 7808, 9760, 13664, 17568, 21472. These sizes are "good" in our definition, and, additionally, they are multiples of the number of threads (244) times the number of elements in a cache line (8 for double precision).
2. For the plot in the right panel, sizes favorable for MKL, but sub-optimal for our routine were chosen: 2048, 3072, 4096, 6144, 8192, 12288, 16384, 21504. These sizes are "ugly" in our definition, however, they are powers of 2 or sums of large powers of 2. 128 threads were used.

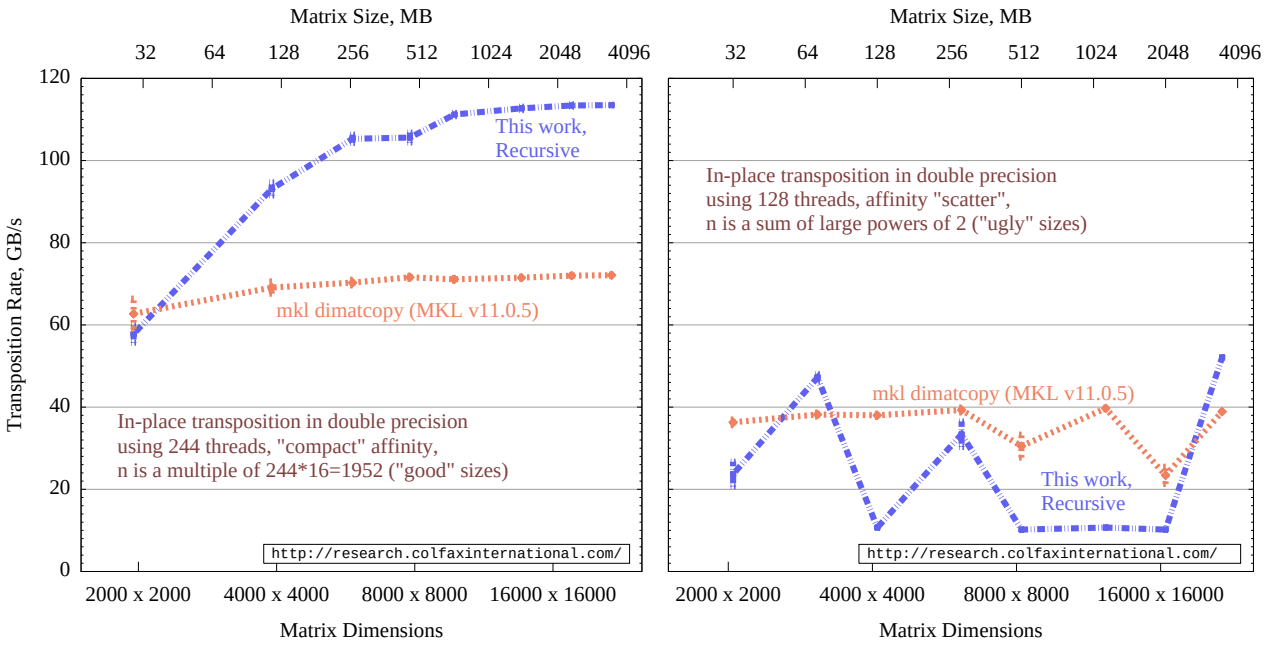

Figure 16: Comparison between the present work and Intel MKL.

According to the plots in Figure 16, the implementation presented here performs faster than MKL by up to 50% if matrix sizes are "good". For n equal to powers of 2, MKL achieves better results.

The choice of the transposition method for a practical application is dependent on the restrictions in the application. If the application requires non-square matrix transposition, or if the transposed matrix is of power of 2 in size, and the number of threads can be coordinated with the matrix size, then `mkl_?imatcopy` is the way to go. However, if the matrix is square, and its size can be padded to a "good" value, or if the number of threads for the transposition routine cannot be coordinated with the matrix size, then the application presented here can perform faster.

If you wish to use one of these codes for your application, it is advisable to download and test the latest version of MKL, because the situation may be different in future versions of the library.

4 Discussion

The cowboy-themed titles of Sections 2.1 — 2.6 reflect the fact that the primary interest of this publication is the speed that can be achieved for the ubiquitous operation of matrix transposition. In double precision with ECC on, the transposition rate for the best of the two plans (nested and recursive) on the two host CPUs approaches 49 GB/s for large matrices, and on the coprocessor, 113 GB/s was achieved for the biggest matrix. These numbers correspond to 82% and of 67%, respectively, of the upper bound on the transposition performance determined by the STREAM "copy" bandwidth. The Intel Xeon Phi 7110P coprocessor performs 2.3x faster than two Intel Xeon E5-2680 CPUs for this operation.

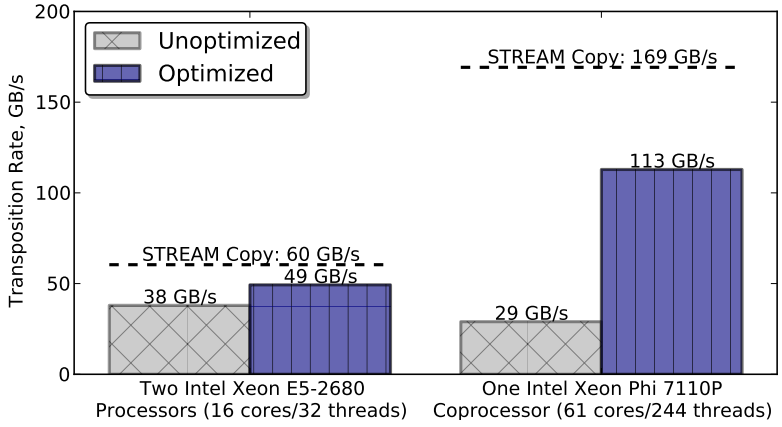

Figure 17: Transposition rate for the the unoptimized method (Figure 2) and for the optimized method (Figure 18) with the best of the two plans (nested and recursive) for matrix size 22000 × 22000 in double precision, and the STREAM "copy" test bandwidth (upper limit on the transposition rate), with ECC on. Same C language code compiled for the CPU and for the MIC platform.

The key to accelerating the transposition operation is cache traffic optimization. The unoptimized code shown in Figure 2 achieves only 38 GB/s on the host and 29 GB/s on the coprocessor for the same case (with `KMP_AFFINITY=scatter` and `OMP_NUM_THREADS=16` and `240`). Achieving the high efficiency and performance benefits from the MIC architecture, as reported above, is possible through the use of loop tiling, which increases data locality in time. Another crucial optimization is enabling fine-granularity thread parallelism with low scheduling overhead, which was achieved by planning the tile traversal pattern before executing the transposition. Additional optimizations (data alignment, using `#pragma simd`, and tuning the compiler arguments) help as well, but to a smaller degree.

It is typical that code optimization yields a greater speedup on the Intel Xeon Phi coprocessor than on the Intel Xeon CPU-based system. The CPU is more forgiving than the coprocessor, because it contains more resources to accommodate sub-optimal code, such as a unified Level-2 cache and hardware prefetchers in the Level-1 cache. As soon as the code allows the Intel Xeon Phi coprocessor to efficiently use the cache and employ all available parallelism, performance on the coprocessor jumps up.

However, besides practical achievable performance of Intel Xeon Phi coprocessor, this paper demonstrates a very important feature of the programmability of this architecture. Indeed, the C language code used for the benchmark on the CPU is transformed into an application for the MIC architecture simply by recompilation. No low-level code is used, and the most sophisticated job of implementing the tile transposition microkernel is handled completely by the compiler.

The application presented here is 82% efficient on the CPU and 67% efficient on the coprocessor. Therefore, additional optimizations are theoretically able to further accelerate the code by no more than 1.2x on the host and 1.5x on the coprocessor. The reader is welcome to download the source code of the benchmark from http://research.colfaxinternational.com/post/2013/08/12/Trans-7110.aspx to assess the effort required for cross-platform portability with efficiency as high as achieved here.

Acknowledgements

I thank Evarist Fomenko of Intel Corporation for a helpful consultation on the functionality of MKL.

A Source code

```
void Transpose(FTYPE* const A, const int n, const int* const plan ) {// FTYPE is float or double
  const int TILE = 32;                                  // Tile size
  const int nEven = n - n%TILE;                         // nEven is a multiple of TILE
  const int wTiles = nEven / TILE;                      // Complete tiles in each dimens.
  const int nTilesParallel = wTiles*(wTiles - 1)/2; // # of complete tiles under the main diag.
#pragma omp parallel                                    // Start of parallel region
  {
#pragma omp for schedule(guided)
    for (int k = 0; k < nTilesParallel; k++) {          // Parallel loop over body tiles
      const int ii = plan[2*k + 0];                     // Top column of the tile (planned)
      const int jj = plan[2*k + 1];                     // Left row of the tile (planned)

      for (int j = jj; j < jj+TILE; j++)                // Tile transposition microkernel:
#pragma simd                                            // Ensure automatic vectorization
        for (int i = ii; i < ii+TILE; i++) {
          const FTYPE c = A[i*n + j];                   //
          A[i*n + j] = A[j*n + i];                      //  Swap matrix elements
          A[j*n + i] = c;                               //
        }
    }                                                   // End of main parallel for-loop

#pragma omp for schedule(static)
    for (int ii = 0; ii < nEven; ii += TILE) {          // Transposing tiles on the main diagonal:
      const int ii = jj;
      for (int j = jj; j < jj+TILE; j++)                // Diagonal tile transposition microkernel:
#pragma simd                                            // Ensure automatic vectorization
        for (int i = ii; i < j; i++) {                  // Avoid duplicate swaps
          const FTYPE c = A[i*n + j];                   //
          A[i*n + j] = A[j*n + i];                      //  Swap matrix elements
          A[j*n + i] = c;                               //
        }
    }

#pragma omp for schedule(static)
    for (int j = 0; j < nEven; j++)                     // Transposing the "peel":
      for (int i = nEven; i < n; i++) {
       const FTYPE c = A[i*n + j];                      //
       A[i*n + j] = A[j*n + i];                         // Swap matrix elements
       A[j*n + i] = c;                                  //
      }
  }                                                     // End of thread-parallel region

  for (int j = nEven; j < n; j++)                       // Transposing bottom-right cornr
    for (int i = nEven; i < j; i++) {
      const FTYPE c = A[i*n + j];                       //
      A[i*n + j] = A[j*n + i];                          // Swap matrix elements
      A[j*n + i] = c;                                   //
    }
}
```

Figure 18: Improved implementation of parallel in-place square matrix transposition with loop tiling.

References

[1] Harald Prokop. Cache-Oblivious Algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.
http://supertech.csail.mit.edu/papers/Prokop99.pdf.

[2] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *40th Annual Symposium on Foundations of Computer Science*, 1999.
http://doi.ieeecomputersociety.org/10.1109/SFFCS.1999.814600.

[3] D. Tsifakis, Alistair P. Rendell, and Peter E. Strazdins. Cache Oblivious Matrix Transposition: Simulation and Experiment. In *International Conference on Computational Science*, pages 17–25, 2004.
http://www.springeronline.com/3-540-22115-8.

[4] Andrey Vladimirov. Cache Traffic Optimization on Intel Xeon Phi Coprocessors for Parallel In-Place Square Matrix Transposition with Intel Cilk Plus and OpenMP.
http://research.colfaxinternational.com/post/2013/04/25/Transposition-Xeon-Phi.aspx.

[5] Boost C++ Libraries.
http://www.boost.org/.

[6] FFTW Library.
http://www.fftw.org/.

[7] Intel Math Kernel Library.
http://software.intel.com/en-us/intel-mkl.

[8] Greg Ruetsch and Paulius Micikevicius. Optimizing Matrix Transpose in CUDA, June 2010.
http://docs.nvidia.com/cuda/samples/6_Advanced/transpose/doc/MatrixTranspose.pdf.

[9] Mark Harris. An Efficient Matrix Transpose in CUDA C/C++, Feb 2013.
https://developer.nvidia.com/content/efficient-matrix-transpose-cuda-cc.

[10] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers.
http://www.cs.virginia.edu/stream/.

[11] Karthik Raman. Optimizing Memory Bandwidth on Stream Triad, Feb 2013.
http://software.intel.com/en-us/articles/optimizing-memory-bandwidth-on-stream-triad.

[12] Intel Xeon Processor E5-2860 (20M Cache, 2.70 GHz, 8.00 GT/s Intel QPI).
http://ark.intel.com/ru/products/64583.

[13] Intel Xeon Phi Coprocessor 7120P (16GB, 1.238 GHz, 61 core).
http://ark.intel.com/products/75799.

[14] Intel Math Kernel Library, mkl_imatcopy.
http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/hh_goto.htm#GUID-B4584828-531D-4385-8F63-22C760C9B4B4.htm.