Fine-Tuning Vectorization and Memory Traffic on Intel Xeon Phi Coprocessors: LU Decomposition of Small Matrices

Andrey Vladimirov
Colfax International

January 27, 2015

Abstract

Common techniques for fine-tuning the performance of automatically vectorized loops in applications for Intel Xeon Phi coprocessors are discussed. These techniques include strength reduction, regularizing the vectorization pattern, data alignment and aligned data hint, and pointer disambiguation. In addition, the loop tiling technique of memory traffic tuning is shown. The optimization methods are illustrated on an example of single-threaded LU decomposition of a single precision matrix of size 128×128.

Benchmarks show that the discussed optimizations improve the application performance on the coprocessor by a factor of 2.8 compared to the unoptimized code, and by a factor of 1.7 on the multi-core host system, achieving roughly the same performance on the host and on the coprocessor.

The code discussed in the paper can be freely downloaded from the Colfax Research Web site.

Table of Contents

1 Intel Xeon Phi Coprocessors, Automatic Vectorization and Future-Proofing . . . 2

2 The Doolittle Algorithm of LU Decomposition . 2

2.1 Numerical Method . . . 2

2.2 Unoptimized Implementation . . . 3

3 Benchmark Methodology . . . 3

3.1 Computing System . . . 3

3.2 Performance Measurement . . . 3

4 Optimization . . . 4

4.1 Clues for Optimization . . . 4

4.2 Strength Reduction . . . 4

4.3 Regularizing Vectorization Pattern . . . 5

4.4 Alignment and Aligned Data Hint . . . 6

4.5 Pointer Disambiguation . . . 7

4.6 Memory Traffic Tuning . . . 7

5 Results and Discussion . . . 9

5.1 Single Code Base . . . 9

5.2 Prior Art: Intel MKL . . . 9

5.3 Value of Intel Xeon Phi with 1.0x Acceleration 10

Colfax International (http://www.colfax-intl.com/) is a leading provider of innovative and expertly engineered workstations, servers, clusters, storage, and personal supercomputing solutions. Colfax International is uniquely positioned to offer the broadest spectrum of high performance computing solutions, all of them completely customizable to meet your needs - far beyond anything you can get from any other name brand. Ready-to-go Colfax HPC solutions deliver significant price/performance advantages, and increased IT agility, that accelerates your business and research outcomes. Colfax International's extensive customer base includes Fortune 1000 companies, educational institutions, and government agencies. Founded in 1987, Colfax International is based in Sunnyvale, California and is privately held.

1. Intel Xeon Phi Coprocessors, Automatic Vectorization and Future-Proofing

Computing accelerators marketed as Intel Xeon Phi coprocessors are based on the Intel Many Integrated Core (MIC) architecture, which may yield better performance than general-purpose multi-core CPU architectures for compute-bound or memory bandwidth-bound applications. The first generation the Intel MIC architecture available today is based on the Knights Corner (KNC) chip, which supports a vector instruction set named Initial Manycore Instructions (IMCI). The second generation, currently in development, will be based on the Knights Landing (KNL) chip, which will support a different instruction set, Advanced Vector Extensions 512 (AVX-512). AVX-512 is not a superset of IMCI, and therefore application codes that explicitly use processor instructions will port from KNC to KNL.

At the same time, in practice, it is often unnecessary to express performance-critical code with explicit vector instructions via assembly or intrinsics. That is because Intel C, C++ and Fortran compilers have extensive support for automatic loop vectorization. That said, a high-level language code that vectorizes and performs well on KNC can be re-compiled to run on KNL, which relieves the developer of the burden of porting to a new instruction set.

However smart compilers may be, they still must operate with uncertainties and possible inefficiencies of the user code. For instance, loads and stores from memory to vector registers in KNC must be performed on 64-byte aligned addresses. If the user code does not guarantee such alignment, the compiler must do what it can to accommodate different runtime situations. Similarly, vectors in KNC are 512 bit wide, which corresponds to short vectors of 16 single precision or 8 double precision numbers. If the length of the user's loops is not a multiple of 16 or 8, the compiler must work around that, possibly losing performance.

The programmer can follow certain guidelines to assist the compiler in its job of producing high-performance executable. The present paper discusses some of the commonly required techniques for fine-tuning high-level language code that result in better performance on coprocessors, as well as on processors. This not only improves performance of applications on the current generation Intel Xeon processors and Intel Xeon Phi coprocessors, but also prepares the code for future architectures, such as KNL.

2. The Doolittle Algorithm of LU Decomposition

2.1. Numerical Method

To demonstrate vectorization tuning techniques, I will use as an example the Doolittle algorithm of LU decomposition. The purpose of this algorithm is to represent a square, non-degenerate matrix A as a product $A = LU$, where L is a unit lower triangular matrix, and U is an upper triangular matrix. Such a decomposition is commonly used to solve systems of linear algebraic equations.

The Doolittle algorithm applied to an $n \times n$ matrix performs $n-1$ iterations generally resembling the Gaussian elimination scheme. For iteration b, matrix row b is multiplied by a certain factor and added to matrix rows $b+1$ through $n-1$ so that the in the resulting matrix $A^{(b)}$, all elements in column b starting from $b+1$ are equal to zero. The coefficients of that multiplication are recorded in a separate matrix L:

$$A^{(0)} = A, \quad (1)$$

$$A^{(b+1)} = L^{(b)} A^{(b)}, \text{ where} \quad (2)$$

$$L_{ij}^{(b)} = \begin{cases} 1, \text{ if } i = j, \\ l_{i,b}, \text{ if } i > j \text{ and } j = b, \\ 0, \text{ otherwise,} \end{cases} \quad (3)$$

$$l_{i,b} = -\frac{A_{i,b}^{(b)}}{A_{b,b}^{(b)}}. \quad (4)$$

As a result, $U \equiv A^{(n-1)}$, and elements of L are

$$L_{i,j} = \begin{cases} 1, \text{ if } i = j, \\ -l_{i,j}, \text{ if } i < j, \\ 0, \text{ otherwise.} \end{cases} \quad (5)$$

2.2. Unoptimized Implementation

A simplified implementation of this procedure is shown in Listing 1. It is simplified because the algorithm does not involve pivoting (i.e., choosing the best row to eliminate elements in other rows) and is not optimized.

```
 6 void LU_decomp(const int n, float* const A) {
 7   // LU decomposition (Doolittle algorithm)
 8   // In-place decomposition of form A=LU
 9   // L is returned below main diagonal of A
10   // U is returned at and above main diagonal
11   for (int b = 0; b < n; b++) {
12     for (int i = b+1; i < n; i++) {
13       A[i*n + b] = A[i*n + b]/A[b*n + b];
14       for (int j = b+1; j < n; j++)
15         A[i*n + j] -= A[i*n + b]*A[b*n + j];
16     }
17   }
18 }
```

Listing 1: LU decomposition, unoptimized.

Our function `LU_decomp()` performs LU decomposition in-place, returning the matrix L (except its unit main diagonal) in the space below the main diagonal of the input matrix A, and returning U at and above the main diagonal of A. This is the traditional approach taken, for example, by the LAPACK implementation of LU decomposition, `?getrf()`.

The algorithm in Listing 1 is single-threaded, and we are going to keep it this way: the assumption here is that this function is called from a parallel region to process multiple independent matrices concurrently as shown in Listing 2. This is useful in applications processing multiple small systems of linear algebraic equations.

```
138 const double tStart = omp_get_wtime();
139 #pragma omp parallel for
140 for (int m = 0; m < nMatrices; m++) {
141   float* matrixA=(float*)(&dataA[m*ctrSize]);
142   LU_decomp(n, matrixA);
143 }
144 const double tEnd = omp_get_wtime();
```

Listing 2: Calling and timing the single-threaded LU decomposition function from a parallel region to concurrently process multiple independent matrices.

3. Benchmark Methodology

3.1. Computing System

All of the benchmarks presented in this section were taken on a Colfax ProEdge™ SXP8600 workstation based on a two-way Intel Xeon E5-2697 v2 processor (12 cores per socket, 24 cores total) with 128 GB DDR3 of RAM at 1600 MHz. The system contains four Intel Xeon Phi 7120P coprocessors (only one was used for benchmarks). Each coprocessor contains 61 active cores (4 hardware threads per core) clocked at 1.24 GHz and 16 GB of GDDR5 memory. The code was compiled using the Intel C++ compiler version 15.0.1.133 and run under MPSS 3.4.1 on CentOS 7.0 Linux.

3.2. Performance Measurement

The full code discussed in the paper, with snapshots for each optimization step, can be freely downloaded from the Colfax Research Web site [1].

The code for the CPU was compiled with the arguments `-qopenmp -xhost`, and for the coprocessor with arguments `-qopenmp -mmic`. To run the code CPU, the thread affinity pattern was set to "scatter" using the environment variable `KMP_AFFINITY`. This was not necessary for the coprocessor because the Intel OpenMP library for the MIC architecture effects this type of affinity by default. To run the native executable on the coprocessor, an SSH session was used. On the 24-core CPU, 2 threads per core were used, and on the 61-core coprocessor – 4 threads per core. So 48 matrices were concurrently processed on the CPU and 244 on the coprocessor.

Performance was measured by timing the parallel loop that performs LU decomposition of 10^4 matrices of size 128×128 as shown in Listing 2. The timing was repeated 10 times; the first two measurements were discarded and the subsequent ones were averaged. The first one or two measurements tend to be slower on the coprocessor, and do not reflect the sustained performance. The execution time was translated into cumulative performance measured in GFLOP/s using a conversion factor that assumes that each LU decomposition requires $(2/3)n^3$ operations, where $n = 128$ is the matrix size.

4. Optimization

Benchmarking the unoptimized code from Listing 1 yielded the following baseline performance: 140.0 ± 0.6 GFLOP/s on the host and 86.3 ± 0.1 GFLOP/s on the Intel Xeon Phi coprocessor. Apparently, the inefficiencies in this code hamper the Intel MIC architecture much more than the multi-core Intel Xeon CPU.

4.1. Clues for Optimization

Areas for potential optimization can be inferred from the optimization report produced by the compiler. Listing 3 shows the relevant parts of the report produced with the argument `-qopt-report=4`.

```
LOOP BEGIN at main.cc(14,7) inlined into main.cc
<Peeled, Multiversioned v1>
   ...reference matrixA has unaligned access...
   ...PEEL LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at main.cc(14,7) inlined into main.cc
<Multiversioned v1>
   ...Loop multiversioned for Data Dependence
   ...reference matrixA has aligned access...
   LOOP WAS VECTORIZED
   ...
LOOP END

...

LOOP BEGIN at main.cc(14,7) inlined into main.cc
<Remainder, Multiversioned v1>
   ...reference matrixA has aligned access...
   ...reference matrixA has unaligned access...
   REMAINDER LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at main.cc(14,7) inlined into main.cc
<Multiversioned v2>
   loop was not vectorized: non-vectorizable
              loop instance from multiversioning
   ...
LOOP END
```

Listing 3: Optimization report for Listing 1

Indeed, the report indicates that the compiler processed different versions of the loop targeted to the different possible alignment and pointer aliasing situations at runtime. Also, code for processing the peel and remainder of the loop was generated.

Multi-versioning and uncertain alignment are hampering the performance of this code, and in the subsequent sections will demonstrate how to deal with it.

4.2. Strength Reduction

Before we even start following up on the optimization report, we can reap a low-hanging fruit by implementing an optimization known as strength reduction.

Line 13 in Listing 1 contains a division, and the denominator in this expression is the same in every iteration in `i`. We can take advantage of this repetitive pattern and replace division with multiplication, which is a lower-latency operation (see, e.g., [2]). The corresponding snippet of the code is shown in Listing 4.

```
11   for (int b = 0; b < n; b++) {
12     // Strength reduction:
13     const float recAbb = 1.0f/A[b*n + b];
14     for (int i = b+1; i < n; i++) {
15       A[i*n + b] = A[i*n + b]*recAbb;
16       for (int j = b+1; j < n; j++)
17         A[i*n + j] -= A[i*n + b]*A[b*n + j];
18     }
19   }
```

Listing 4: LU decomposition with strength reduction.

With this optimization, the performance on the host increased only marginally, to 144.2 ± 0.4 GFLOP/s, however, on the coprocessor, the performance increased by 12% to 96.5 ± 0.1 GFLOP/s.

Replacement of division with multiplication by the reciprocal value is an example of strength reduction. This is a class of optimizations where expensive operations are replaced with less expensive, approximately equivalent operations.

Due to the limitations of finite-precision arithmetics, the programmer must always consider whether the stability and accuracy requirements of the algorithm permit such optimizations.

With strength reduction done, we can proceed to more subtle optimizations for which we have hints from the optimization report.

4.3. Regularizing Vectorization Pattern

Vector processing units (VPUs) of Intel Xeon Phi coprocessors can process vector instructions on vectors of exactly 16 or 8 elements in single and double precision, respectively. Nevertheless, the Intel C++ compiler is able to vectorize loops for which loop count is not known at compilation time. Furthermore, load and store instructions in the KNC architecture can operate only on 64-byte aligned memory regions. However, this does not prevent the Intel C++ compiler from vectorizing loops in which the alignment of element is not known at compilation time.

In cases where the loop count or alignment situation is not known at compilation time, the compiler implements a runtime check of alignment and loop count. Depending on the result, the code may peel off a few iterations at the beginning of the loop, or process the tail of the loop with scalar or masked instructions. Naturally, a loop with beginning or tail peeled off is less efficient than a loop with only vector iterations. Peeling is illustrated in Figure 1. "Code Path 1" and "Code Path 2" may be taken at runtime depending on the length of the loop and on the data alignment situation at runtime.

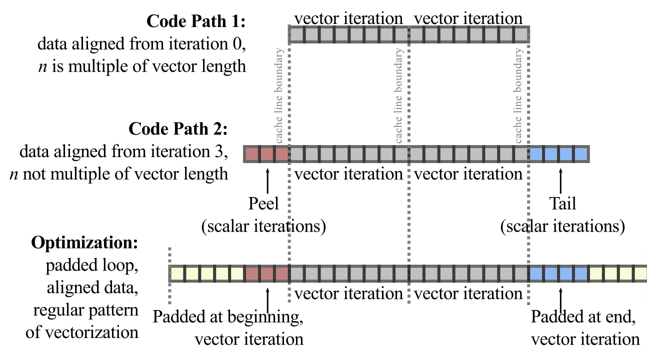

Figure 1: Compiler may peel an irregular loop. To prevent that, the programmer may regularize the vectorization pattern.

The programmer can alleviate the load on the platform by regularizing the pattern of vectorization, so that the operations are always vector and always on aligned data. This may require three measures:

1. Padding loop count to a multiple of vector length
2. Padding data to a multiple of vector length
3. Aligning data on the appropriate boundary

This paper refers to optimizations that lead to the elimination of loop peel and tail as "regularizing the vectorization pattern". The regularized loop may have more iterations, where in marginal iterations the processor crunches dummy data (see Figure 1, case "Optimization"). However, the performance of regular loops on the Intel MIC architecture is generally better than that of irregular loops, because the peeled iterations take more clock cycles than a single vector iteration.

In our code (Listing 1), the inner loop in `j` has different loop counts in every iteration in `b`. Additionally, it sometimes begins on an aligned data element, other times it does not. To regularize this loop, instead of starting the inner loop from `j=b+1`, let's start it from `jMin`, which is the greatest multiple of the vector/alignment length not exceeding `b+1`. Our procedure assumes that the matrix size, `n`, is also a multiple of the vector length. The resulting code snippet is shown in Listing 5, and explanation of additional measures taken in this code are provided below.

```
12 #ifdef __MIC__
13   const int tile=32;
14 #else
15   const int tile=8;
16 #endif
17   assert(n%tile==0);
18   // Must store L separately from A
19   float L[n*n] __attribute__((aligned(64)));
20   for (int i = 0; i < n; i++) {
21     L[i*n:n]=0.0f;
22     L[i*n+i]=1.0f;
23   }
24   for (int b = 0; b < n; b++) {
25     const int jMin = b - b%tile;
26     // Strength reduction:
27     const float recAbb = 1.0f/A[b*n + b];
28     for (int i = b+1; i < n; i++) {
29       L[i*n + b] = A[i*n + b]*recAbb;
30       // Regularized pattern of vector loop:
31       for (int j = jMin; j < n; j++)
32         A[i*n + j] -= L[i*n+b]*A[b*n + j];
33     }
34   }
35   // Copy temp matrix L into matrix A
36   for (int i = 0; i < n; i++)
37     for (int j = 0; j < i; j++)
38       A[i*n + j] = L[i*n + j];
```

Listing 5: LU decomposition, regularized vectorization pattern.

With the `j`-loop beginning at `jMin`$\leq$`b`, we have to deal with the fact that now the loop in `j` will overwrite

some of the elements of A. In order to retain correctness, a temporary storage matrix L was introduced, which is stored separately from A during the calculation, and is copied into A before the function returns.

There is also freedom to choose the value into which jMin must divide. Experiments established that on the CPU, a multiple of 8 works well, while on the coprocessor, multiples of 32 are better. To optimize the code, target-specific tuning can be done using the preprocessor macro __MIC__.

With regularized vectorization pattern, the performance on the host increased 20% to 173.5 ± 0.1 GFLOP/s and on the coprocessor it jumped 30% to 126.0 ± 0.3 GFLOP/s.

4.4. Alignment and Aligned Data Hint

Data alignment on a 64-byte boundary is required for vector instructions in the Many Integrated Core architecture of Intel Xeon Phi coprocessors. The alignment of the first element in a pointer-based array is generally not known at compile time. Therefore, in automatically vectorized loops, the compiler must implement a check for alignment. Depending on the results of the check, the application may peel off several iterations at the beginning of the loop in order to reach the first aligned element. The check may take a significant portion of the loop calculation time, especially for short loops, and multiple versions of the code required for execution take up space in the instruction cache.

If the programmer can guarantee that pointer-based arrays in a vectorized loop are aligned, it is beneficial inform the compiler of this. This is done using #pragma vector aligned.

This pragma is applicable to our case. Indeed, in the code the memory buffer for matrix A is allocated on a 64-byte aligned boundary using the allocator _mm_malloc(). Matrix L, placed on the stack, is also allocated on a 64-byte boundary thanks to the declaration __attribute__((aligned(64))) (see line 19 of Listing 5). Additionally, the code checks that n is a multiple of 16, which amounts to 64 bytes, and begins the loop in j on jMin, which is a multiple of 32 bytes on host or 128 bytes on coprocessor. Therefore, all arrays in line 32 of Listing 5 are aligned. However, according to the optimization report in Listing 3 (or the respective line in optimization report for Listing 5), the compiler implements code that would work with both aligned and unaligned accesses to matrix A.

Applying the aligned data hint (#pragma vector aligned) to our LU decomposition results in the following (the relevant snippet is shown below).

```
28      for (int i = b+1; i < n; i++) {
29        L[i*n + b] = A[i*n + b]*recAbb;
30        // Aligned data hint:
31  #pragma vector aligned
32        // Regularized pattern of vector loop:
33        for (int j = jMin; j < n; j++)
34          A[i*n + j] -= L[i*n+b]*A[b*n + j];
35      }
```

Listing 6: Aligned data hint in the LU decomposition code.

With #pragma vector aligned, the compiler report indicates only aligned accesses to matrix A (see Listing 7).

```
LOOP BEGIN at main.cc(33,7) inlined into main.cc
<Multiversioned v1>
   Loop multiversioned for Data Dependence
   ...reference matrixA has aligned access...
   LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at main.cc(33,7) inlined into main.cc
<Remainder, Multiversioned v1>
   ...reference matrixA has aligned access...
   remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at main.cc(33,7) inlined into main.cc
<Multiversioned v2>
   loop was not vectorized: non-vectorizable
              loop instance from multiversioning
LOOP END
```

Listing 7: Optimization report for Listing 6

Due to this optimization with aligned data hint, the performance on the host slightly decreased to 164.1 ± 0.1 GFLOP/s, but on the coprocessor it increased by 3% to 142.4 ± 0.3 GFLOP/s.

Optimization with the aligned data hint is likely to be important for relatively short loops with lightweight operations, because in longer loops, the relative amount of time used for runtime checks is smaller.

4.5. Pointer Disambiguation

Another hint given to us by the optimization report of the LU decomposition code (Listing 7) is that the compiler instrumented multiversioning. This occurred because at compilation time, it is not certain whether pointers `L` and `A` are aliased (i.e., pointing to the same memory regions) or not.

We can gain additional performance by eliminating multiversioning. This can be using `#pragma ivdep`. This pragma instructs the compiler to ignore assumed vector dependencies, and assume all pointers in the loops to be non-aliased.

The result of this optimization in shown in Listing 8. The new pragma can be combined with the pragma used in the previous step.

```
28      for (int i = b+1; i < n; i++) {
29        L[i*n + b] = A[i*n + b]*recAbb;
30        // Aligned data hint:
31  #pragma vector aligned
32        // Pointer disambiguation:
33  #pragma ivdep
34        // Regularized pattern of vector loop:
35        for (int j = jMin; j < n; j++)
36          A[i*n + j] -= L[i*n+b]*A[b*n + j];
37      }
```

Listing 8: Pointer disambiguation in the LU decomposition code.

Once assumed vector dependence was dropped, the performance on the CPU was bumped by around 4% to 174.7 ± 0.1 GFLOP/s and on the coprocessor it increased by 10% to 157 ± 1 GFLOP/s. Furthermore, the reader may check that the optimization report no longer includes information about multiversioning, which means that only one code path was implemented, with non-aliased `L` and `U`.

An alternative to `#pragma ivdep` is the qualifier `restrict`, which is useful when some pointers in the loop are actually aliased (see [3]). Qualifier `restrict` can be used to disambiguate only non-aliased pointers, and this may be sufficient information for the compiler to implement a more optimal code path.

It is important to remember that when `#pragma ivdep` is used, the penalty for supplying aliased pointers may be incorrect results, without a warning or error message.

4.6. Memory Traffic Tuning

With vectorization optimized as done above, the CPU still outperforms the coprocessor, albeit by a small margin. Hints for further improvement may be obtained from performance analysis with Intel VTune amplifier. Figure 2 shows a screenshot of the summary screen for performance analysis of the last code with pointer disambiguation on the coprocessor.

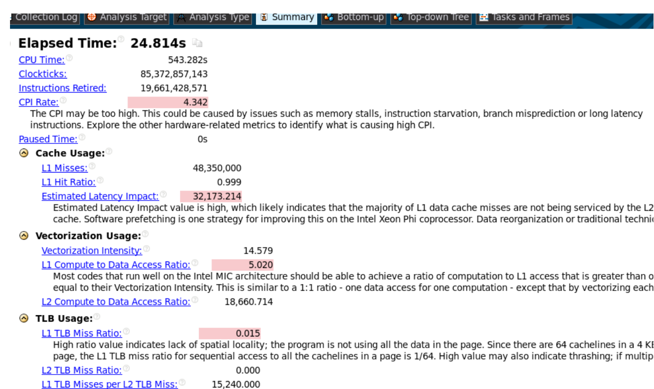

Figure 2: Performance analysis with VTune.

The metrics that VTune finds suspicious is “L1 compute to data access ratio” and the related metric “estimated latency impact”. Both of these metrics point to insufficient arithmetic intensity, i.e., data fetched into caches is not used for computation a sufficient number of times. This situation can usually be resolved with a technique called loop tiling.

Loop tiling transforms two (or more) nested loops into three (or more) nested loops in such a way that a data element used once is re-used as soon as possible. This increases the temporal locality of data access. The basic form of loop tiling is shown in Listing 9.

```
1  // Before tiling:
2  for (int i = 0; i < n; i++)
3    for (int j = 0; j < n; j++)
4      Compute(A[i], B[j]);
5  // Tiled i-loop:
6  const int tile = 16;
7  for (int jj = 0; jj < n; jj+=tile)
8    for (int i = 0; i < n; i++)
9      for (int j = jj; j < jj+tile; j++)
10       Compute(A[i], B[j]);
```

Listing 9: Loop tiling.

Loop tiling changes the order of operations, but does not necessarily change their nature (unless vectorization is involved). The purpose of tiling as shown in Listing 9 is to re-use `B[jj]` every `tile=16` iterations rather than every `n` iterations. This ensures that the re-used elements of `B` are still in a cache. Indeed, after `n` iterations, the beginning of the array `B` may be evicted from cache, depending on the size of the elements and of the cache.

For our LU decomposition, tiling is not trivial, because one has to ensure correctness. The Doolittle algorithm has sequential nature in that it cannot start eliminating elements in line `i` with elements in line `b+1` until it has done it with elements in line `b`. Therefore, opportunities for changing the order of operations in `b` are limited, and correctness check must be performed.

Additionally, the inner loop in variable `j` conveniently has unit-stride access favorable for vectorization and bandwidth. So it is best to retain unit-stride in `j`, which further restricts the types of tiling that we can utilize.

Tiling the loop in `b` leads us to the final optimized version of the code. A snippet of the performance-critical set of loops is shown Listing 10. Arriving to this code required some experimentation. First, the variable for tiling had to be chosen (`b` seems to work best). After that, the order of size of the tile had to be tuned (`btile=16` for MIC and `btile=8` for CPU seem to provide the best results). Finally, the order of tiled loops in `i`, `b` and `j` must be chosen from a total of $3! = 6$ options (experiments show that `jib` works best for MIC and `ijb` is better for the CPU).

In addition to the tuning described above, tiled code has a few additional complications. First, it may be necessary to manually peel off some of the iterations to maintain correctness. Second, vectorization must be retained in the dimension that is accessed with unit stride. In our case, this is the `j`-dimension. In order to vectorize the `j`-loop, which, after tiling, is no longer inner, the compiler hint `#pragma simd` was used.

After memory traffic optimization, performance on the host increased by 34% to 233 GFLOP/s, and on the coprocessor by 56% to 244 GFLOP/s. At this point, the coprocessor marginally outperforms the host.

```
6  #ifdef __MIC__
7    const int tile=32;  // Tuning parameters
8    const int btile=16; // for MIC
9  #else
10   const int tile=16;  // Tuning parameters
11   const int btile=8;  // for CPU
12 #endif
13
14   // ...header code skipped...
15
16   // Tiling in b allows to eliminate line i
17   // using several lines b, facilitating
18   // cached data re-use for b-lines
19   for (int bb = 0; bb < n; bb += btile) {
20     // ...skipped loops that compute the
21     // L-factors. These loops do not take
22     // a significant fraction of time...
23
24     // This block uses the L-factors
25     // to eliminate the bulk of the i-lines
26     // #pragma simd vectorizes the j-loop
27     // rather than the i- or b-loop
28 #ifdef __MIC__
29 #pragma vector aligned
30 #pragma ivdep
31 #pragma simd
32     for (int j = jMin+tile; j < n; j++)
33       for (int i = bb+btile; i < n; i++)
34         for (int b = bb; b < bb+btile; b ++)
35           A[i*n + j] -= L[i*n + b]*A[b*n + j];
36 #else
37     for (int i = bb+btile; i < n; i++)
38 #pragma vector aligned
39 #pragma ivdep
40 #pragma simd
41       for (int j = jMin+tile; j < n; j++)
42         for (int b = bb; b < bb+btile; b ++)
43           A[i*n + j] -= L[i*n + b]*A[b*n + j];
44 #endif
45   }
```

Listing 10: Fragment of LU decomposition, tiled.

Because the purpose of tiling is to fit certain portions of the data set into the processor's cache, the strategy of tiling may vary with the problem size. This includes the choice of the tiled variable or variables and order of loops. In our case, tiling was targeted for matrix size 128×128, which are 64 KB in size. This is greater than the level 1 cache, but smaller than the level 2 cache size per core on both Xeon and Xeon Phi.

Cache-oblivious algorithms [4, 5] may yield a code less sensitive to problem size and the properties of the computing platform. A cache-oblivious algorithm is non-trivial to implement for LU decomposition because of its partially sequential nature.

5. Results and Discussion

5.1. Single Code Base

Performance summary of LU decomposition at different optimization stages is shown in Figure 3.

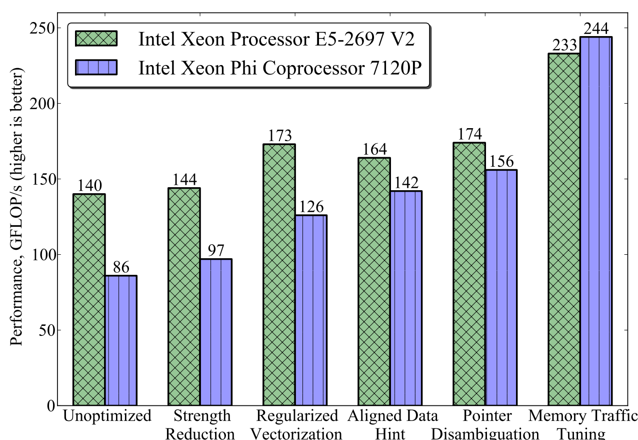

Figure 3: Summary of LU decomposition performance.

It is evident that in all cases but one, the optimization methods discussed here had positive impact on performance both on the host and on the coprocessor. The optimizations performed were all constrained to high-level language programming and compiler hints:

1. strength reduction replaces expensive operations with less expensive;
2. regularizing the vectorization pattern requires changing the loop iteration count;
3. compiler hint on data alignment is a one-line comment-like pragma;
4. another compiler hint for pointer disambiguation is also a pragma;
5. memory traffic tuning requires changing the order of operations in loops.

We ended up with a single code (except for platform-specific tuning parameters and different order of loops), which may be used with high performance both on the multi-core and the manycore platforms. In this sense, the educational goal of the paper is achieved. However, to put the result in context, let us also discuss the absolute value of achieved performance.

Two questions that remain to be clarified are: (i) how do the achieved results compare to results with other optimization methods, and (ii) are the performance results "good enough"?

5.2. Prior Art: Intel MKL

In order to assess the absolute performance achieved, we can use the LU decomposition function from the Intel Math Kernel Library (MKL), which is implemented as a LAPACK function `sgetrf`. The MKL implementation is more general and complex than ours. First, it is tuned for a greater range of matrix sizes, and second, it may use pivoting (interchanging rows to reduce rounding errors). However, for clarity of experiment, the benchmark code uses a diagonally-dominated matrix, for which pivoting does not kick in.

The performance of MKL is compared to the performance of this work in Figure 4.

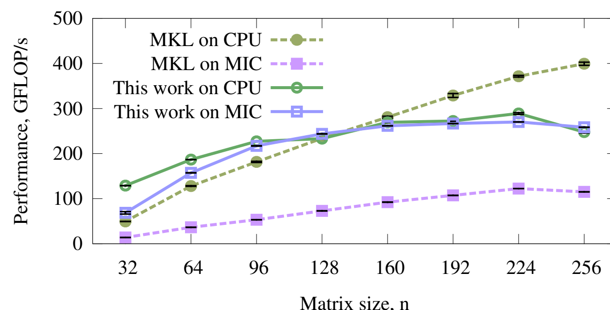

Figure 4: LU decomposition: this work and Intel MKL. Reported performance is for 48 concurrent single-threaded decompositions on a 24-core Intel Xeon E5-2697 V2 processor, and 244 concurrent decompositions on a 61-core Intel Xeon Phi 7120P coprocessor.

This figure leads to two conclusions:

1. On the host's Intel Xeon CPU, our high-level language code performs on par with the industry-leading MKL implementation at our target matrix size 128×128. For smaller matrices, our code is actually more efficient than MKL, however, for larger matrices, MKL performs better. Indeed, optimization techniques such as loop regularization are only important for short loops; for longer loops, other strategies may be used, such as tiling the `j`-loop or multiple loops.

2. On the Intel Xeon Phi coprocessor, the MKL implementation loses by a large factor to both the MKL code on the CPU, and to our high-level language code on CPU and coprocessor. It indicates that the MKL code, likely hand-tuned with explicit assembly or intrinsics, is not portable to the MIC architecture, while our high-level language approach with tuning for the CPU also results in high performance on the coprocessor. While the MKL developers have yet to approach optimizing `?getrf()` for Intel Xeon Phi coprocessors, we developed for two platforms *almost* for the effort of one. The word "*almost*" is used because we did tune the sizes of tiles and the order of loops separately for the CPU and MIC; however, even without this fine-tuning the loss of performance on either platform is relatively small. The reader can verify this using the code supplied with this paper.

5.3. Value of Intel Xeon Phi with 1.0x Acceleration

In the past, Colfax Research has published case studies that show versus CPU speedups of up to 3x (e.g., [6, 7, 8, 9]). Here, the application only achieved the same performance on the coprocessor and on the host. This is in part because this is a complicated problem to optimize: with small matrices, intricate details of vectorization and cache operation are crucial for performance. Additionally, not helping the value of the acceleration factor is the fact that the CPU used for baseline is the fastest model in the Ivy Bridge architecture generation. However, the bottom line is 1x acceleration factor through the use of an Intel Xeon Phi coprocessor. Is it good enough to justify using a coprocessor?

"Good enough" has different meanings in different usage scenarios. Sometimes it is important to maximize performance per watt (e.g., in computing centers for which the operating costs are the major concern). Other times, performance to set-up cost ratio is important (when purchasing budget is the limitation). Finally, the sought-for metric may be the best performance per system (for users of a single workstation and in cases where rack space in computing facility is limited). Where the 1x acceleration factor stands in terms of these metrics is addressed in a companion publication [10].

References

[1] Landing page for this paper, "Fine-Tuning Vectorization and Memory Traffic...". http://research.colfaxinternational.com/post/2015/01/27/LU.aspx.

[2] Andrey Vladimirov. Arithmetics on Intels Sandy Bridge and Westmere CPUs: not all FLOPs are created equal. http://research.colfaxinternational.com/post/2012/04/30/FLOPS.aspx.

[3] User and Reference Guide for the Intel C++ Compiler 15.0: restrict, Qrestrict. https://software.intel.com/en-us/node/523123.

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

[5] 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.

[6] Andrey Vladimirov and Cliff Addison. Cluster-Level Tuning of a Shallow Water Equation Solver on the Intel MIC Architecture. http://research.colfaxinternational.com/post/2014/05/12/Shallow-Water.aspx.

[7] Crash Course on Programming and Optimization with Intel Xeon Phi Coprocessors at SC14. http://research.colfaxinternational.com/post/2014/11/16/SC14.aspx.

[8] Andrey Vladimirov. Multithreaded Transposition of Square Matrices with Common Code for Intel Xeon Processors and Intel Xeon Phi Coprocessors. http://research.colfaxinternational.com/post/2013/08/12/Trans-7110.aspx.

[9] Accelerated Simulations of Cosmic Dust Heating Using the Intel Many Integrated Core Architecture. http://research.colfaxinternational.com/post/2013/06/07/HEATCODE.aspx.

[10] Andrey Vladimirov. Performance to Power and Performance to Cost Ratios with Intel Xeon Phi Coprocessors (and why 1x Acceleration May Be Enough). http://research.colfaxinternational.com/post/2015/01/27/1x.aspx.