HPC Programming on Intel Many-Integrated-Core Hardware with MAGMA Port to Xeon Phi

Jack Dongarra[1,2,3], Mark Gates[1], Azzam Haidar[1], Yulu Jia[1], Khairul Kabir[1], Piotr Luszczek[1], and Stanimire Tomov[1]

[1]University of Tennessee Knoxville

[2]Oak Ridge National Laboratory

[3]University of Manchester

April 29, 2014

Abstract

This paper presents the design and implementation of several fundamental dense linear algebra (DLA) algorithms for multicore with Intel Xeon Phi Coprocessors. In particular, we consider algorithms for solving linear systems. Further, we give an overview of the MAGMA MIC library, an open source, high performance library that incorporates the developments presented here, and, more broadly, provides the DLA functionality equivalent to that of the popular LAPACK library while targeting heterogeneous architectures that feature a mix of multicore CPUs and coprocessors. The LAPACK-compliance simplifies the use of the MAGMA MIC library in applications, while providing them with portably performant DLA. High performance is obtained through the use of the high-performance BLAS, hardware-specific tuning, and a hybridization methodology whereby we split the algorithm into computational tasks of various granularities. Execution of those tasks is properly scheduled over the heterogeneous hardware by minimizing data movements and mapping algorithmic requirements to the architectural strengths of the various heterogeneous hardware components. Our methodology and programming techniques are incorporated into the MAGMA MIC API, which abstracts the application developer from the specifics of the Xeon Phi architecture and is therefore applicable to algorithms beyond the scope of DLA.

Keywords: numerical linear algebra • Intel Xeon Phi processor • Many Integrated Cores • hardware accelerators and coprocessors • dynamic runtime scheduling using dataflow dependences • communication and computation overlap

1 Introduction and Background

Solving linear systems of equations and eigenvalue problems is fundamental to scientific computing. The popular LAPACK library [3], and in particular its vendor optimized implementations such as Intel's MKL [11] or AMD's ACML [2], have been the software of choice to provide solver routines for dense matrices on shared memory systems. This paper considers a redesign of the LAPACK algorithms and their implementation to add efficient support for heterogeneous systems of multicore processors with Intel Xeon Phi coprocessors. This is not the first time that DLA libraries have needed a redesign to be efficient on new architectures. Notable examples being the transition from LINPACK [9] to LAPACK [3] in the 1980's to make algorithms cache-friendly. Also, ScaLAPACK [7] in the 1990's added support for distributed memory systems. And at present time, the PLASMA and MAGMA libraries [1] target efficiency on, respectively, multicore and heterogeneous architectures.

The Intel Xeon Phi coprocessor is a hardware accelerator that made its debut in the late 2012 as a platform for high-throughput technical computing. It is sometimes known under an alternative name of Many Integrated Cores (MIC). For the purposes of this paper, the common mode of operation for the device is called off-load. However, the stand-alone and reverse off-load modes are also valid possibilities. When in off-load mode, the device receives work from the host processor and reports back as soon as the computational task completes. Any such assignment of work proceeds and completes without the host device being involved. In a typical scenario, the host is an Intel x86 CPU such as Sandy Bridge, Ivy Bridge or even more recent Haswell and Ivy Town. The CPU may monitor the activity of communication and/or computation through an event-based interface and can also pursue its own computational activities between events. This is very similar to the operation of hardware accelerators based on throughput-oriented GPUs and compute-capable FPGAs that are specialized for certain types of workloads beyond what could be achieved on standard multicore CPUs. In fact, Xeon Phi is often considered to be an alternative to the hardware accelerators from AMD and NVDIA despite the fact that there exist many technical differences between the three.

The development of new high-performance numerical libraries is a complex endeavor, which requires meticulous accounting for the extreme levels of parallelism, heterogeneity, and wide variety of accelerators and coprocessors available in the current architectures. Challenges vary from new algorithmic designs to choices of programming models, languages, and frameworks that ease the development, future maintenance, and portability. This paper addresses these issues while presenting our approach and algorithmic designs in the development of the MAGMA MIC [14] library.

To provide a uniform portability across a variety of coprocessors/accelerators, we developed an API that abstract the application developer from the low level specifics of the architecture. In particular, we use low level vendor libraries, like SCIF for Intel Xeon Phi (see Section 3), to define API for memory management and off-loading computations to coprocessors and/or accelerators.

To deal with the extreme level of parallelism and heterogeneity in the current architectures, MAGMA MIC uses a hybridization methodology, described in Section 4, where we split the algorithms of interest into computational tasks of various granularities, and properly schedule those tasks' execution over the heterogeneous hardware. Thus, we use a Directed Acyclic Graph (DAG) approach to parallelism and scheduling that has been developed and successfully used for dense linear algebra libraries such as PLASMA and MAGMA [1], as well as in general task-based approaches to parallelism, such as runtime systems like StarPU [4] and SMPSs [6].

Obtaining high performance depends on a combination of algorithmic and hardware-specific optimizations, discussed in Section 4.3. This is in addition to the use of high-performance low-level libraries, which we address in Section 3. This has implications on the resulting software: in order to maintain the performance portability across hardware, it is necessary to provide in the library a number of algorithmic variations that are tunable, e.g., at installation time. This is the basic premise of autotuning – a prominent example of these kinds of advanced optimization techniques.

A performance study is presented in Section 5. Besides verifying our approach and confirming the appeal of the Intel Xeon Phi coprocessors for high-performance DLA, the results open up a number of future work opportunities discussed in Section 6 that concludes the paper.

2 Compiler Support for Offload

In this paper, we consider the off-load mode as the primary mode of operation for the Xeon Phi coprocessor. The device receives work from the host processor and reports back upon completion of the assignment without the host being involved in between these two events. This is very similar to the operation of network off-load engines, specifically, the TCP Off-load Engines (TOEs) that feature an optimized implementation of the TCP stack that handles the majority of the network traffic to lessen the burden of the main processor, which handles other operating system and user application tasks.

The off-load mode for the Xeon Phi devices has direct support from the compiler in that it is possible to issue requests to the device and ascertain the completion of tasks directly from the user's C/C++ code. The support for this mode of operation is offered by the Intel compiler through Phi-specific pragma directives: offload, offload_attribute, offload_transfer, and offload_wait [10]. This is very closely related to the off-load directives now included in the OpenMP 4 standard. In fact, the two are syntactically and semantically equivalent, barring the difference in the "omp" prefix for the OpenMP syntax. A similar standard for GPUs is called OpenAcc. A summary of various programming methods on Xeon Phi is provided in Table 1.

3 Programming Model: Host-Device with a Server based on LLAPI

For many scientific applications, the offload model offered by the Intel compiler, described in §2, is sufficient. This is not the case for a fully equivalent port of MAGMA to the Xeon Phi because of the very rich functionality that MAGMA inherits from both its CUDA and OpenCL ports. We had to use the LLAPI (Low-Level API) based on Symmetric Communication InterFace (SCIF) that offers, as the name suggests, a very low level interface to the host and device hardware. The use of this API is discouraged for most workloads as it tends to be error-prone and offers very little abstraction on top of the hardware interfaces. What motivated us to use it for the port of our library was: 1) the asynchronous events capability that allows low-latency messaging between the host and the device to notify about completion of kernels on Xeon Phi; 2) the possibility of hidding the cost of data transfer between the host and the device which requires the transfer of submatrices to overlap with the computation. The direct access to the DMA (Direct Memory Access) engine allowed us to maximize the bandwidth of data transfers over the PCI Express bus. The only requirement was that the memory regions for transfer be page-aligned and pinned to guarantee their fixed location in the physical memory. Figure 1a shows the interaction between the host and the server running on the Xeon Phi and responding to requests that are remote invocations of numerical kernels on data that have already been transferred to the device.

4 Hybridaziation Methodology and Optimization strategies

The hybridization methodology used in MAGMA [17] is an extension of the task-based approach for parallelism and developing DLA on homogeneous multicore systems [1]. In particular,

- The computation is split into BLAS-based tasks of various granularities, with their data dependencies, as shown in Figure 1b.
- Small, non-parallelizable tasks with significant control-flow are scheduled on the CPUs.

Programming model/API	Status	Portability	Overhead	Language Support
SCIF	Mature	No	None	No
COI	Mature	Yes	Minimal	Yes
OpenMP 4.0	Early	Yes	Varies	Yes
OpenCL	Experimental	Yes	Minimal	No

Table 1: Programming models for the Intel Xeon Phi coprocessors and their current status and properties.

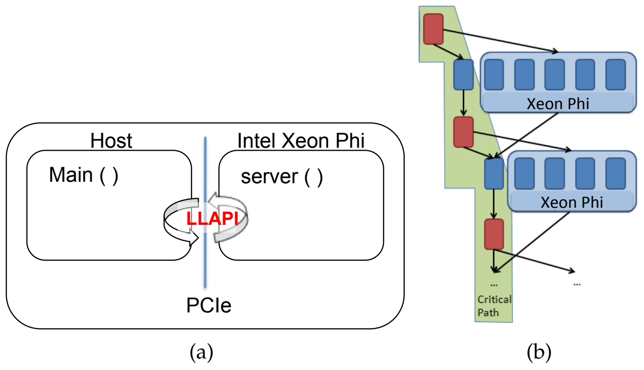


Figure 1: (a) MAGMA MIC programming model with a LLAPI server mediating requests between the host CPU and the Xeon Phi device. (b)DLA algorithm as a collection of BLAS-based tasks and their dependencies. The algorithm's critical path is, in general, scheduled on the CPUs, and large data-parallel tasks on the Xeon Phi.

- Large, parallelizable tasks are scheduled on Xeon Phi.

The difference with multicore algorithms is the task splitting, which here is of various granularities to make different tasks suitable for particular architectures, and the scheduling itself. Specific algorithms using this methodology, and covering the main classes of DLA, are described in the subsections below.

4.1 Design and functionality

The MAGMA interface is similar to LAPACK. For example, compare LAPACK's LU factorization interface vs. MAGMA's:

```
  lapackf77_dgetrf(&M,&N, hA,   &lda, ipiv, &info)
   magma_dgetrf_mic( M, N, dA,0, ldda, ipiv, &info,
queue)
```

Here hA is the typical CPU pointer (`double *`) to the matrix of interest in the CPU memory and dA is a pointer in the Xeon Phi memory (`magmaDouble_ptr`). The last argument in every MAGMA call is an Xeon Phi queue, through which the computation will be streamed on the Xeon Phi device (`magma_queue_t`).

To abstract the user from knowing low level directives, main functions, such as BLAS, CPU-Phi data transfers, and memory allocations and deallocations, are redefined in terms of MAGMA data types and functions. This design allows us to more easily port the MAGMA library to other device such as the GPU accelerator using either CUDA or OpenCL and eventually to merge them while maintaining a single source. Also, the MAGMA wrappers provide a complete set of functions for programming hybrid high-performance numerical libraries. Thus, not only users but application developers as well can opt to use the MAGMA wrappers. MAGMA provides the standard four floating point arithmetic precisions – single real, double real, single complex, and double complex. There are routines for the so called one-sided factorizations (LU, QR, and Cholesky), and recently we are developing the two-sided factorizations (Hessenberg, bi-, and tridiagonal reductions), linear system and least squares solvers, matrix inversions, symmetric and nonsymmetric standard eigenvalue problems, SVD, and orthogonal transformation routines.

4.2 LU, QR, and Cholesky factorizations

The one-sided factorization routines implemented and currently available through MAGMA are:

`magma_zgetrf_mic` computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges;

`magma_zgeqrf_mic` computes a QR factorization of a general M-by-N matrix A;

`magma_zpotrf_mic` computes the Cholesky factorization of a complex Hermitian positive definite matrix A.

Routines in all standard four floating point precision arithmetics are available, following LAPACK's naming convention. Namely, the first letter of the routine name (after the prefix `magma_`) indicates the precision – z, c, d, or s for correspondingly double complex, single complex, double real, or single real. The suffix `_mic` indicates that the input matrix and the output are on the Xeon Phi memory.

The typical hybrid computation and communication pattern for the one-sided factorizations (LU, QR and Cholesky) is shown in Figure 2. At a given iteration, panel dP is copied to the CPU and factored using LAPACK, and the result is copied back to GPU. The trailing matrix, consisting of the next panel T_1 and submatrix T_2, is updated on the GPU. After receiving dP back from the CPU, T_1 is updated first using dP and the result is sent to the CPU (as being the next panel to be factored there). While the CPU starts the factorization of T_1, the rest of trailing matrix, T_2, is updated on the GPU in parallel with the CPU factorization of panel T_1. In this pattern, only data to the right of the current panel is accessed and modified, and the factorizations that use it are known as right-looking. The computation can be organized

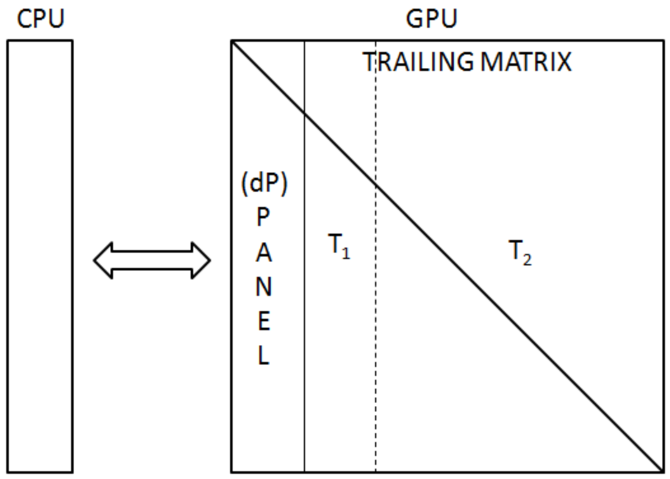
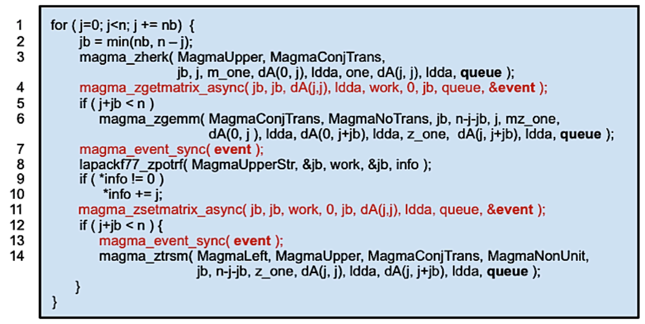


Figure 2: Typical computational pattern for the hybrid one-sided factorizations in MAGMA.



```
1   for ( j=0; j<n; j += nb) {
2         jb = min(nb, n – j);
3         magma_zherk( MagmaUpper, MagmaConjTrans,
                              jb, j, m_one, dA(0, j), ldda, one, dA(j, j), ldda, queue );
4         magma_zgetmatrix_async( jb, jb, dA(j,j), ldda, work, 0, jb, queue, &event );
5         if ( j+jb < n )
6             magma_zgemm( MagmaConjTrans, MagmaNoTrans, jb, n-j-jb, j, mz_one,
                                      dA(0, j ), ldda, dA(0, j+jb), ldda, z_one,  dA(j, j+jb), ldda, queue );
7         magma_event_sync( event );
8         lapackf77_zpotrf( MagmaUpperStr, &jb, work, &jb, info );
9         if ( *info != 0 )
10             *info += j;
11        magma_zsetmatrix_async( jb, jb, work, 0, jb, dA(j,j), ldda, queue, &event );
12        if ( j+jb < n ) {
13            magma_event_sync( event );
14            magma_ztrsm( MagmaLeft, MagmaUpper, MagmaConjTrans, MagmaNonUnit,
                                    jb, n-j-jb, z_one, dA(j, j), ldda, dA(j, j+jb), ldda, queue );
       }
  }
```

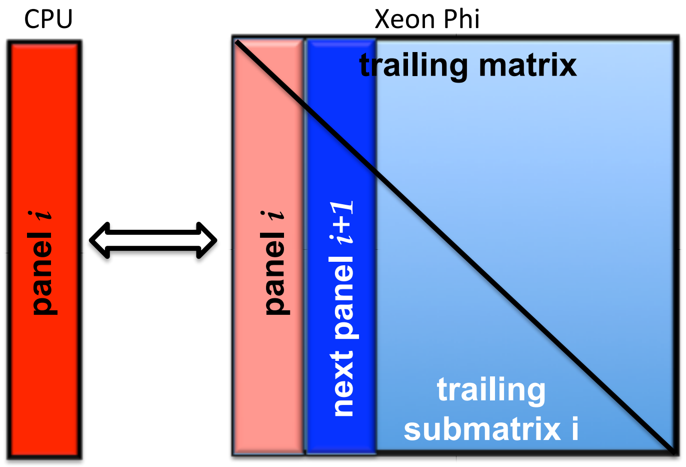
Figure 3: Cholesky factorization in MAGMA.

differently – to access and modify data only to the left of the panel – in which case the factorizations are known as left-looking.

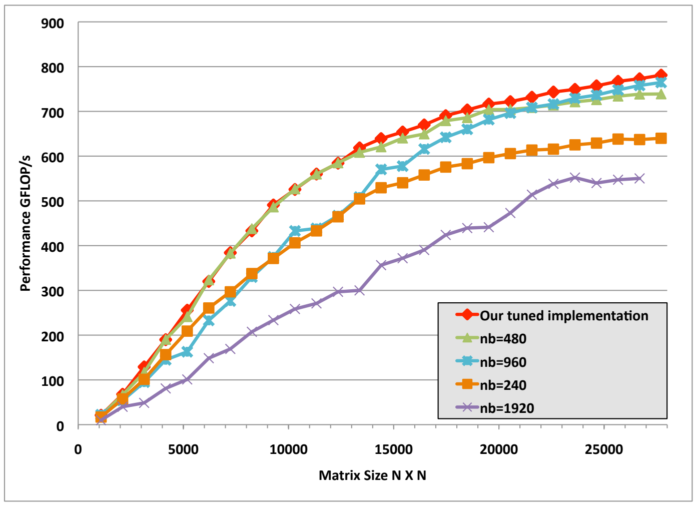
An example of a left-looking factorization, demonstrating a hybrid algorithm implementation, is given in Figure 3 for the Cholesky factorization. Copying the panel to the CPU, in this case just a square block on the diagonal, is done on line 4. The data transfer is asynchronous, so before we factor it on the CPU (line 8), we synchronize on line 7 to enforce that the data has arrived. Note that the CPU work from line 8 is overlapped with the GPU work on line 6. This is indeed the case because line 6 is an asynchronous call/request from the CPU to the GPU to start a ZGEMM operation. Thus control is passed to lines 7 and 8 while the GPU is performing the ZGEMM. The resulting factored panel from the CPU work is sent to the GPU on line 11 and used on line 14, after making sure that it has arrived (the sync on line 13).

4.3 Hybrid implementation and Optimization techniques

In order to explain our hybrid methodology and the optimization that we developed, let us give a detailed analysis for the QR decomposition algorithm. While the description below only addresses the QR factorization, it is straightforward to derive with the same ideas the analysis for both the Cholesky and LU factorizations. For that we start briefly by recalling the description of the QR algorithm.



(a) Typical computational pattern for the hybrid one-sided factorizations in MAGMA



(b) Effect of the blocking factor.

Figure 4: Effect of the blocking factor.

The QR factorization is a transformation that factorizes an $m \times n$ matrix A into its factors Q and R where Q is a unitary matrix of size $n \times n$ and R is a triangular matrix of size $m \times m$. The QR algorithm can be described as a sequence of steps where, at each step, a QR of a panel is performed based on accumulating a number of Householder transformations in what is called a "*panel factorization*" which are, then, applied all at once by means of high performance Level 3 BLAS operations in what is called the "*trailing matrix update*". Despite that this approach can exploit the parallelism of the Level 3 BLAS during the trailing matrix update, it has a number of limitations when implemented on massively multithreaded system such as the Intel Xeon Phi coprocessor due to the nature of its operations. On the one hand, the panel factorization relies on Level 2 BLAS operations that cannot be efficiently parallelized on either Xeon Phi or any accelerator such as GPU-based architectures, and thus it can be considered to be close to sequential operations that limit the scalability of the algorithm. On the other hand, this algorithm is referred as the *fork-join approach* since the execution flow will show a sequence of sequential operations (panel factorizations) interleaved with parallel ones (trailing matrix updates). In order to take advantage of the high execution rate of the massively multithreaded system, in particular, the Phi coprocessor we redesigned the standard algorithm in a way to perform the Level 3 BLAS operations (Trailing matrix update) on the Xeon Phi while performing the Level 2 BLAS operations (panel factorization) on the CPU. We also proposed an algorithmic change to remove the fork join bottleneck and to minimize the overhead of the panel factorization by hiding its costs behind the parallel trailing matrix update. This approach can be described as the "*scalable lookahead techniques*". Our idea is to split of the trailing matrix update into two phases, the update of the lookahead panel (panel of step $i+1$, i.e., dark blue portion of Figure 4a) and the update of the remaining trailing submatrix (clear blue portion of Figure 4a). Thus, during the submatrix update the CPU can receive asynchronously the panel $i+1$ and performs its factorization. As a result, our MAGMA implementation of the QR factorization can be described by a sequence of the three phases described below. Consider a matrix A that can be represented as:

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}, \qquad (1)$$

- **Phase 1, the panel factorization:** at a step i, this phase consists of a QR transformation of the panel $A_{i:n,i}$ as in Equation 2. This operation consists of calling two routines. The DGEQR2 that factorizes the panel and produces nb Householder reflectors (V_{*i}) and an upper triangular matrix R_{ii} of size $nb \times nb$, which is a portion of the final R factor, and the DLARFT that generates the triangular matrix T_{ii} of size $nb \times nb$ used for the trailing matrix update. This phase is performed on the CPU.

$$\begin{bmatrix} A_{11} \\ A_{21} \\ A_{31} \end{bmatrix} \Longrightarrow \begin{bmatrix} V_{11} \\ V_{21} \\ V_{31} \end{bmatrix}, [R_{1,1}), [T_{1,1}]. \qquad (2)$$

- **Phase 2, the look ahead panel update:** the transformation that was computed in the panel factorization needs to be applied to the rest of the matrix (trailing matrix, i.e., the blue portion of Figure 4a). This phase consists into updating only the next panel (dark blue portion of Figure 4a) in order to let the CPU start its factorization as soon as possible while the update of the remaining portion of the matrix is performed in phase 3. The idea is to hide the cost of the panel factorization. This operation presented in Equation 3, is performed on the Phi coprocessor and involves the DLARFB routine which has been redesigned as a sequence of DGEMM's to better take advantage of the Level 3 BLAS operations.

$$\begin{bmatrix} R_{12} \\ \tilde{A}_{22} \\ \tilde{A}_{32} \end{bmatrix} = \left[I - V_{*i} T_{ii}^T V_{*i}^T \right] \begin{bmatrix} A_{12} \\ A_{22} \\ A_{32} \end{bmatrix}. \qquad (3)$$

- **Phase 3, the trailing matrix update:** Similarly to phase 2, this phase consists into applying the Householder reflectors generated during the panel factorization of step i according to Equation 3, but to the remaining portion of the matrix (the trailing submatrix i.e., the clear blue portion of Figure 4a). This operations is also performed on the Phi coprocessor, while in parallel to it, the CPU performs the factorization of the panel $i+1$ that has been computed in Phase 2.

This hybrid technique of distribution of tasks between CPU-Phi allows us to hide the memory bound operations occurred during the panel factorization (Phase 1) by performing such operation on the CPU in parallel with the trailing submatrix update (Phase 3) on the Phi coprocessor. However, one of the key parameters to performance tuning is the blocking size as the performance and the overlap between the CPU-Phi will be solely guided by it. Figure 4b illustrates the effect of the blocking factor on the performance. It is obvious that, a small nb will reduce the cost of the panel factorization phase 1, but it decreases the efficiency of the Level 3 BLAS kernel of phase 2 and phase 3 and thus resulting a bad performance. As opposed, a large nb will dramatically affect the panel factorization phase 1 which becomes slow and thus the CPU/Phi computation cannot be overlapped, providing a deterioration in the performance as shown in Figure 4b. As a consequence, the challenging problem is the following: on the one hand, the blocking size nb needs to be large enough to extract high performance from the Level 3 BLAS phase 3 and on the other hand, it has to be small enough to extract efficiency (thanks to the cache speed up) from the Level 2 BLAS phase 1 and overlap CPU/Phi computation. Figure 4b show the performance obtained for different blocking sizes and we can see a trade-off between small and large nb's. Either $nb = 480$ or $nb = 960$ can be considered as a good choice because MKL Phi BLAS is optimized for multiples of 240. Moreover, to extract the maximum performance and allow the maximum overlap between both of the CPU and the Xeon Phi coprocessor, we developed a new variant that can use a variable nb during the steps of the algorithm. The flexibility of our implementation allows an efficient task execution overlap between the CPU host and the Phi coprocessor which enables the algorithm to scale almost perfectly in the Phi coprocessor and provides very good performance close to the practical peak obtained on such system. Our tuned variable implementation is represented by the red curve of Figure 4b where we can easily observe its advantage over the other variants.

4.4 Hardware Capability Task Distribution

Programming models that raise the level of abstraction are of great importance for reducing software development efforts. A traditional approach has been to organize algorithms in terms of BLAS calls, where hardware specific optimizations would be hidden in BLAS implementations such as Intel's MKL or AMD's ACML. This is still valid and used but has shown some drawbacks on new architectures. In particular, parallelization is achieved using a fork-join approach since

Algorithm 1: Two-phase factorization of $A = [P_1, P_2, \ldots]$ with lookahead of depth 1. Matrix A and the result are assumed to be on the MIC memory.

```
PanelStartReceiving $_{on\ CPU}(P_1)$ ;
for $P_i = P_1, P_2, \ldots$ do
    PanelFactorize $_{on\ CPU}(P_i)$ ;
    PanelSend $_{to\ MIC}(P_i)$ ;
    TrailingMatrixUpdate $_{on\ MIC}(P_{i+1})$ ;
    PanelStartReceiving $_{on\ CPU}(P_{i+1})$ ;
    TrailingMatrixUpdate $_{onMIC}(P_{i+2}, \ldots)$ ;
```

each BLAS call, e.g., a matrix-matrix multiplication, can be performed in parallel (fork) but a synchronization is needed before performing the next call (join). The number of synchronizations thus can become a prohibitive bottlenecks for performance on highly parallel devices such as the MICs. This type of programming has been popularized under the Bulk Synchronous Processing name [19, 18].

Instead, the algorithms (like matrix factorizations) are broken into computational tasks (e.g., panel factorizations followed by trailing submatrix updates) and pipelined for execution on the available hardware components (see below). Moreover, particular tasks are scheduled for execution on the hardware components that are best suited for them. Thus, this task distribution based on *hardware capability* allows the user for the efficient use of each hardware component. In the case of DLA factorizations, the less parallel panel tasks are scheduled for execution on multicore CPUs, and the parallel updates mainly on the MICs. We illustrate this in Algorithm 1.

4.5 Task Based Runtime Model

The scheduling of tasks for execution can be static or dynamic. In either case, the small and not easy to parallelize tasks from the critical path (e.g., panel factorizations) are executed on CPUs, and the large and highly parallel task (like the matrix updates) mostly on the MICs.

The use of multiple coprocessors complicates the development using static scheduling. Instead, the use of a lightweight runtime system is preferred as it can keep scheduling overhead low, while enabling the expression of parallelism through sequential-like code. The runtime system relieves the developer from keeping track of the computational activities that, in the case of heterogeneous systems, are further exacerbated by the separation between the address spaces of the main memory of the CPU and the MICs. Our runtime model is build on the QUARK [20] superscalar execution environment that has been originally used with great success for linear algebra software on just multicore platforms [13]. The conceptual work though could be replicated within other models such as StarPU [5], OmpSS [15], Cilk [8], and Jade [16], to just mention a few.

4.6 Improving offload mode communication

It is well known that the off-load transfer mode copies only continuous chunks of data from and to the coprocessors. However most of the scientific application algorithms require to exchange data with 2D or 3D storage and thus this may create an issue when using the off-load transfer mode. In particular, the one-sided factorizations (Cholesky, LU, and QR) require to send the panel to the CPU and then receive it later after being factorized by the CPU. A simple implementation loop over one direction and call the off-load section to send/receive a contiguous vector. Such implementation behaves poorly and as a result the communication will become expensive and slow down the algorithm. Indeed, another alternative is to copy the 2D panel to a contiguous temporary space on the MIC, and then to send it and vice versa. Hence, there are two points that need to be taken into consideration. Firstly, the copy needs to be implemented as a multi-threaded operation in order to hide its cost. For that, we implemented a parallel copy that uses all of the 240 hardware threads of the MIC to perform the copy. This might be against the common wisdom that multi-threading is of little help for bandwidth-limited operations such as a memory copy. This is not the experience on the MIC, where the clock frequency of the compute cores is twice as low as that of the memory – the exact opposite of what is the case in Intel x86 multicore processors. In addition to the low frequency, the current MIC hardware is to a large degree an in-order architecture with dual-pipeline execution and single-issue fetch/decode units [12] which poses constraints on the amount bandwidth that can be utilized from a single core. These can be overcome in multiple ways, including the use of streaming loads and have the multiple threads requesting data. Secondly, when the MIC copies data to or from the temporary space, it should be the only kernel running, otherwise, it will run on top of other kernel running and this may slow down both of the kernels. For that, we represented the copy kernel as a task with high priority and the scheduler is responsible for executing it as soon as possible and to handle the dependencies such as no other kernel will be running at the same time. Experiments showed that when using these optimizations the performance of the off-load communication mode is comparable to both the SCIF and the COI mode with a variance of less than 5%.

4.7 Trading Extra Computation for Higher Execution Rate

The optimization discussed here is MIC-specific but is often valid for any hardware architecture with multilayered memory hierarchy. The dlarfb routine used by the QR decomposition consists of two dgemms and one dtrmm. Since coprocessors are better at handling compute-bound tasks, for computational efficiency, we replace the dtrmm by dgemm, yielding 5-10% performance improvement. For the Cholesky factorization, the trailing matrix update requires a dsyrk. Due to uneven storage, the multi-device dsyrk cannot be assembled purely from regular dsyrk calls on each device. Instead, each block column must be processed individually.

The diagonal blocks require special attention. One can use a dsyrk to update each diagonal block, and a dgemm to update the remainder of each block column below the diagonal block. The small dsyrk operations have little parallelism and therefore their execution is inefficient on MICs. This can be improved to some degree by using pragma to run several dsyrk's simultaneously. Nevertheless, because we have copied the data to the device, we can consider the space above the diagonal to be a scratch workspace. Thus, we update the entire block column, including the diagonal block, writing extra data into the upper triangle of the diagonal block, which is subsequently ignored. We do extra computation for the diagonal block, but gain efficiency overall by launching fewer BLAS kernels on the device and using the more efficient dgemm kernels, instead of small dsyrk kernels, resulting in overall 5-10% improvement in performance.

5 Performance Results

This section presents the performance results obtained by our hybrid CPU-Xeon Phi implementation in the context of the development of the state-of-the-art numerical linear algebra libraries.

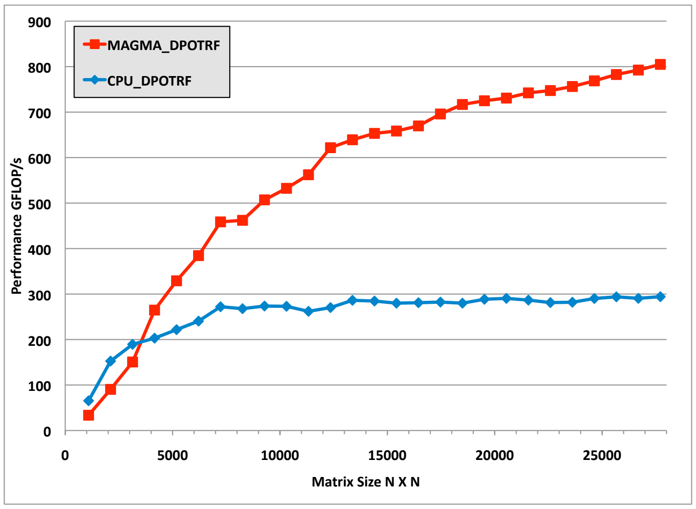
5.1 Experimental Environment

Our experiments were performed on a system equipped with Intel Xeon-Phi. It is representative of a vast class of servers and workstations commonly used for computationally intensive workloads. We benchmarked all implementations on an Intel multicore system with dual-socket, 8 core Intel Xeon E5-2670 (Sandy Bridge) processors, each running at 2.6 GHz. Each socket has a 24 MB shared L3 cache, and each core has a private 256 KB L2 and 64 KB L1. The system is equipped with 52 Gbytes of memory. The theoretical peak for this architecture in double precision is 20.8 Gflop/s per core, giving 332 Gflops in total. The system is also equipped with an Intel Xeon Phi cards with 7.7 Gbytes per card running at 1.09 GHz, and giving a double precision theoretical peak of 1046 Gflops.

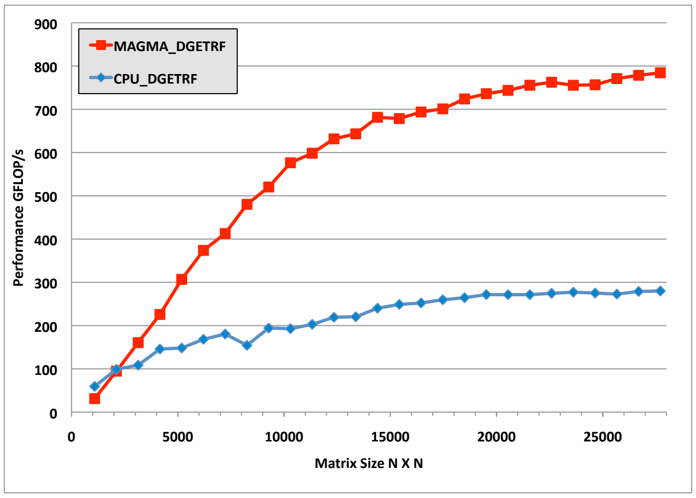
There are a number of software packages available. On the CPU side we used the MKL (Math Kernel Library) [11] which is a commercial software package from Intel that is a highly optimized numerical library. On the Intel Xeon side, we used the MPSS 2.1.5889-16 as the software stack, icc 13.1.1 20130313 which comes with the composer_xe_2013.3.163 suite as the compiler and the BLAS-3 routine GEMM from MKL 11.00.03.

5.2 Experimental results

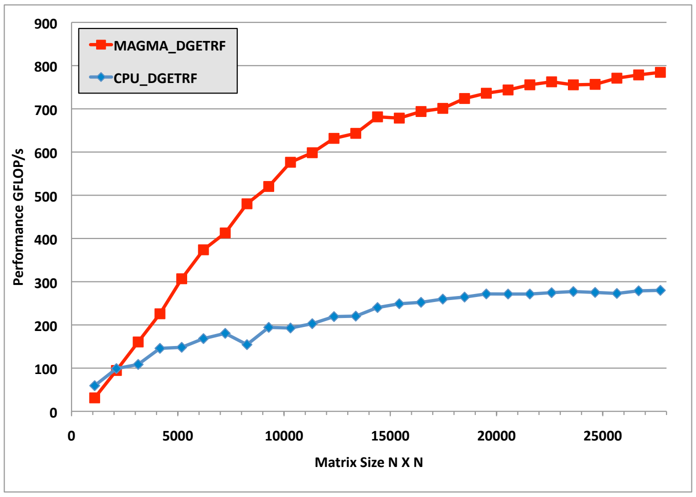
Figure 5 reports the performance of the three amigos linear algebra kernels, the Cholesky, QR and LU factorizations with our hybrid implementation and compare it to the performance of the CPU implementation of the MKL libraries. For our implementation, the blocking factor has been chosen to be flexible in order to achieve the best performance, as a reference it is in the range of 480-960 as described in



(a) Cholesky factorization



(b) LU factorization



(c) QR decomposiiton

Figure 5: Comparison of the performance versus the optimized CPU version of the MKL libraries for the three amigos.

section 4.3. The graphs show the performance measured using all the cores available on the system (i.e., 60 for the Intel Phi and 16 for the CPU) with respect to the problem size. In order to reflect the time completion, for each algorithm the operation count is assumed to be the same as that of the LAPACK algorithm (i.e., $\frac{1}{3}N^3$, $\frac{2}{3}N^3$, and $\frac{4}{3}N^3$ for the Cholesky factorization, the LU factorization and the QR decomposition respectively)

Figure 5a, 5b and 5c provide roughly the same information: our MAGMA algorithm with hybrid techniques delivers higher execution rates than the CPU optimized counterpart. Such comparison is not fair, our goal is not to compare, but it is rather to show the boost that the hybrid CPU+Phi coprocessor implementation provides, versus a CPU implementation. The figures show that the MAGMA hybrid algorithms are capable of completing any of the three amigos algorithms as twice faster as the CPU optimized version for a matrix of size larger than 10000; and more than three times faster when the matrix size is large enough (larger than 20000). The actual curves of Figure 5 illustrates the efficiency of our hybrid techniques where we note that the performance obtained by our implementation, achieves a very close level to the practical peak of the Intel Xeon Phi coprocessor computed by running the GEMM routine (which is around 850 Gflops). This gain is mostly obtained by two improvements. First the nature of the operations involved in the Phi side which are mostly BLAS Level 3 operations redesigned and redeveloped as a combination of DGEMM's. For more details we denote below the routines executed on the Xeon Phi coprocessor:

- The DSYRK operations for the Cholesky factorization where the DSYRK has been redesigned as a combination of DGEMM's routines,
- The DGEMM for the LU factorization,
- The DLARFB for the QR decomposition where also its has been redesigned as a combination of DGEMM's.

Second, all of the Level 2 BLAS routines that are memory bound and that represent a limit for the performance (i.e., DPOTF2, DGETF2, and DGEQR2 for Cholesky, LU, and QR factorization respectively) are executed on the CPU side while being overlapped with the Phi coprocessor execution as described in section 4.3.

An important remark has to be made here for the Cholesky factorization: the *left-looking* algorithm as implemented in LAPACK is considered as well optimized for memory reuse but at the price of less parallelism and thus is not suitable for massively multicore machines. This variant delivers poor performance when compared to the *right looking* variant that allows more parallelism and thus run at higher speed.

6 Conclusions and Future Work

In this article, we have shown how to extend our hybridization methodology from existing systems to a new hardware platform. The challenge of the porting effort stemmed from the fact that the new coprocessor from Intel, the Xeon Phi, featured programming models and relative execution overheads, that were markedly different from what we have been targeting on GPU-based accelerators. Nevertheless, we believe that the techniques used in this paper adequately adapt our hybrid algorithm to best take advantage of the new heterogeneous hardware. We have derived an implementation schema of the dense linear algebra kernels that also can be applied to either the two-sided factorization used for solving the eigenproblem and the SVD or to the sparse linear algebra algorithms. We plan to further study the implementation of multi-Xeon Phi algorithms in a distributed computing environment. We think that the techniques presented will become more popular and will be integrated into dynamic runtime system technologies. The ultimate goal is that this integration will help to tremendously decrease development time while retaining high-performance.

Acknowledgments

The authors would like to thank the National Science Foundation, the Department of Energy and ISTC for Big Data for supporting this research effort.

References

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys.: Conf. Ser.*, 180(1), 2009.

[2] AMD. AMD Core Math Library (ACML). Available at http://developer.amd.com/tools/.

[3] E. Anderson, Z. Bai, C. Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, Third edition, 1999.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[5] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[6] Barcelona Supercomputing Center. *SMP Superscalar (SMPSs) User's Manual, Version 2.0*, 2008. http://www.bsc.es/media/1002.pdf.

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997. http://www.netlib.org/scalapack/slug/.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.

[9] J. Dongarra, J. Bunch, C. Moler, and GW. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, PA, 1979.

[10] Intel. Intel xeon phi coprocessor system software developers guide. http://software.intel.com/en-us/articles/.

[11] Intel. Math Kernel Library. Available at http://software.intel.com/en-us/articles/intel-mkl/.

[12] Jim Jeffers and James Reinders. *Intel® Xeon Phi^{TM}Coprocessor High-Performance Programming*. Morgan Kaufmann Publishers, 2013.

[13] J. Kurzak, P. Luszczek, A. YarKhan, M. Faverge, J. Langou, H. Bouwmeester, and J. Dongarra. Multithreading in the PLASMA Library. In *Handbook of Multi and Many-Core Processing: Architecture, Algorithms, Programming, and Applications*, Computer and Information Science Series. Chapman and Hall/CRC, April 26 2013.

[14] Software distribution of MAGMA MIC version 1.0. http://icl.cs.utk.edu/magma/software/, May 3 2013.

[15] Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*, pages 142–151. IEEE, 2008.

[16] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: a high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, 1993. DOI: 10.1109/2.214440.

[17] Stanimire Tomov and Jack Dongarra. Dense linear algebra for hybrid GPU-based systems. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, *Scientific Computing with Multicore and Accelerators*. Chapman and Hall/CRC, 2010.

[18] L. G. Valiant. Bulk-synchronous parallel computers. In M. Reeve, editor, *Parallel Processing and Artificial Intelligence*, pages 15–22. John Wiley & Sons, 1989.

[19] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), Aug. 1990. DOI 10.1145/79173.79181.

[20] Asim YarKhan. *Dynamic Task Execution on Shared and Distributed Memory Architectures*. PhD thesis, University of Tennessee, December 2012.