Cache Traffic Optimization on Intel Xeon Phi Coprocessors for Parallel In-Place Square Matrix Transposition with Intel Cilk Plus and OpenMP

Andrey Vladimirov
Stanford University
for Colfax Research

April 25, 2013

Abstract

Numerical algorithms sensitive to the performance of processor caches can be optimized by increasing the locality of data access. Loop tiling and recursive divide-and-conquer are common methods for cache traffic optimization. This paper studies the applicability of these optimization methods in the Intel Xeon Phi architecture for the in-place square matrix transposition operation. Optimized implementations in the Intel Cilk Plus and OpenMP frameworks are presented and benchmarked. Cache-oblivious nature of the recursive algorithm is compared to the tunable character of the tiled method. Results show that Intel Xeon Phi coprocessors transpose large matrices faster than the host system, however, smaller matrices are more efficiently transposed by the host. On the coprocessor, the Intel Cilk Plus framework excels for large matrix sizes, but incurs a significant parallelization overhead for smaller sizes. Transposition of smaller matrices on the coprocessor is faster with OpenMP.

Contents

1 Memory Access, Cache Oblivious Algorithms, and Transposition

In modern computer architectures, the latency of main memory access is far greater than the latency of common arithmetic operations. In order to keep the hardware arithmetic units occupied, computer architectures use hierarchical caches that can hold a smaller amount of data than the main memory, but provide much lower latency. In addition, software and hardware prefetching can request the movement of data from memory to caches ahead of time. Such cache-based hardware architectures provide optimum performance when the numerical algorithm has a property known as "data locality". Data locality means that

a) if a memory address is accessed by the code, it is likely that the adjacent memory address will soon be accessed by the code as well, and

b) when a memory access brings data from memory to cache, this data will be re-used as soon as possible.

Techniques for the optimization of data locality generally re-order memory accesses in a pattern that provides the best locality. A common technique for such optimization is loop tiling, also known as loop blocking. Loop tiling changes the order of operations in nested loops operating on a multi-dimensional array. In a tiled loop, for every iteration in the inner dimension, a small number of outer dimension operations are performed. The tile size (i.e., the number of outer dimension operations) must be tuned to the size of the cache in the system, and therefore the tiling method is termed "cache-aware".

A potentially more portable technique for memory locality improvement is recursive divide-and-conquer. Recursive methods of cache traffic optimization, termed "cache-oblivious algorithms", were proposed and developed by Prokop [1] and by Frigo et al. [2]. Recursive algorithms partition the problem into sub-problems operating on a smaller data subsets. These subsets are then recursively partitioned further. The recursion proceeds either to a pre-set recursion threshold, or to the smallest possible subset. The idea is that eventually the sub-problem will be small enough to fit in cache. This recursive technique is designed to be "cache-oblivious", i.e., it does not require fine tuning to the size of the cache. Indeed, once the sub-problem fits in cache, the recursion can continue further without harming the cache performance.

Cache optimization is of particular interest for the matrix transposition problem. The transposition operation transforms matrix A into matrix A^T by swapping matrix rows with columns:

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

The difficulty with this operation in computer architectures with cached memory is the non-local nature of data access in the straightforward implementation. Indeed, if the matrix is stored in a row-major order, then a "cache-ignorant" algorithm (see code listing below) necessarily has memory accesses of stride `n` (non-local access in space). In addition, by the time that the `j`-loop completes and `i` is incremented, the cache line containing `A[j*n + i]` may be evicted from cache, and the adjacent element must be fetched again (non-local access in time).

```
1  for (int i = 0; i < n; i++)
2    for (int j = 0; j < i; j++) {
3      const float aij = A[i*n + j];
4      A[i*n + j] = A[j*n + i];
5      A[j*n + i] = aij;
6    }
```

This algorithm contains two nested loops and therefore can be improved by the loop tiling technique. For best performance, both of the nested loops can be tiled.

Alternatively, a recursive algorithm can be used for the optimization of transposition as proposed by Tsifakis et al. [3]. The algorithm proposed in that work applies to rectangular matrices of any dimensions, but the present discussion is restricted to square matrices. In the recursive algorithm, matrix A must be divided into two sub-matrices either horizontally, or vertically. Then each sub-matrix must be recursively partitioned further. The direction of partitioning alternates between horizontal and vertical, according to which dimension of the matrix is greater. This partitioning eventually makes the sub-matrix small enough

to fit into the system cache. Then the standard transposition algorithm is applied to transpose these small sub-matrices.

This paper researches the applicability of the tiling and recursion methods to optimize the performance of in-place square matrix transposition on two platforms:

1) Multi-core platform: two Intel Xeon E5 series processors based on the Sandy Bridge architecture, and
2) Many Integrated Core (MIC) platform: a single Intel Xeon Phi coprocessor based on the Knights Corner architecture chip.

The MIC platform discussion in this paper applies to cases when the transposed matrix data is already present on the coprocessor, as opposed to offloading the matrix to the coprocessor just to perform the transposition. Offloading data to the MIC architecture only for transposition is unlikely to accelerate a host-based calculation, because this problem has a computational cost of $O(N)$. Such low compute intensity algorithms are generally bottlenecked by the data transport time across the PCIe bus. However, in cases when transposition is a part of a greater application executing on the coprocessor, it is usually cheaper to avoid data transfer across the PCIe bus and transform the matrix on the accelerator. This motivates the need for the optimization of matrix transposition on the MIC architecture.

2 Baseline: Unoptimized Parallel Transposition

Transposition on both the Intel Xeon and Intel Xeon Phi architectures must be thread-parallel in order to be efficient. It is easy to achieve thread parallelism in the naïve transposition algorithm using the Intel Cilk Plus parallel framework. Because there are no data races in this problem, the parallelization only requires replacing the keyword `for` with `_Cilk_for`:

```
1 void Transpose(float* const A, const int n){
2   _Cilk_for (int i = 0; i < n; i++) {
3     for (int j = 0; j < i; j++) {
4       const float aij = A[i*n + j];
5       A[i*n + j] = A[j*n + i];
6       A[j*n + i] = aij;
7     }
8   }
9 }
```

The `_Cilk_for` keyword indicates to the Intel Cilk Plus runtime library that the loop is thread-parallel. The library automatically creates as many parallel workers as there are logical cores in the system (as long as `n` is greater than the number of logical cores), and distributes the work across the workers.

The above code works on both the processor-based host system, and on the coprocessor. It is assumed that the coprocessor application is run in the native execution mode for the MIC platform. Accordingly, no additional modifications are necessary in order to port this code for Intel Xeon Phi coprocessors. However, in order to execute this function on the coprocessor, the code must be compiled with the compiler argument `-mmic` and called from similarly compiled native application.

3 Optimized Tiled and Recursive Implementations with Intel Cilk Plus

In an effort to improve the cache performance of the transposition code, loop tiling can be applied. The listing below demonstrates this technique.

```
1  void Transpose(float* const A, const int n){
2      // Tiled algorithm improves data locality by re-using data already in cache
3      const int TILE = 16;
4      _Cilk_for (int ii = 0; ii < n; ii += TILE) {
5          const int iMax = (n < ii+TILE ? n : ii+TILE);
6          for (int jj = 0; jj <= ii; jj += TILE)  {
7              for (int i = ii; i < iMax; i++) {
8                  const int jMax = (i < jj+TILE ? i : jj+TILE);
9  #pragma loop count avg(TILE)
10 #pragma simd
11                 for (int j = jj; j<jMax; j++) {
12                     const float c = A[i*n + j];
13                     A[i*n + j] = A[j*n + i];
14                     A[j*n + i] = c;
15                 }
16             }
17         }
18     }
19 }
```

Tiling is applied twice: once for the outer `i`-loop and once for the inner `j`-loop. As a result, the inner two out of four loops operate on a "tile", i.e., a section of the original matrix bounded by indices `ii...(ii+TILE)` and `jj...(jj+TILE)`. Here `TILE` is a constant defined at compile time, equal to 16 in this case. Section 4 explains the choice of the tiling constant. Defining the tiling constant at compile time allows the compiler to make optimal decisions regarding the unrolling and vectorization of the inner loop.

Special care is taken in order to handle situations when the matrix size `n` is not a multiple of `TILE`. The constants `iMax` and `jMax` restrict loop bounds so that the matrix is always accessed within the array bounds.

Another addition to the code is pragma-based compiler hints. `#pragma simd` informs the compiler that the loop following it must be vectorized. `#pragma loop count` suggests to the compiler to optimize the loop vectorization strategy for the anticipated average number of iterations in the loop. The iteration count in the clause `avg(TILE)` must be chosen at compile time in order for this compiler hint to be effective.

The recursive optimization of the matrix transposition algorithm is shown below.

```
1  void transpose_cache_oblivious_thread(
2          const int iStart, const int iEnd,
3          const int jStart, const int jEnd,
4          float* A, const int n){
5      const int RT = 32; // Recursion threshold
6      if ( ((iEnd - iStart) <= RT) && ((jEnd - jStart) <= RT) ) {
7          for (int i = iStart; i < iEnd; i++) {
8              int je = (jEnd < i ? jEnd : i);
9  #pragma simd
10 #pragma loop_count avg(RT)
11             for (int j = jStart; j < je; j++) {
12                 const float c = A[i*n + j];
13                 A[i*n + j] = A[j*n + i];
14                 A[j*n + i] = c;
15             }
16         }
17         return;
18     }
19
20     // Recursive fork-join below:
21     if ((jEnd - jStart) > (iEnd - iStart)) {
22         int jSplit = jStart + (jEnd - jStart)/2;
23         if (jSplit % 16) // Split at 64-byte aligned boundary
24           jSplit -= jSplit % 16;
25         _Cilk_spawn // Execute the following function asynchronously
26                 transpose_cache_oblivious_thread(iStart, iEnd, jStart, jSplit, A, n);
27         transpose_cache_oblivious_thread(iStart, iEnd, jSplit, jEnd, A, n);
28     } else {
29         const int iSplit = iStart + (iEnd - iStart)/2;
30         const int jMax = (jEnd < iSplit ? jEnd : iSplit);
31         if (iSplit % 16) // Split at 64-byte aligned boundary
32                 iSplit -= iSplit % 16;
33         _Cilk_spawn // Execute the following function asynchronously
34                 transpose_cache_oblivious_thread(iStart, iSplit, jStart, jMax, A, n);
35         transpose_cache_oblivious_thread(iSplit, iEnd, jStart, jEnd, A, n);
36     }
37 }
38
39 void Transpose(float* const A, const int n){
40     transpose_cache_oblivious_thread(0, n, 0, n, A, n);
41 }
```

The recursive algorithm calls a utility function `transpose_cache_oblivious_thread` (line 40), which recurses into itself. The arguments passed to this function are the original matrix pointer and size, and the values `iStart`, `iEnd`, `jStart` and `jEnd`, which indicate the sub-matrix for the recursive step in the method of Tsifakis et al. [3].

At the beginning of the recursive part of the code, the program checks whether the sub-matrix is small enough to stop recursion and perform a serial transposition operation (line 6). The threshold at which this decision is made is controlled by the constant `RT`. Section 4 explains the choice of `RT` = 32 (line 5). The loops in the serial transposition code (lines 7 and 11) are not tiled, because the practical value of the optimal recursion threshold is too low. However, the same vectorization pragmas (lines 9 and 10) are used as in the tiled version shown in Section 3.

If the threshold has not been reached, the function proceeds to split the matrix and recursively launch the sub-matrix transpositions (lines 20 to 36). Depending on which dimension of the sub-matrix is longer, the matrix is split either horizontally, or vertically. Pruning is used to eliminate recursive function calls for sub-matrices above the main diagonal of the main matrix (line 30). After that, two recursive instances of the function are launched to process these sub-matrices.

The recursive code continues to use the Intel Cilk Plus parallel framework to express parallelism. Parallel recursion can be expressed with the keyword `_Cilk_spawn` (lines 25 and 33). This keyword instructs the

Cilk runtime library that the function following it (lines 26 and 34) must be executed asynchronously. The second recursive call of the function (lines 27 and 35) with the second sub-matrix starts immediately, without waiting for the first call to return. Intel Cilk Plus takes care of work scheduling. It launches spawned functions as system resources become available. This ensures that the system is not over-subscribed by running too many parallel functions. There is an implicit synchronization point at the end of the scope in which a `_Cilk_spawn` call has been made. At the synchronization point, execution blocks until all spawned functions have returned.

One additional optimization specific to the Intel Xeon Phi architecture is made in lines 23, 24 and 31,32. This optimization adjusts the sub-matrix dimensions in such a way that as many sub-matrices as possible have dimensions equal to a multiple of 16. In the double precision version, the value 8 is used instead of 16. This helps the execution of the vector loop in line 11, because

a) when the number of vector loop iterations is not a multiple of 16, the beginning or end of the loop must be performed with scalar or masked operations, and

b) when the data accessed by the loop is aligned on a 64-byte boundary, the memory access for loading data to and from vector registers is more efficient.

In order to facilitate aligned memory access, the memory block holding the matrix must be allocated on a 64-byte aligned boundary. This can be achieved by using the function `_mm_malloc` as illustrated in the listing below.

```
1 #include <malloc.h>
2   // ...
3   float* A = _mm_malloc(sizeof(float)*n*n, 64);
4   // ...
5   Transpose(A, n);
6   // ...
7   _mm_free(A);
```

Note that the corresponding function `_mm_free` must be used to deallocate the memory. If `n` is a multiple of 16 (for single precision) or 8 (for double precision), the effect of aligned memory access is especially strong.

4 Tuning the Algorithms

The optimal values for the tile size and the recursion threshold must be found empirically for each hardware platform. In this paper, all tests are run on a CX2265i-XP5 server[1] 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 Intel C++ Composer XE 13.1.1 to compile the code;
2) Coprocessor: one 60-core Intel Xeon Phi coprocessor SKU B1QS-5110P with 8 GB of GDDR5 RAM, running MPSS (MIC Platform Software Stack, the driver suite for Intel Xeon Phi coprocessors) version 2.1.5889-16.

For finding the optimum value of the tuning parameters, a square matrix of size 24000×24000 is used. In double precision, this size corresponds 4.3 GB of data, which is close to the size of the largest single contiguous array that can be allocated on the coprocessor. Figures 1 and 2 demonstrate the performance in single and double precision of the tiled and recursive codes with the value of the tuning parameter ranging from 1 to 2048. The performance is expressed in GB/s, which is the ratio of the matrix data size to the transposition time. The error bars are obtained by repeating the measurements multiple times (no less than 6) and calculating the mean and standard deviation. The first two trials for each data point are dropped from the statistical average.

The plots in the left-hand side of Figures 1 and 2 correspond to the tiled algorithm. For the tile size of 16 and 32 in single precision, the performance reaches a value of 20-25 GB/s on the host and 30-35 GB/s on the coprocessor. For double precision, the optimum tile sizes are 8 and 16, and the performance reaches 20-25 GB/s on the host and 35-40 GB/s on the coprocessor. For tile sizes smaller or greater than the optimum values, the performance quickly decreases. The peaked shape of the performance curve is especially pronounced for the Intel Xeon Phi architecture. This expresses the need for fine tuning of the tiled transposition algorithm on this platform.

The plots in the right-hand side of Figures 1 and 2 show the results for the recursive algorithm. With this algorithm, the window of optimal recursion threshold can be considerably wider. On the host, any threshold value equal to or greater than 16 (single precision) or 4 (double precision) provides the maximum performance of 20-25 GB/s. On the coprocessor, in single precision, values 16, 32 and 64 achieve the performance of 35-40 GB/s, while in double precision, the window of optimal threshold values is between 8 and 512. This illustrates the "cache-oblivious" nature of the recursive algorithm: the tuning parameter needs not be as finely constrained as in the tiled, "cache-aware" method.

With very low values of the tuning parameter (less than 16), the reason for low performance is that the vectorization in the inner loop is not employed fully, because the the length of the loop smaller than the SIMD register capacity. With very large values of the tuning parameter, two reasons contribute to the reduction in performance. First, the cache-efficient nature of the algorithm is not fully exploited, and the application algorithm degrades to the "cache-ignorant", unoptimized transposition. The second reason is that the number of parallel strands is inversely proportional to the tile size (in the tiled algorithm), and inversely proportional to the square of the recursion threshold (in the recursive algorithm). With too few parallel strands to utilize 240 logical cores in Intel Xeon Phi coprocessors, the parallel scalability of the tiled algorithm suffers. In the recursive algorithm, this effect occurs at considerably larger values of the recursion threshold.

For the performance measurements (Section 5), the tile size is chosen as 16 for both single and double precision, and the threshold value as 32. The optimum parameters may not work best for all matrix sizes, and in practical applications, runtime adjustment may be needed. Such runtime calibration approach is sometimes taken in high-performance computing applications that aim to tune the algorithm to the problem size and the actual hardware configuration (e.g., FFTW [4]).

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

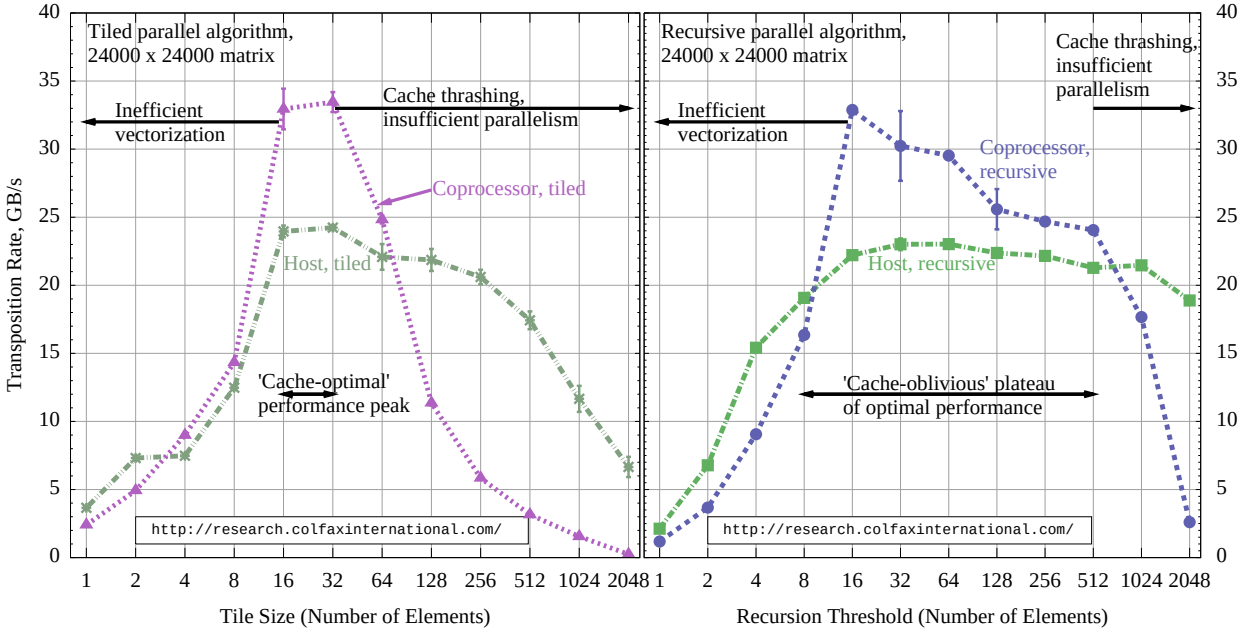


Figure 1: Tiled and recursive transposition performance for a range of values of the tuning parameter, single precision.

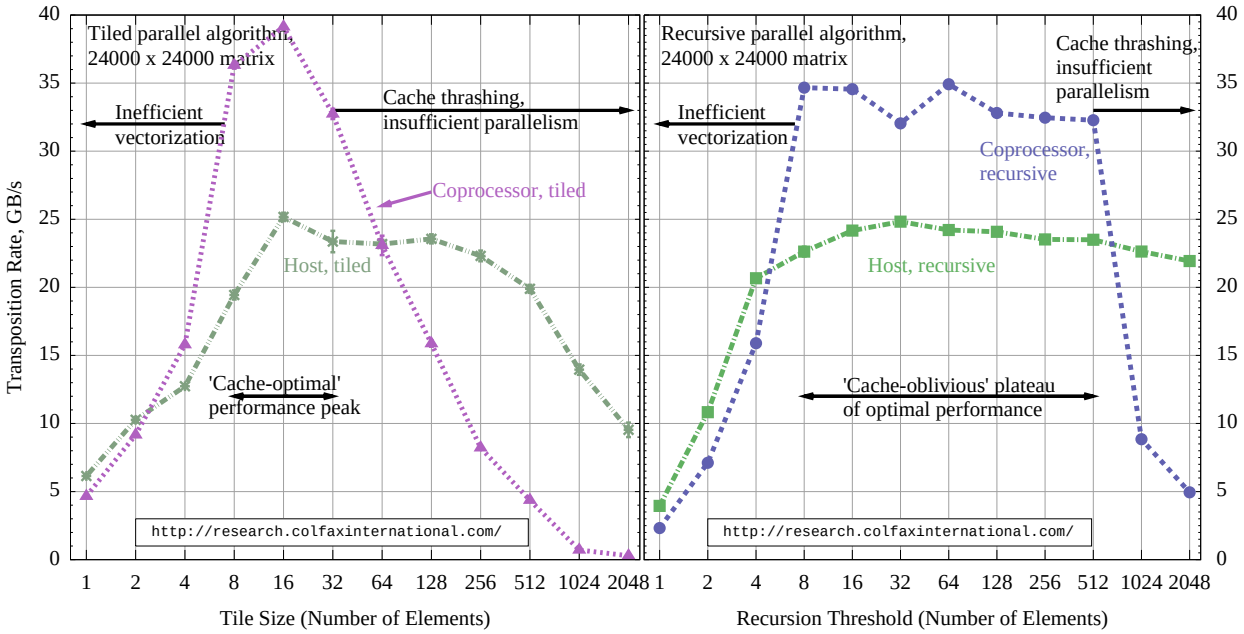


Figure 2: Tiled and recursive transposition performance for a range of values of the tuning parameter, double precision.

5 Performance Results

The system described in Section 4 was also used to benchmark the transposition performance for several matrix sizes. The sizes of the matrices tested are $n \in \{400, 600, 800, 1100, 1600, 2200, 3200, 4500, 6400, 9000, 12800, 18000 \text{ and } 24500\}$. Powers of 2 for the value of `n` were avoided, because the performance in this case is anomalously low. This is related to the finite cache associativity and the least recently used (LRU) replacement policy, which leads to constant fetching and eviction of data to and from the same cache way.

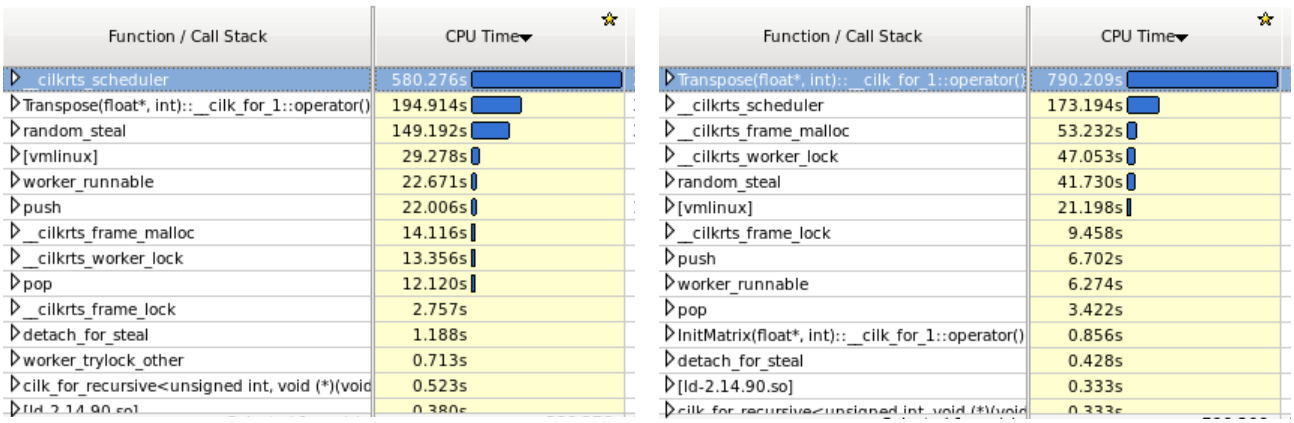
The results are shown in Figures 4 and 5. First of all, the expected result of optimizations is confirmed in the performance plots. That is, for almost all matrix sizes and for all platforms, the optimized codes (tiled as well as recursive) perform better than the unoptimized codes on the same platform. The exception to this trend is small matrices (4000×4000 or less) on the Intel Xeon Phi architecture, for which optimized code performance is approximately the same as for the unoptimized code. This issue is explained at the end of this section.

Comparison between the tiled and recursive implementations shows that the tiled version wins in almost all cases by 10 to 30%. The cases where this is not true are large matrices (greater than 4000×4000) on the host: both algorithms produce approximately the same performance in this case.

The coprocessor produces better performance than the host system only for large matrices (greater than 10000×10000 in single precision or 8000×8000 in double precision). In double precision, the tiled algorithm for the biggest matrix tested works 1.6x faster on the coprocessor than on the host. For small matrices (2000×2000 and less), the host performs up to an order of magnitude faster than the coprocessor.

The reason for poor performance of the coprocessor for small matrices is the parallel scheduling overhead. Figure 3 shows the the result of performance analysis with the Intel VTune Amplifier XE tool. This analysis is performed for the tiled algorithm on the Intel Xeon Phi architecture. Two profiling runs were made: one for a matrix of size 2400×2400 and another for a matrix of size 24000×24000. The number of trials for each matrix size is chosen so that both runs take approximately the same wall clock time. According to the hotspot profile, in the small matrix case (left-hand side), the majority of the CPU time is spent in function `cilkrts_scheduler`. This is, as the name suggest, the scheduling module of the Intel Cilk Plus runtime library. The transposition function `Transpose()` takes less than 25% of execution time in this case. In contrast, for the large matrix (right-hand side), the scheduler takes relatively little time, and the transposition function is the main hotspot.

Relatively large scheduling overhead for the 2400×2400 matrix means that this problem is too small to employ the massively parallel Intel Xeon Phi coprocessor.



Tiled transposition of a 2400×2400 matrix

Tiled transposition of a 24000×24000 matrix

Figure 3: Intel Vtune Amplifier XE analysis for tiled matrix transposition.

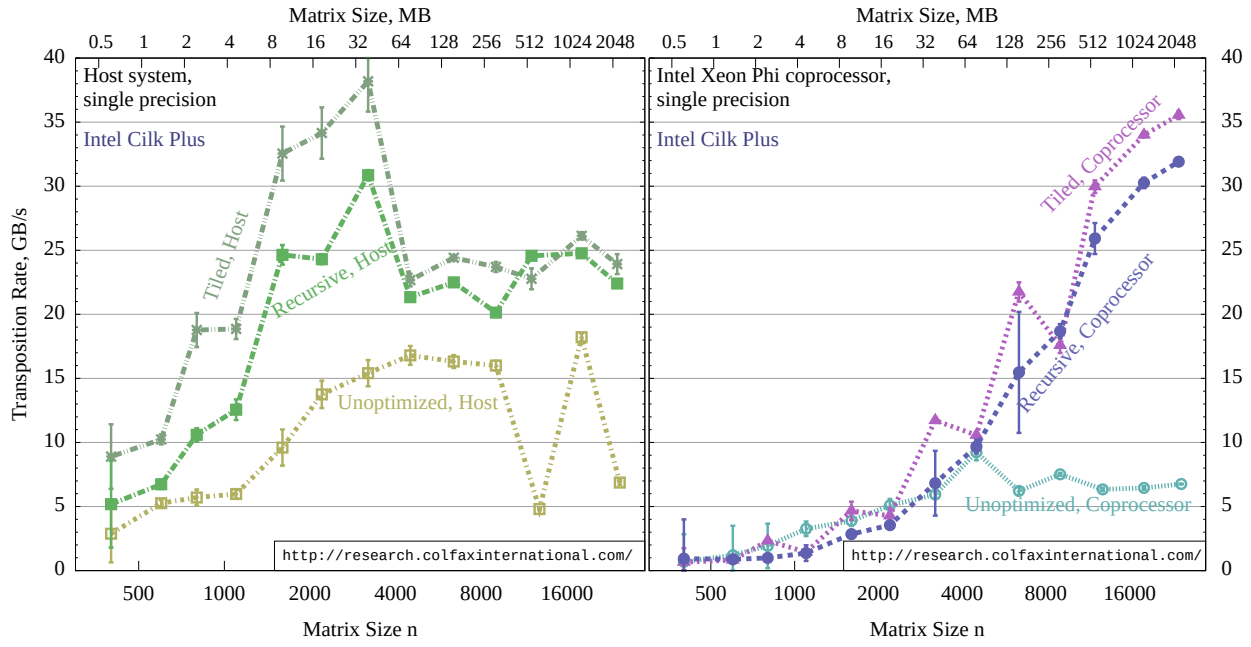


Figure 4: Matrix transposition performance in single precision with the Intel Cilk Plus parallel framework.

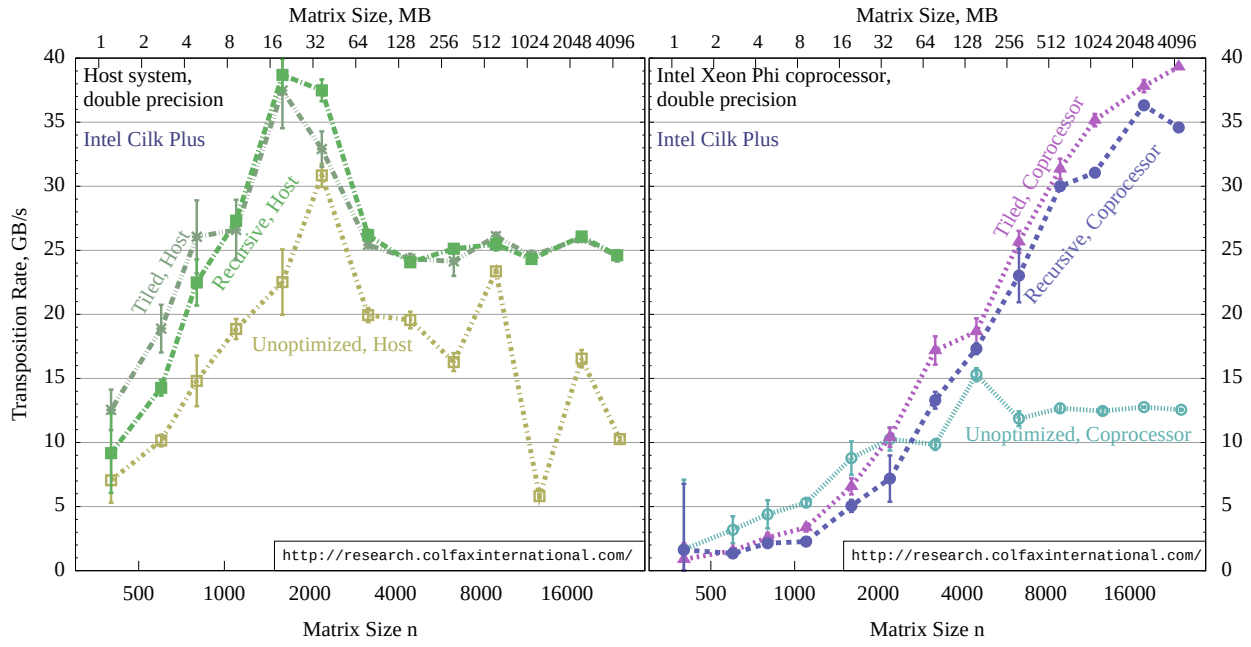


Figure 5: Matrix transposition performance in double precision with the Intel Cilk Plus parallel framework.

6 OpenMP vs Intel Cilk Plus

Considering that the value of the transposition rate for small matrices on the MIC architecture is less than 5 GB/s, it may be more efficient to reverse-offload these matrices to the host, perform transposition there, and return them to the coprocessor. However, from the convenience and performance standpoints, this is usually not a satisfactory solution, and other optimization methods must be considered. The most obvious one is to look for a parallel scheduler with less overhead. For instance, OpenMP should be considered, a common parallel framework available for both general-purpose processors and Intel Xeon Phi coprocessors. OpenMP has schedulers for parallel loops and fork-join algorithms, and therefore it is straightforward to implement the tiled and recursive matrix transposition methods in OpenMP.

The listing below is the tiled matrix transposition algorithm parallelized using OpenMP:

```
1  void Transpose(float* const A, const int n){
2      const int TILE = 16;
3  #pragma omp parallel for schedule(static)
4      for (int ii = 0; ii < n; ii += TILE) {
5          const int iMax = (n < ii+TILE ? n : ii+TILE);
6          for (int jj = 0; jj <= ii; jj += TILE)  {
7              for (int i = ii; i < iMax; i++) {
8                  const int jMax = (i < jj+TILE ? i : jj+TILE);
9  #pragma loop count avg(TILE)
10 #pragma simd
11                 for (int j = jj; j<jMax; j++) {
12                     const float c = A[i*n + j];
13                     A[i*n + j] = A[j*n + i];
14                     A[j*n + i] = c;
15                 }
16             }
17         }
18     }
19 }
```

In order to distribute loop iterations across the system's logical cores, `#pragma omp parallel for` is used. The static scheduling mode is used this case as the regime with the lowest scheduling overhead.

The recursive transposition algorithm can be implemented using the OpenMP tasking functionality in place of `_Cilk_spawn`, as shown in the listing below:

```
1  void transpose_cache_oblivious_thread(
2          const int iStart, const int iEnd,
3          const int jStart, const int jEnd,
4          float* A, const int n){
5
6      const int RT = 16;
7
8      if ( ((iEnd - iStart) <= RT) && ((jEnd - jStart) <= RT) ) {
9          for (int i = iStart; i < iEnd; i++) {
10             int je = (jEnd < i ? jEnd : i);
11 #pragma simd
12 #pragma loop_count avg(RT)
13             for (int j = jStart; j < je; j++) {
14                 const float c = A[i*n + j];
15                 A[i*n + j] = A[j*n + i];
16                 A[j*n + i] = c;
17             }
18         }
19         return;
20     }
21
22     if ((jEnd - jStart) > (iEnd - iStart)) {
23         // Split into subtasks j-wise
24         int jSplit = jStart + (jEnd - jStart)/2;
25         // The following line slightly improves performance by splitting on aligned
```

```
26          // boundaries
27          if (jSplit - jSplit%VECLEN > jStart) jSplit -= jSplit%VECLEN;
28  #pragma omp task firstprivate(iStart, iEnd, jStart, n, A, jSplit)
29            {
30              transpose_cache_oblivious_thread(iStart, iEnd, jStart, jSplit, A, n);
31            }
32          transpose_cache_oblivious_thread(iStart, iEnd, jSplit, jEnd, A, n);
33      } else {
34          // Split into subtasks i-wise
35          int iSplit = iStart + (iEnd - iStart)/2;
36          // The following line slightly improves performance by splitting on aligned
37          // boundaries
38          if (iSplit - iSplit%VECLEN > iStart) iSplit -= iSplit%VECLEN;
39          const int jMax = (jEnd < iSplit ? jEnd : iSplit);
40  #pragma omp task firstprivate(iStart, iEnd, jStart, n, A, iSplit)
41          {
42            transpose_cache_oblivious_thread(iStart, iSplit, jStart, jMax, A, n);
43          }
44          transpose_cache_oblivious_thread(iSplit, iEnd, jStart, jEnd, A, n);
45      }
46  #pragma omp taskwait
47  }
48
49  void Transpose(float* const A, const int n){
50  #pragma omp parallel
51    {
52  #pragma omp single
53      {
54        transpose_cache_oblivious_thread(0, n, 0, n, A, n);
55      }
56    }
57  }
```

The performance of the transposition algorithms implemented in the OpenMP framework can be improved by setting the environment variable `KMP_AFFINITY` to the value `compact,granularity=fine` in order to prevent thread migration across logical cores. With this runtime environment optimization, the benchmark of the recursive transposition method is shown in Figures 6 and 7. The performance of tiled transposition with OpenMP is shown in Figures 8 and 9.

The outcome of this test is mixed. On the one hand, the problem with poor performance on the coprocessor of small matrix transposition is alleviated. Indeed, both recursive and tiled methods with OpenMP achieve 5 to 15 GB/s for matrix sizes below 2000×2000, for which Intel Cilk Plus yields 2 to 7 GB/s. On the other hand, large matrix (over 10000×10000) transposition on the coprocessor is slower with OpenMP than with Intel Cilk Plus.

The host performance in single precision with OpenMP is considerably lower across the board than with Intel Cilk Plus. The same holds for double precision, except for small matrices (smaller than 30 MB in size), which fit in the L2 cache of the host system. For these matrices, with static loop scheduling and fixed thread affinity, the data never leaves the cache, and the performance goes off the chart, exceeding the main memory bandwidth. This result is an accurate performance measurement for applications that perform multiple transpositions of the same small matrix. For applications that perform transpositions of multiple distinct small matrices, this gigantic performance metric is inaccurate.

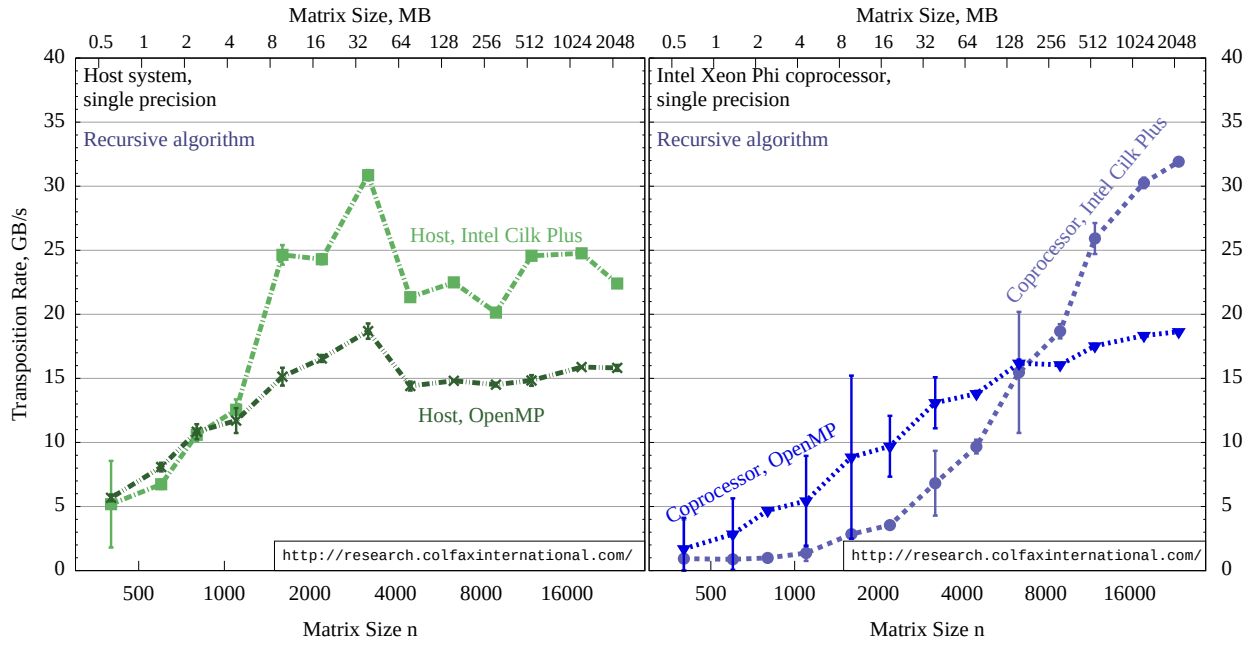


Figure 6: Recursive transposition using OpenMP versus Intel Cilk Plus, single precision.

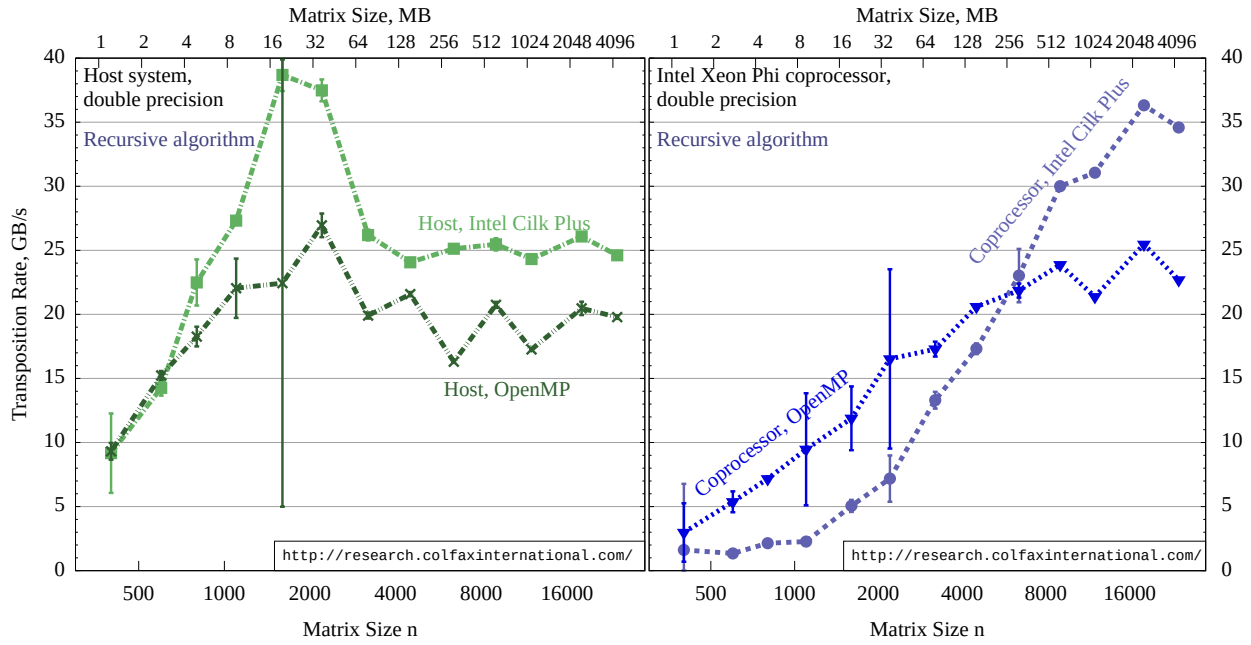


Figure 7: Recursive transposition using OpenMP versus Intel Cilk Plus, double precision.

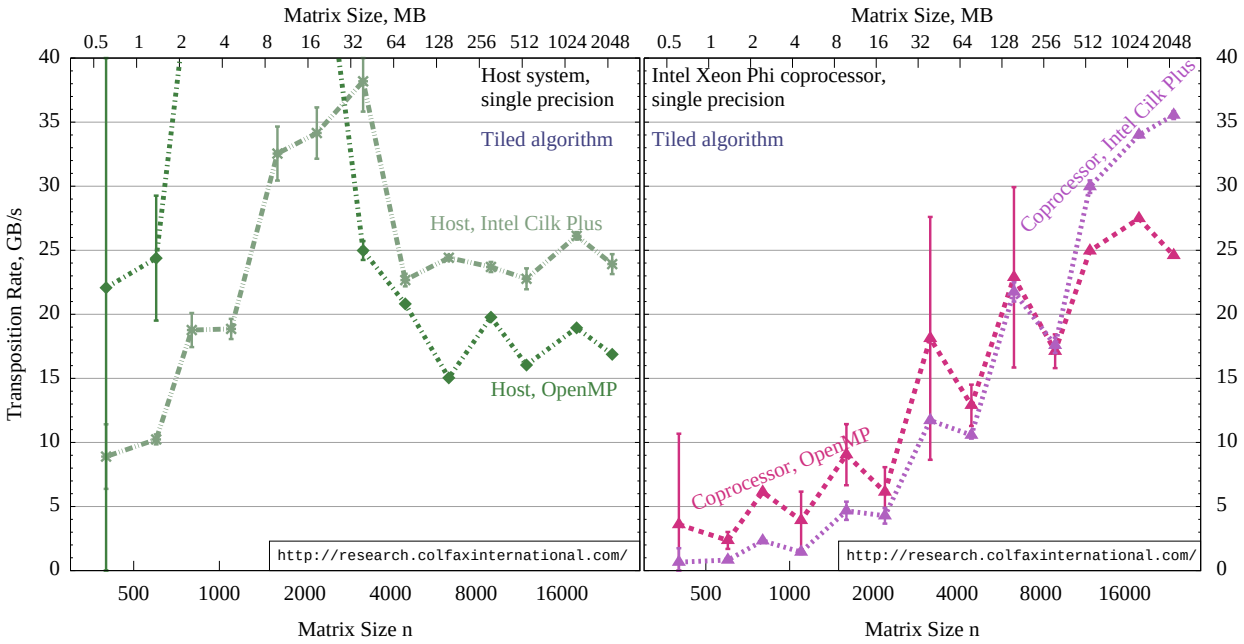


Figure 8: Tiled transposition using OpenMP versus Intel Cilk Plus, single precision.

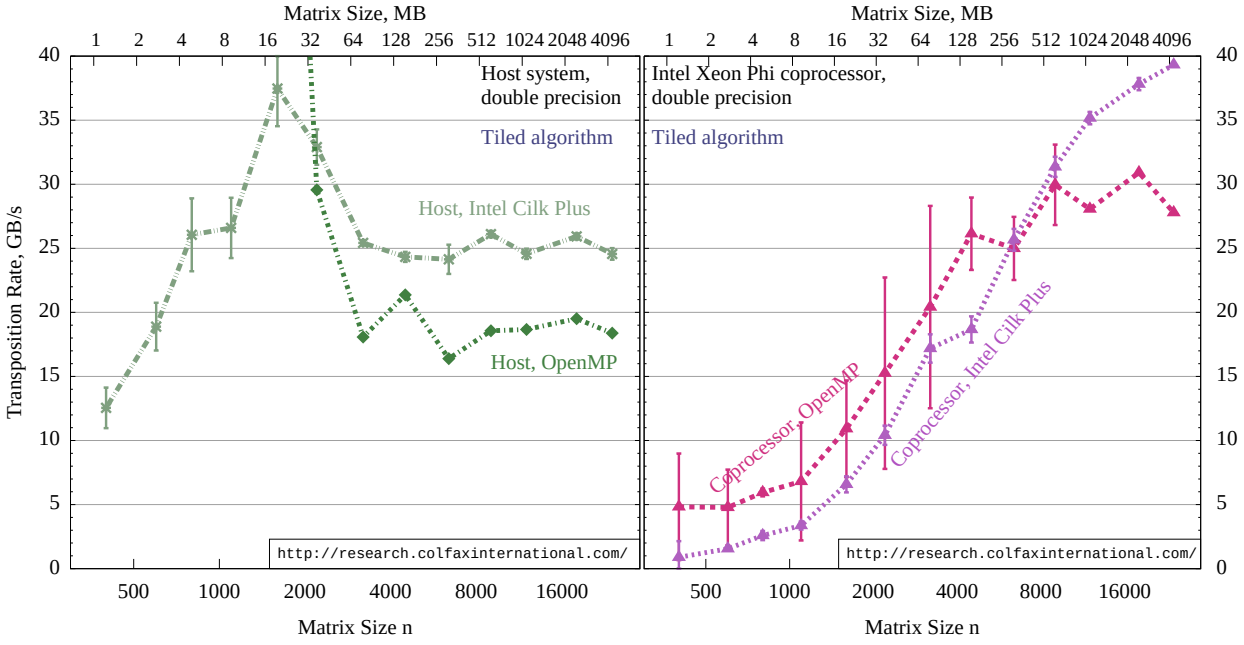


Figure 9: Tiled transposition using OpenMP versus Intel Cilk Plus, double precision.

7 Summary

This study has confirmed that the techniques of loop tiling and recursive divide-and-conquer significantly improve the performance of in-place square matrix transposition on Intel Xeon processors and Intel Xeon Phi coprocessors. Both algorithms must be tuned by testing a range of values of the tile size and the recursive threshold. For the tiled algorithm, the range of optimal values is narrow. The recursive algorithm is more forgiving, exhibiting good performance in with a wider range of parameters. This confirms the "cache-oblivious" nature of this algorithm.

For large matrices, exceeding in size the L2 cache of the system, the absolute value of performance on both platforms corresponds to approximately a quarter to a third of the practical main memory bandwidth: up to 25 GB/s on the host system 40 GB/s on the coprocessor. This performance is achieved by the tiled transposition method expressed in the Intel Cilk Plus parallel framework.

However, for small matrices, the situation is more complicated. For matrices between 10 and 100 MB in size, the host system achieves up to 40 GB/s with the tiled algorithm. However, the performance of the coprocessor is considerably impaired in this matrix size range, between 2 and 10 GB/s. The cause of this performance degradation is the thread scheduling overhead, which is large for a massively parallel system.

The scheduling overhead can be reduced by expressing the transposition algorithms with the OpenMP parallel framework. With thread affinity set to type `compact`, transposition with OpenMP achieves 5 to 20 GB/s for small matrices on the coprocessor with both the recursive and the tiled algorithms.

At the same time, the implementation of *large* matrix transposition on coprocessor with OpenMP yields less performance than the implementation with Intel Cilk Plus. Another win for Intel Cilk Plus is that on the host, it outperforms OpenMP across a wide range of matrix sizes.

Among the algorithms and parallel frameworks considered here, there is no single optimal solution for a universal transposition application for matrices of arbitrary size. However, the choice of transposition methods discussed here allows to achieve reasonable to good performance for a limited range of matrix sizes for both the multi-core and the many-core platform.

Acknowledgements

The study presented here is based on an example from the book "Parallel Programming and Optimization with Intel Xeon Phi Coprocessors" [5].

Please visit http://research.colfaxinternational.com/ to learn more about the Colfax Research project, comment on this article, and subscribe for updates.

References

[1] Harald Prokop. Cache-Oblivious Algorithms. Master's thesis, Massachusetts Institute of Technology, 1999. URL: 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. URL: 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. URL: http://www.springeronline.com/3-540-22115-8.

[4] FFTW Library. URL: http://www.fftw.org/.

[5] Colfax International. *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*. Colfax International, 2013. URL: http://colfax-intl.com/xeonphi/book.html.