Task-Based Cholesky Decomposition on Knights Corner Using OpenMP

Joseph Dorris(✉), Jakub Kurzak, Piotr Luszczek, Asim YarKhan, and Jack Dongarra

Innovative Computing Laboratory, Knoxville, TN 37996, USA
jdorris7@vols.utk.edu, {kurzak,luszczek,yarkhan,dongarra}@icl.utk.edu

Abstract. The growing popularity of the Intel Xeon Phi coprocessors and the continued development of this new many-core architecture have created the need for an open-source, scalable, and cross-platform task-based dense linear algebra package that can efficiently use this type of hardware. In this paper, we examined the design modifications necessary when porting PLASMA, a task-based dense linear algebra library, to run effectively on Intel's Knights Corner Xeon Phi coprocessor. First, we modified PLASMA's tiled Cholesky decomposition to use OpenMP for its scheduling mechanism to enable Xeon Phi compatibility. We then compared the performance of our modified code to that of the original dynamic scheduler running on an Intel Xeon Sandy Bridge CPU. Finally, we looked at the performance of the new OpenMP tiled Cholesky decomposition on a Knights Corner coprocessor. We found that desirable performance for this architecture was attainable with the right code optimizations; these changes were necessary to account for differences in the runtimes and in the hardware itself.

Keywords: Cholesky decomposition · Linear algebra · OpenMP · PLASMA · Task-based programming · Tile algorithms · Xeon Phi

1 Introduction

Linear systems of equations and eigenvalue problems are integral parts of many different scientific and engineering applications. These codes can be very computationally intensive and much effort has been dedicated to increasing the speed and efficiency of these codes. New accelerator and coprocessor architectures such as GPUs and the Intel Xeon Phi offer the potential for increased performance, but due to major architecture design differences (traditional CPU vs. accelerator), accelerators/coprocessors also have substantial overhead in terms of optimizing old code to achieve the potential performance benefit of the new architecture.

Developers of linear algebra libraries who want to utilize the Xeon Phi have previously used techniques that offloaded the specific Basic Linear Algebra Subprogram (BLAS) routines to the Xeon Phi, or used a hybrid approach that offloaded some of the work, such as in Matrix Algebra on GPU and Multicore

M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS 9945, pp. 544–562, 2016.
DOI: 10.1007/978-3-319-46079-6_37

Architectures (MAGMA) [8]. This approach is designed based on the assumption that the controlling thread needs to be run on a separate primary processor such as is required for a GPU. However, the Xeon Phi architecture differs from a GPU in that it allows more complex threads than GPUs, which makes it more similar to a traditional CPU. This architecture, which has been referred to as "many-core," seems to be reverting back to a traditional multi-core lineage, and Intel announced that the next generation architecture (Intel Knights Landing) will work as a primary processor [17].

There are fewer options available for dense linear algebra when using the currently available Xeon Phi (Knights Corner) as a primary processor. Intel's Math Kernel Library (MKL) provides dense linear algebra routines on Intel architectures including the Xeon Phi. However, it is not open-source. Open-source software would allow developers to better understand the execution behavior of these routines and have the ability to customize parameters to tailor to their specific application. Also, MKL is not available for other non-Intel architectures that require effective and scalable dense linear algebra libraries.

One method for performing dense linear algebra on multi-core architectures is to use a task-based model for computations. This is the approach taken by PLASMA (Parallel Linear Algebra Software for Multicore Architectures), which has shown good performance on many different machines [1,4], but has yet to target the Xeon Phi due to differences in architectures. However, the implementation of task-dependencies in OpenMP 4.0 provides an opportunity to port this library to the Xeon Phi as well as decrease the size of the code base.

1.1 Contributions

The contributions of this paper are:

- We implemented a task-based tile Cholesky decomposition using OpenMP 4.0 directives based on the PLASMA linear algebra library.
- We compared the performance of using OpenMP's tasking dependencies with the previous dynamic scheduling mechanism.
- We measured the performance of this task-based tile Cholesky algorithm on Knights Corner.
- We investigated the execution behavior of this algorithm and discovered various ways of improving performance.

These contributions show the viability of task-based algorithms on the Xeon Phi architecture.

2 Background

2.1 Intel Xeon Phi Coprocessor

Intel developed the Xeon Phi coprocessor in response to the growing demand for accelerators to provide high performance and efficiency. The Xeon Phi's high

performance is obtained by using a large number of cores, wide vector units, and multiple threads per core [10]. While code can be compiled for this architecture without major changes, reaching peak performance still depends on careful distribution of work across the threads and cores, as well as consideration of the vector units. Also, the Xeon Phi relies heavily on effective usage of the caches, and it can be difficult to use the caches in a way that does not incur cache consistency penalties with this many cores.

The most recent model of Xeon Phi is called Knights Corner (KNC) and has 61 cores operating at 1.238 GHz. Knights Corner has a 512-bit instruction set and 8 double precision wide vector processing units. It also supports fused multiply add, so it is capable of 16 double precision floating point operations per cycle [5]. This gives Knights Corner a theoretical double precision peak performance of 1,208.29 GFLOPS.

Knights Corner acts as a *coprocessor*. It is connected to a primary processor through a PCI Express bus and has its own embedded Linux operating system that handles all of the scheduling functionality and process and thread management. The operating system stack allows for a secure shell interface, through which code can be executed natively; heterogeneous code is also possible using the compiler offload capabilities [10].

Threads. The main difference between the Xeon Phi and other Intel multi-core architectures is its use of up to 4 hardware threads on each core with a short in-order pipeline. These are different from *hyperthreads*, which can be found on a Xeon CPU, in that hyperthreads are hardware threads on an out-of-order execution engine. In a Xeon CPU, the full floating point potential can be reached using a single thread and the out-of-order execution allows it to tolerate latency. Additional threads are only helpful for more latency tolerance, but often put more pressure on the memory. For this reason, typically only 1 thread is used per core for dense linear algebra codes on CPUs.

The Xeon Phi, on the other hand, schedules using a simple round-robin scheme with its 4 threads, and is able to execute 2 vector instructions in parallel, but they must come from different threads [15]. This means that peak performance is likely only possible with at least 2 threads per core. However, providing 4 threads per core provides more latency tolerance and is what is typically recommended. Drawbacks to adding additional threads can occur, however, in that they can negatively affect caching behaviors which could be especially detrimental in codes that are not compute bound.

2.2 PLASMA

The Parallel Linear Algebra Software for Multicore Architectures (PLASMA) project was developed at the Innovative Computing Laboratory (ICL) starting in 2007 to provide high performance dense linear algebra routines for multiple socket and multiple core architectures [4]. PLASMA contains many different linear algebra algorithms and supports single, double, single complex, and double complex precision. PLASMA is able to efficiently use the hardware by leveraging

algorithms that can distribute the workload, and a system of dynamic scheduling in which work is assigned based on data and core availability. Thus, this is a system of asynchronous, out-of-order scheduling of task-structured operations.

Tiling. The benefit of PLASMA comes from its ability to effectively distribute the computation to multiple cores that can operate simultaneously on their contiguous memory blocks. This maximizes the operations performed on the data cached by each core prior to eviction, while also limiting synchronization issues. It accomplishes this feat using tiling algorithms.

Tile algorithms work by first separating the matrix into memory contiguous tiles. Thus, a matrix of size N by N will be divided into tiles of size NB by NB, producing $(N/NB)^2$ tiles of the matrix. Operations are performed between individual tiles and then combined to produce the overall desired computation. The tile operations can then be performed in parallel, when there are no dependencies between them, with minimized synchronization points and without the risk of cache consistency problems.

Deciding tile size is necessary to obtain good performance, since overall performance is affected by tile size and number of tiles, but the process is not always straightforward. Because of factors like memory latency and throughput of a given architecture, tiles can become increasingly memory-bound with smaller tiles, and thus can have reduced performance. However, if the tiles are too large then there may not be enough parallel work to be effectively distributed to all cores.

All of the separate tile operations are self-contained tasks that have memory dependencies associated with the data to be operated on, and some dependency-based order of operations specified by the tiled algorithm. This can be viewed as a graph where nodes represent tasks and edges represent dependencies between them. This forms a directed acyclic graph (DAG), and a DAG representation can help discover work that can be run in parallel because there are no remaining dependencies. Ideally, a scheduler would be able to identify some of this parallel work and distribute the work in a way that would allow for the fastest computation. The ability to transform linear algebra algorithms into a task-based model provides a representation that helps simplify parallelization.

Scheduling. PLASMA has two types of scheduling available: static and dynamic. The static scheduling mechanism will assign tasks to cores before execution and the tasks will wait to begin execution until all of their dependencies are met. Task dependencies and completions are then tracked by a global progress table. Performance then depends on using the static pipeline [12]. However, this method lacks the ability to schedule all tasks whose dependencies have been met as quickly as possibly because the scheduling is performed beforehand and will not be able to account for variations in task execution times. Artificial synchronization points expose serial sections of code, and this can leave some cores idle. Static scheduling also cannot distribute the tasks as well across a large number of cores, and lacks generality in that the pipeline must be considered when designing the algorithm.

PLASMA is designed to achieve the best performance when using a dynamic scheduling mechanism. This is different from static scheduling in that as cores finish tasks they can be assigned any tasks whose dependencies have been met at runtime. This is considered "data-driven" scheduling. This allows better work distribution and less idle time on all cores.

The dynamic scheduling was previously controlled by an internal runtime called QUARK (Queueing And Runtime for Kernels). This scheduler was shown to perform very well for previous PLASMA work distribution on other architectures. However, when we compiled PLASMA using QUARK on the Xeon Phi, initial tests showed that QUARK did not produce sufficient performance because multiple Xeon Phi threads were necessary per core. While only slight modifications to the code could have fixed this problem, another solution presented itself when the PLASMA project decided to transition to a new dynamic scheduling mechanism... OpenMP.

2.3 OpenMP

OpenMP [6] was created in October 1997 to provide an easy method for exploiting shared memory parallelism. OpenMP is an API that uses a collection of compiler directives, library routines, and environment variables to control underlying implementation. It is now an option provided by most compilers including Intel, which allows it be a viable option for parallel programming on the Xeon Phi. It was designed in a way that focused on ease of use but still allows a wide variety of features. It has continued to add to this list of features over the years, one of the most recent being tasking.

In 2009, the release of OpenMP 3.0 added support for the tasking model of parallelism which added the ability for parallelization of irregular problems, which have recursive, unbounded loops. In 2013, the release of OpenMP 4.0 added new capabilities to allow tasks to specify data dependencies. This provides support for a task-based model for programs in which each task can depend on data which may be manipulated by earlier tasks. The program can then be represented as a DAG of tasks, and these tasks are made to execute on available hardware as their dependencies are met.

GNU and Intel currently support OpenMP 4.0. This new support for tasks with dependencies provides the necessary abstraction to allow PLASMA to easily replace its internal dynamic scheduler with OpenMP task directives, and thus be able to run on a Xeon Phi coprocessor.

2.4 Cholesky Decomposition

Algorithm. Cholesky decomposition is the decomposition of a symmetric positive definite matrix A into a lower triangular matrix and its conjugate transpose (Eq. 2). Cholesky decomposition is used for solving linear systems of equations, which is common in many science and engineering applications. The formula for calculating each matrix entry can be seen in Eqs. 4 and 5. As the matrix grows in size, this algorithm for solving the matrix will depend on accessing increasingly

distant memory locations which can make the work difficult to parallelize and lead to memory thrashing.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{1}$$

$$A = LL^T = \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{pmatrix} \tag{2}$$

$$= \begin{pmatrix} l_{11}^2 & l_{21}l_{11} & l_{31}l_{11} \\ l_{21}l_{11} & l_{21}^2 + l_{22}^2 & l_{31}l_{21} + l_{32}l_{22} \\ l_{31}l_{11} & l_{31}l_{21} + l_{32}l_{22} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{pmatrix} \tag{3}$$

$$l_{i,i} = \sqrt[2]{a_{i,i} - \sum_{k=1}^{i-1} l_{i,k}^2} \tag{4}$$

$$l_{i,j} = \frac{1}{l_{j,j}} \left(a_{i,j} - \sum_{k=1}^{j-1} l_{i,k} l_{j,k} \right), \quad i > j \tag{5}$$

Tiled Cholesky Decomposition. PLASMA uses a tiled version of Cholesky decomposition. The premise of this method is to separate the operations that are taking place in the above algorithm to allow effective parallelization. The creation of this algorithm can be seen in the LAPACK User's Guide [2]. It is composed of the BLAS tile operations: matrix-matrix multiplications (GEMM), solving the triangular matrix equation (TRSM), symmetric rank-k update (SYRK), and Cholesky decomposition (POTRF). All of these are Level 3 BLAS, which means that they are no longer memory bound and the peak theoretical performance will increase as the tile size increases.

There are three common variations for scheduling the tile operations necessary to complete the whole computation. They all have the same tasks and dependencies, as they are performing the same computation. However, the order in which these operations are scheduled can vary, and these variations can drastically affect the view of the tasks presented to the scheduler, and thus the order of completion of tasks.

Scheduling Variations. The three variations of tiled Cholesky decomposition are: right-looking (Fig. 1), left-looking (Fig. 2), and top-looking (Fig. 3). The availability of work as seen by the scheduler can be seen in the task dependency DAGs in Fig. 4.

The right-looking version can be considered the most aggressive and offers the most parallelization with its breadth first task exploration. This is why right-looking was previously selected for PLASMA dynamic scheduling. The top-looking version can then be described as the "lazy" version because it is using depth first exploration of the task graph, which limits the number of tasks

```
begin
    for k = 0 to nt − 1 do
        POTRF(A(k,k)));
        for m = k + 1 to nt − 1 do
            TRSM(A(k,k),A(m,k));
        end
        for m = k + 1 to nt − 1 do
            SYRK(A(m,k), A(m,m));
            for n = k + 1 to m − 1 do
                GEMM(A(m,k),A(n,k),A(m,n));
            end
        end
    end
end
```

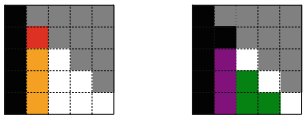


Fig. 1. Right-looking variation of the tiled Cholesky decomposition (green = GEMM, red = POTRF, orange = TRSM, and purple = SYRK) (Color figure online)

that are immediately able to be run. The PLASMA static scheduler uses left-looking Cholesky decomposition because it was determined to be the best for the static pipeline [12].

3 Related Work

This paper is building off of previous work that took place to create PLASMA at the Innovative Computing Laboratory [4] in order to broaden the scope of the library to include Xeon Phi coprocessors. Virouleau et al. [18] evaluated replacing QUARK calls with OpenMP tasks with dependencies on Intel and AMD multi-core machines, but they did not measure performance on the Intel Xeon Phi and also did not compare to the performance when using the Intel OpenMP runtime.

LibFLAME [14] is a dense linear algebra library developed at the University of Texas at Austin. Dolz et al. [7] tested running libFLAME on a Xeon Phi while attempting to balance task and data parallelism to maximize performance and energy efficiency. However, they did not consider algorithm specific effects or the possibility of different BLAS routines being optimal with varying numbers of threads (to be discussed later). OmpSs [9] and XKaapi [13] provide OpenMP tasking-like alternatives for task implementations of Cholesky decomposition on a Xeon Phi.

Knights Corner has been available since 2012, allowing ample time for analysis. Schmidl et al. [16] studied the performance of OpenMP programs as compared to an Intel Xeon Sandy Bridge in terms of memory bandwidth and overhead of OpenMP constructs when utilizing the dynamic scheduler. However, the authors were not looking at tasks with dependencies and the degraded performance with a large number of tasks (likely because it was written before tasks with dependencies were implemented). Fang et al. [11] studied the Xeon Phi Architecture and performance, but was not focused on optimizations to increase performance.

```
begin
    for k = 0 to nt − 1 do
        for n = 0 to k − 1 do
            SYRK(A(k,n), A(k,k));
            for m = k + 1 to nt − 1 do
                GEMM(A(m,n),A(k,n),A(m,k));
            end
        end
        POTRF(A(k,k)));
        for m = k + 1 to nt − 1 do
            TRSM(A(k,k),A(m,k));
        end
    end
end
```

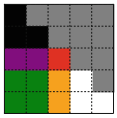


Fig. 2. Left-looking variation of the tiled Cholesky decomposition (green = GEMM, red = POTRF, orange = TRSM, and purple = SYRK) (Color figure online)

```
begin
    for k = 0 to nt − 1 do
        for n = 0 to k − 1 do
            for m = 0 to n − 1 do
                GEMM(A(k,m),A(n,m),A(k,n));
            end
            TRSM(A(n,n),A(k,n));
        end
        for n = 0 to k − 1 do
            SYRK(A(k,n), A(k,k));
        end
        POTRF(A(k,k)));
    end
end
```

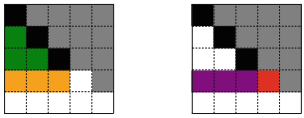


Fig. 3. Top-looking variation of the tiled Cholesky decomposition (green = GEMM, red = POTRF, orange = TRSM, and purple = SYRK) (Color figure online)

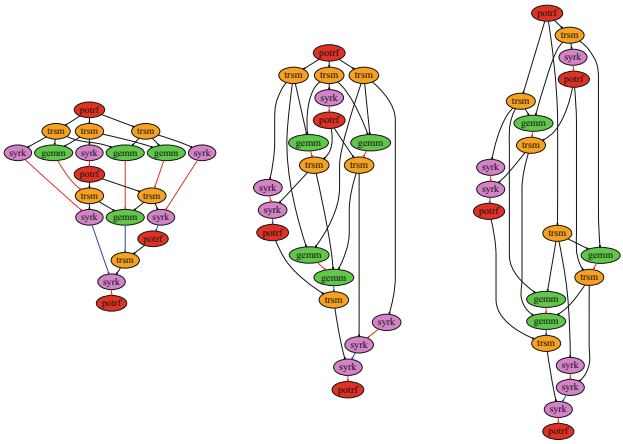


Fig. 4. DAGs for 3 variations of tiled Cholesky decomposition (from left to right): right-looking, left-looking, and top-looking. These show how the order in which tasks are presented to the scheduler affect the available parallelization (green = GEMM, red = POTRF, orange = TRSM, and purple = SYRK). (Color figure online)

4 OpenMP Task-Based Cholesky Decomposition

To transition the PLASMA tiled Cholesky decomposition to run on the Xeon Phi, we wrote the three different tiled versions in C, replacing the previous QUARK calls with OpenMP 4.0 tasking directives (right-looking in Fig. 5). This implementation starts a pool of threads with "**#pragma** omp parallel," and then uses a master thread to sequentially create all of the tasks and specify their dependencies. After the tasks are created, the scheduler can assign them to available threads/cores for execution.

OpenMP allows for specifying whether the task only needs to read data (in:), write data (out:), or both (inout:). The scheduler will then be able to use this information to safely start tasks when data dependencies are met. Specifying the dependencies is straightforward with the tile layout because each tile is contiguous in memory so they can be specified by the start of the tile and the size of the tiles.

This method can be applied to all the linear algebra routines that are included in PLASMA. By removing the internal scheduler, it would make the software more minimalist and standardized, as well as allow PLASMA to gain all of the customization and support of OpenMP.

Performance of Task-Based Runtimes on Xeon Sandy Bridge. We tested the OpenMP double precision right-looking tiled Cholesky decomposition performance on an Intel Xeon Sandy Bridge with QUARK, GCC OpenMP, and Intel OpenMP. This processor has 16 cores, a clock frequency of 2.6 GHz, and 8 double precision FLOPS/clock to give a theoretical peak of 332.8 GFLOPS. We set the outer blocking size to be 128 and varied N from 128 to 14080 to see how the scheduling mechanisms behaved as the number of tasks increased. The results can be seen in Fig. 6.

```
#pragma omp parallel
#pragma omp master
PLASMA POTRF( tiled_matrix A, tilesize ts) {
  for (k = 0; k < M; k++) {
    #pragma omp task depend(inout:A(k,k)[0:ts])
    { POTRF( A(k,k) ); }
    for (m = k+1; m < M; m++)
      #pragma omp task depend(in:A(k,k)[0:ts]) depend(inout:A(m,k)[0:ts])
      { TRSM( A(k,k), A(m,k) ); }
    for (m = k+1; m < M; m++) {
      #pragma omp task depend(in:A(m,k)[0:ts]) depend(inout:A(m,m)[0:ts])
      { SYRK( A(m,k), A(m,m) ); }
      for (n = k+1; n < m; n++)
        #pragma omp task depend(in:A(m,k)[0:ts], A(n,k)[0:ts]) \
          depend(inout:A(m,n)[0:ts])
        { GEMM( A(m,k), A(n,k), A(m,n) ); }
    }
  }
}
```

Fig. 5. Right-looking tiled Cholesky decomposition with OpenMP tasks. This code segment shows how PLASMA-style tile algorithms can be expressed using OpenMP pragmas.

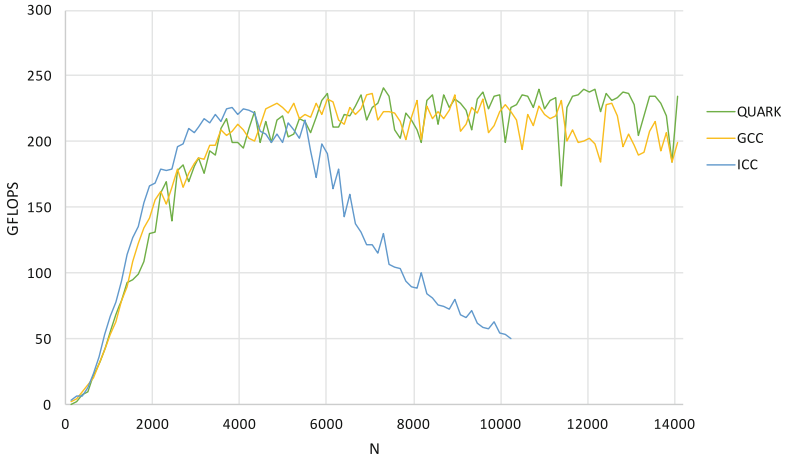


Fig. 6. Performance of double precision, right-looking, tiled Cholesky decomposition with different scheduler implementations on Xeon Sandy Bridge (QUARK runtime, GCC OpenMP, Intel OpenMP)

Based on the results, we can see that the GCC OpenMP implementation behaves similarly to the internally developed task-based runtime QUARK, which shows that OpenMP has the potential to be a complete replacement for our dynamic scheduler. However, the Intel OpenMP implementation has severely decreased performance when the matrix size N exceeds 4000, likely due to the large number of tasks. The Intel implementation of the OpenMP runtime is the only option available to the Xeon Phi, so this must be considered when optimizing for performance.

5 Task-Based Cholesky Decomposition on a Xeon Phi

5.1 Experimental Setup

Hardware. We ran all tests on a 61 core MIC 7120 (Knights Corner). We compiled our proof of concept code for this architecture using the Intel compiler and the "-mmic" flag. We launched every run using "micnativeloadex" which required 1 core for operating system functions and communication, leaving the other 60 cores available for the Cholesky decomposition. This left a theoretical maximum of 1,188.48 double precision GFLOPS, assuming each core was able to make full use of its vector instructions, use fused multiply add, and properly use multiple threads to perform 16 double precision FLOPS per cycle.

MKL Performance. To give a baseline for the possible performance of Cholesky decomposition on Knights Corner, we ran the MKL version 11.3.1 double precision Cholesky decomposition (DPOTRF) on matrices of varying sizes. The points tested for MKL performance were multiples of 200 and multiples of 256

up to 16000. The goal of this project was not to outperform MKL but rather to show the application of task-based algorithms on the Xeon Phi architecture. However, if the task-based method for Cholesky decomposition can be shown to have reasonable performance, it provides evidence that tile-based linear algebra algorithms from PLASMA can provide benefits over MKL for some of its other routines, including tall-and-skinny QR, SVD, and EVP as it has done on other architectures.

Tile Size. Measuring performance for a tile-based algorithm required considering various tile sizes. This was necessary because the optimal tile size varies depending on the size of the matrix on which the user intends to operate. A certain number of tiles will be necessary to successfully distribute the computation across the large number of cores on Knights Corner. However, smaller tiles will have lower performance due to being more memory bound and thus will limit the theoretical peak of the whole computation. This creates a need to find a tile size that balances these two considerations optimally for the overall matrix size. PLASMA intends to have desirable performance for all ranges of matrix sizes, so extensive testing on a wide range of tile sizes was required.

BLAS Library. We used the Intel multi-threaded MKL math library for the individual tile kernels (GEMM, POTRF, TRSM, and SYRK), which is optimized for Xeon Phi cores. This means that the computation used nested levels of threading. The top level distributed the work across the many cores and the second level provided multiple hardware threads for MKL. MKL could easily be replaced with other libraries as they become available or if they are necessary for another architecture.

Warmup. The first time we run an MKL routine, we incur some overhead from loading the libraries. When used in practice, it is likely that many calls will be made to these linear algebra routines, so this overhead can be ignored when timing for performance measurement. To account for this extra overhead, the PLASMA library timing examples provide a command line option to do a dry run of the algorithm once before running a computation for timing. This option must be used for all timings, and a warmup method was also used before the MKL performance measurement in Fig. 16.

Traces. To help understand the flow of execution and the scheduling of work on cores, traces were used (Figs. 7, 8, 13, and 14). These are figures that show the compute cores on the y axis and time along the x axis. This is a helpful tool for viewing how the computation progresses and how tasks are scheduled on the cores. These figures can also provide insight into factors that affect performance. Creating this visualization involved keeping track of which core the kernels ran on and the start and completion times for each. We recompiled the code separately with these function calls when tracing, and these runs were not used for measuring performance. The colored blocks on the trace represent the kernel that is running: green = GEMM, red = POTRF, orange = TRSM, and purple = SYRK.

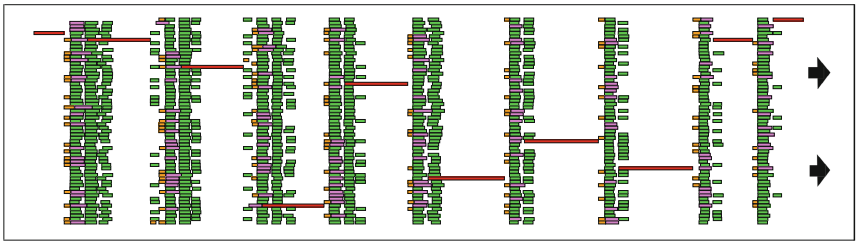


Fig. 7. PLASMA OpenMP Cholesky decomposition trace: all kernels use 4 threads (incomplete) (Color figure online)

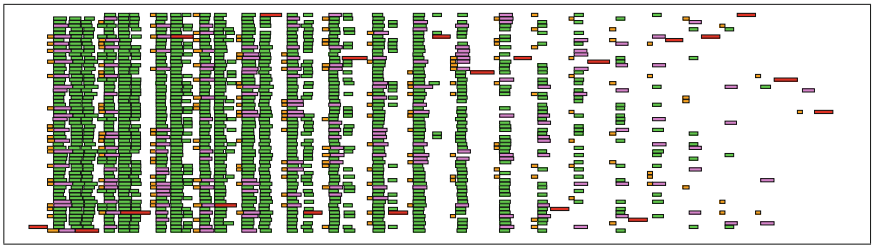


Fig. 8. PLASMA OpenMP Cholesky decomposition trace: DGEMM, DSYRK, and DTRSM use 4 threads, and DPOTRF uses 1 thread (Color figure online)

5.2 Execution Environment

Running a program using OpenMP on a Xeon Phi can be controlled by a large number of environment variables. These variables communicate to the Xeon Phi operating system and OpenMP about what hardware to use, how to schedule work on that hardware, the available threads, and many other customizations. We discovered, through investigation and testing, that the desired behavior of tiled OpenMP Cholesky required setting the following variables:

- KMP_PLACE_THREADS = 60t, 4c - use 60 cores and 4 hardware threads on each
- KMP_HOT_TEAMS_MODE = 1 - allows OpenMP threads to stay alive
- KMP_HOT_TEAMS_MAX_LEVEL = 2 - keeps nested level OpenMP threads alive
- OMP_NESTED = TRUE - allows multiple levels of parallelism
- OMP_NUM_THREADS = 60, 4 - a hierarchy of 60 threads and 4 subthreads
- OMP_PROC_BIND = spread, close - specifies how threads are bound to resources
- MKL_DYNAMIC = FALSE - disable MKL dynamic adjustment of threads
- MKL_DOMAIN_NUM_THREADS = MKL_DOMAIN_BLAS = 4 - suggests number of threads for a particular function domain

After we set these environment variables, we created an initial trace for the right-looking Cholesky decomposition on the Xeon Phi to discover what factors affected performance, and to gain insight into how the performance of the

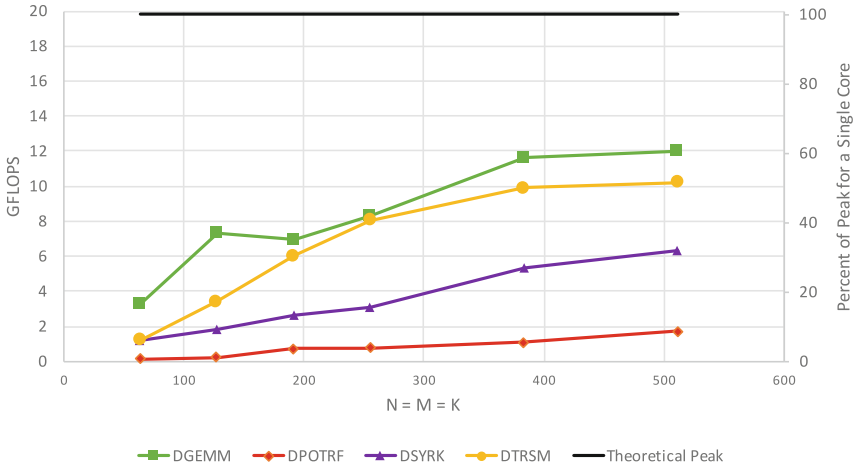
Cholesky decomposition can be improved. A trace for a matrix of size N = 5120 and with a tile size NB = 256 is shown in Fig. 7.

5.3 Individual Kernel Performance

When examining the initial trace, the DPOTRF kernel, which consists of the fewest FLOPS of all of the kernels [3], is taking considerably longer to execute than all the other kernels. One can also see that in the task dependency DAG representation the DPOTRF kernels are a common path and a bottleneck for execution. These two observations make this kernel a prime target for optimization.

As a test, we decided to vary the number of threads used per core by the individual kernels to determine which number of threads would be best for the performance of each kernel. Tiles sizes of 64, 128, 192, 256, 384, and 512 were tested, and the performance was calculated based on the median runtime for each kernel and configuration.

Figure 9 shows that on average GEMM, TRSM, and SYRK performed best with 4 threads, but POTRF performed best with 1 thread. The MKL library allows runtime switching the number of threads used for a kernel, so it can be



DGEMM Performance (GFLOPS)						
threads	NB=64	NB=128	NB=192	NB=256	NB=384	NB=512
1	2.963	3.486	3.104	3.404	6.384	6.570
2	2.805	3.998	3.807	4.635	6.583	7.674
3	2.675	3.478	4.530	4.244	6.705	7.869
4	3.297	7.321	6.925	8.331	11.592	12.036

DSYRK Performance (GFLOPS)						
threads	NB=64	NB=128	NB=192	NB=256	NB=384	NB=512
1	1.133	1.875	1.970	1.971	2.295	3.003
2	1.057	1.794	1.726	1.711	2.145	2.834
3	0.993	1.659	1.470	1.527	1.947	2.682
4	1.199	1.830	2.665	3.121	5.295	6.367

DPOTRF Performance (GFLOPS)						
threads	NB=64	NB=128	NB=192	NB=256	NB=384	NB=512
1	0.137	0.220	0.748	0.792	1.088	1.724
2	0.100	0.062	0.426	0.544	0.944	1.627
3	0.091	0.036	0.339	0.480	0.826	1.402
4	0.081	0.134	0.323	0.333	0.802	0.884

DTRSM Performance (GFLOPS)						
threads	NB=64	NB=128	NB=192	NB=256	NB=384	NB=512
1	1.304	3.262	3.960	4.176	5.395	5.323
2	1.331	3.623	5.420	6.828	9.092	9.272
3	1.267	3.350	6.096	7.457	8.164	8.478
4	1.214	3.427	6.050	8.061	9.894	10.199

Fig. 9. MKL v11.3.1 kernel performance on a single Knights Corner core

switched to 1 thread whenever the core is going to perform a POTRF, and then set back to 4 when the POTRF is completed to allow maximum performance for the other kernels. This decreased runtime and its effect on the trace can be seen in Fig. 8. The DPOTRF kernels complete much more quickly and therefore do not stall the execution of the other tasks to the same extent, which leads to increased overall performance.

This was an unexpected result, as it is commonly suggested to use 2–4 threads for peak performance. It was also very poor performance even with the best configuration which becomes obvious when one realizes that this scenario results in performance of less than 10 % of peak for a single core, with a tile of size 512 by 512. This result seems to suggest that an improved implementation of this kernel might be possible for small tile sizes, which would drastically improve performance of this algorithm, but is beyond the scope of this paper.

5.4 Scheduling Variations

The next test was to see which variations of tiled Cholesky decomposition (right-looking, left-looking, and top-looking) would perform the best on Knights Corner. Tiles of size 128, 256, and 512 were tested to observe the behavior of the different algorithms at different granularities.

The results are shown in Figs. 10, 11, and 12. For all tile sizes, the top-looking Cholesky implementation performed the best or equal to the other variations. While the right-looking implementation seemed to offer the most parallelism, and hence the hypothesized best performance on Knights Corner, this was not the case. Also, it can be seen that even when switching to the top-looking algorithm, using a tile size of 128 does not benefit from using a dynamic scheduler because of the immense load on the scheduler to manage the increased number of very small tasks.

The fact that the top-looking implementation, which was supposed to be the least aggressive, performed the best raised some questions as to why this was occurring. We obtained traces of a right-looking and a top-looking Cholesky decomposition at N = 5120, NB = 128 when the performance of each had diverged

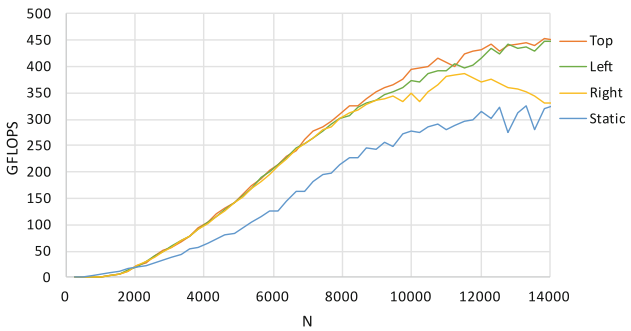


Fig. 10. Tiled Cholesky decomposition variations, NB = 256

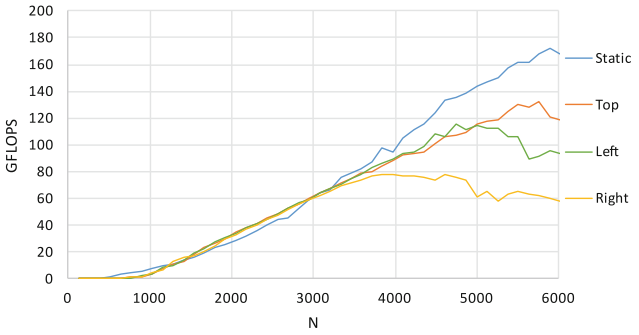


Fig. 11. Tiled Cholesky decomposition variations, NB = 128

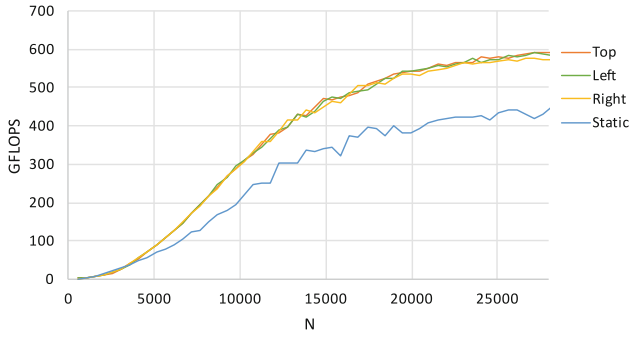


Fig. 12. Tiled Cholesky decomposition variations, NB = 512

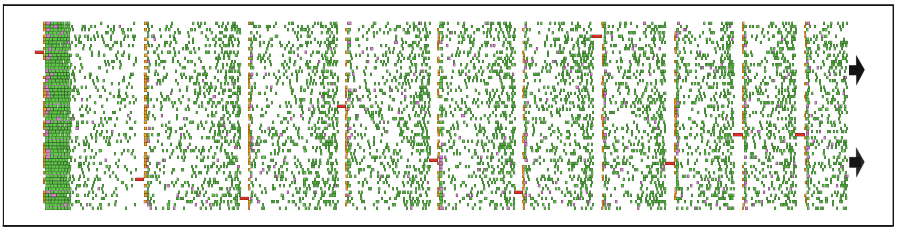


Fig. 13. OpenMP right-looking Cholesky decomposition-N = 5120, NB = 128 (incomplete)

(Figs. 13 and 14, respectively). There is considerable idle time on the right-looking implementation when there is a large number of GEMMs that need to be completed. Their dependencies have been met according to DAG representation for right-looking Cholesky, yet there is delay in scheduling them.

The Intel runtime is proprietary software, so we were unable to investigate further. We believe that the Intel implementation of the OpenMP runtime does not handle a large number of tasks well because of its method of maintaining the

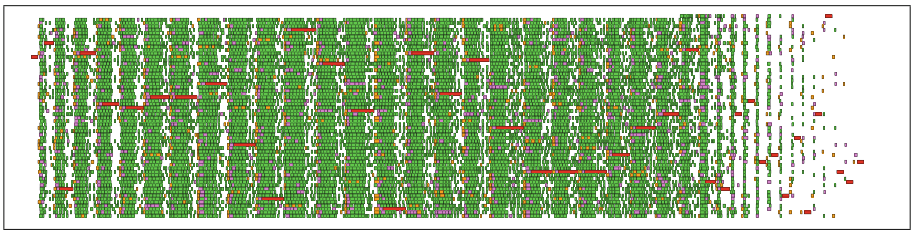


Fig. 14. OpenMP top-looking Cholesky decomposition-N = 5120, NB = 128

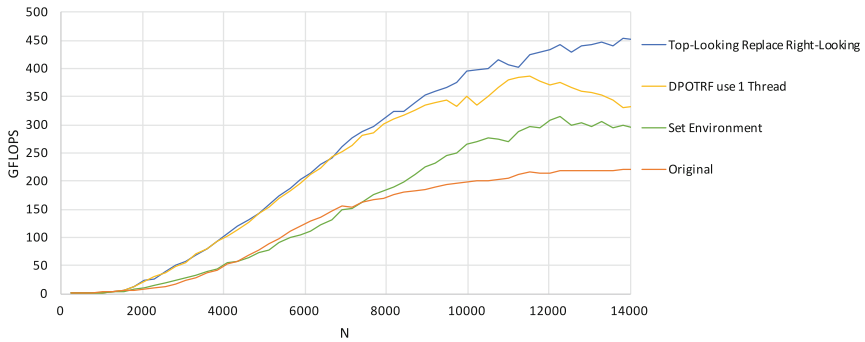


Fig. 15. OpenMP Cholesky decomposition incremental performance improvement (NB = 256)

tasks and the overhead associated with them. The top-looking version unrolls the DAG slower, so the runtime has less work when updating dependencies after the completion of tasks, and is better able to handle it.

5.5 Comparison and Final Performance

The combination of correctly setting environment variables, modifying the number of threads for DPOTRF, and using top-looking Cholesky decomposition—as opposed to the original right-looking Cholesky—offered the best performance. The incremental benefits of each modification, while using tiles of size 256, are shown in Fig. 15.

After these optimizations, the curves for different tile sizes can be compared to a standard LAPACK-style implementation in MKL. Tile sizes of 192, 256, and 384 are shown in Fig. 16 as they were found to have the best performance curves after sweeping through various tile sizes with all of the combined optimizations. In fact, as matrix size increases, the optimal tile size will also increase. This is caused by balancing individual kernel performance and work distribution. However, if set correctly by the user, tiled Cholesky decomposition can obtain performance comparable to MKL and can reach around 50 % of peak.

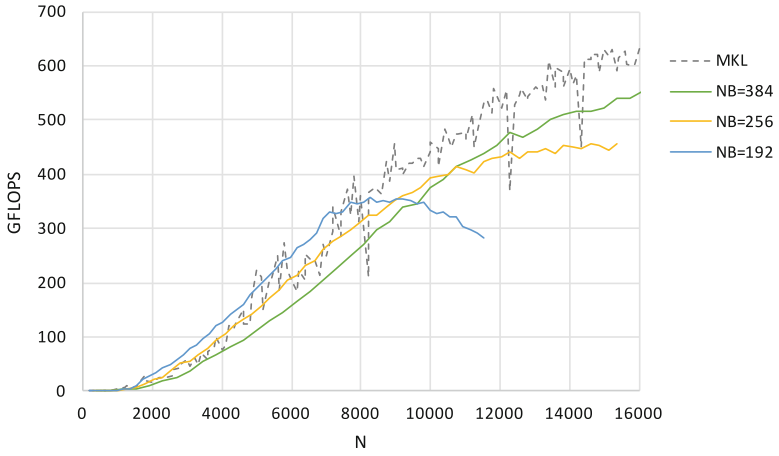


Fig. 16. Final OpenMP task-based double precision Cholesky decomposition performance

6 Conclusion

The architectural differences between the Xeon Phi and previous multi-core processors provided many challenges that had to be addressed to achieve good performance. This performance was only possible with multiple threads per core, which created hierarchical levels of parallelism that were not previously considered with PLASMA. Additionally, the optimal number of threads in this parallelism was not consistent between different kernels. This created issues like having to dynamically set the number of threads for MKL calls depending on the kernel.

The Intel OpenMP runtime had difficulty handling a large number of tasks, which added to the challenge. The improved performance of GCC OpenMP runtime implementation on a traditional CPU gives credence to the idea that this method of runtime could also provide improved performance on the Xeon Phi. However, until the Intel implementation is improved, we demonstrated that a method for mitigating these runtime issues is to use algorithms that limit the parallelism presented to the scheduler.

The PLASMA OpenMP framework can produce good performance for Cholesky decomposition on a Knights Corner after making only minor modifications. This proved that a port of PLASMA to the Xeon Phi will be straightforward and has potential for high performance.

7 Future Work

Many of the parameter configurations for optimal performance, such as using one thread for POTRF and choosing top-looking tiled Cholesky decomposition as

opposed to right-looking, were based on underlying kernel and scheduler implementation issues that we believe may be changed in the future. The process outlined in this paper will need to be repeated for the Knights Landing processor to see if these decisions are still applicable. Also, PLASMA contains many other routines. Extensive testing is required for the other remaining algorithms to determine if any other kernels perform better with one thread, or if there are other factors that affect the performance. There is more work to be done before a Xeon Phi PLASMA release, but the applicability of this task-based approach to an order of magnitude more cores and the next generation of architectures is becoming evident.

Acknowledgements. This work has been funded by the National Science Foundation through the Sustained Innovation for Linear Algebra Software project (grant #1339822) and the Empirical Autotuning of Parallel Computation for Scalable Hybrid Systems project (grant #1527706).

References

1. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. J. Phys. Conf. Ser. **180**, 012037 (2009). IOP Publishing
2. Anderson, E., Bai, Z., Bischof, C., Blackford, S.L., Demmel, J.W., Dongarra, J.J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.C.: LAPACK User's Guide, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia (1999)
3. Blackford, S., Dongarra, J.J.: Installation guide for LAPACK. Technical report 41, LAPACK Working Note, June 1999 (originally released March 1992)
4. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. **35**(1), 38–53 (2009)
5. Chrysos, G.: Intel® Xeon Phi coprocessor-the architecture. Intel Whitepaper (2014)
6. Dagnum, L., Menon, R.: OpenMP: an industry-standard API for shared memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
7. Dolz, M.F., Igual, F.D., Ludwig, T., Piñuel, L., Quintana-Ortí, E.S.: Balancing task-and data-level parallelism to improve performance and energy consumption of matrix computations on the Intel Xeon Phi. Comput. Electr. Eng. **46**, 95–111 (2015)
8. Dongarra, J., Gates, M., Haidar, A., Jia, Y., Kabir, K., Luszczek, P., Tomov, S.: Portable HPC programming on Intel many-integrated-core hardware with MAGMA port to Xeon Phi. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2013, Part I. LNCS, vol. 8384, pp. 571–581. Springer, Heidelberg (2014)
9. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(02), 173–193 (2011)

10. Duran, A., Klemm, M.: The Intel many integrated core architecture. In: 2012 International Conference on High Performance Computing and Simulation (HPCS), pp. 365–366. IEEE (2012)
11. Fang, J., Varbanescu, A.L., Sips, H., Zhang, L., Che, Y., Xu, C.: An empirical study of Intel Xeon Phi (2013). arXiv preprint: arXiv:1310.5842
12. Kurzak, J., Ltaief, H., Dongarra, J., Badia, R.: Scheduling linear algebra operations on multicore processors. Concurr. Comput. Pract. Exp. **22**, 15–44 (2010)
13. Lima, J.V., Broquedis, F., Gautier, T., Raffin, B.: Preliminary experiments with XKaapi on Intel Xeon Phi coprocessor. In: 2013 25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 105–112. IEEE (2013)
14. Quintana-Ortí, G., Quintana-Ortí, E.S., Geijn, R.A., Zee, F.G.V., Chan, E.: Programming matrix algorithms-by-blocks for thread-level parallelism. ACM Trans. Math. Softw. (TOMS) **36**(3), 14 (2009)
15. Reinders, J.: In response to a forum post on 'what is the relation between "hardware thread" and "hyperthread"?', May 2014. https://software.intel.com/en-us/forums/intel-many-integrated-core/topic/515522
16. Schmidl, D., Cramer, T., Wienke, S., Terboven, C., Müller, M.S.: Assessing the performance of OpenMP programs on the Intel Xeon Phi. In: Mohr, B., Mey, D., Wolf, F. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 547–558. Springer, Heidelberg (2013)
17. Trader, T.: Intel Debuts 'Knights Landing' Ninja Developer Platform. HPCwire, April 2016
18. Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., Gautier, T.: Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In: DeRose, L., Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 16–29. Springer, Heidelberg (2014)