LU, QR, and Cholesky factorizations: Programming Model, Performance Analysis and Optimization Techniques for the Intel Knights Landing Xeon Phi

Azzam Haidar, Stanimire Tomov
University of Tennessee
Knoxville, TN 37916

Konstantin Arturov
Intel Corporation
Novosibirsk, Russia

Murat Guney, Shane Story
Intel Corporation
Hillsboro, OR 97124

Jack Dongarra
University of Tennessee, Knoxville
Oak Ridge National Laboratory, USA
University of Manchester, UK

Abstract— **A wide variety of heterogeneous compute resources, ranging from multicore CPUs to GPUs and coprocessors, are available to modern computers, making it challenging to design unified numerical libraries that efficiently and productively use all these varied resources. For example, in order to efficiently use Intel's Knights Langing (KNL) processor, the next-generation of Xeon Phi architectures, one must design and schedule an application in multiple degrees of parallelism and task grain sizes in order to obtain efficient performance. We propose a productive and portable programming model that allows us to write a serial-looking code, which, however, achieves parallelism and scalability by using a lightweight runtime environment to manage the resource-specific workload, and to control the dataflow and the parallel execution. This is done through multiple techniques ranging from multi-level data partitioning to adaptive task grain sizes, and dynamic task scheduling. In addition, our task abstractions enable unified algorithmic development across all the heterogeneous resources. Finally, we outline the strengths and the effectiveness of this approach – especially in regards to hardware trends and ease of programming high-performance numerical software that current applications need – in order to motivate current work and future directions for the next generation of parallel programming models for high-performance linear algebra libraries on heterogeneous systems.**

I. Introduction

Hardware technologies based on many-core architectures have gained ground not just in terms of computational performance but also in terms of ease of use, e.g., due to software technologies such as OpenMP, CUDA, OpenCL, and OpenACC. The Many Integrated Core (MIC) technology offered by Intel, incorporated in the Xeon Phi product family, is well established, offering a high-performance computational edge, as well as a dominant software stack and support for open standards, e.g., OpenMP, that have served the community well in terms of productivity to render auxiliary the burden of using lower level libraries, like SCI and COI. We refer to this match of hardware with its software stack as the vertical integration of a platform.

Interestingly, horizontal integration, or combining different platforms and their software stacks is much less supported or researched. Yet, modern computing systems are likely to feature multiple accelerators and/or coprocessors, often not the same kind, in order to accommodate the whole variety of scientific workloads.

Finally, the constant technological progress of hardware accelerators, now featuring nearly 10 billion transistors, foreshadowed the advancement in traditional multicore CPU technology. A CPU can now produce half a Tera-flop of computations per second in double precision on a single motherboard socket. This CPU performance was made possible by the fast paced improvement in CPU vectorization standards with the flagship AVX and multi-argument fused multiply-add instructions coupled with multiple floating units per core that (with proper use of Level 1 cache) can sustain very high performance without being burdened by the vagaries of modern memory systems and NUMA overheads.

We are presenting a programming model for the rapid development of linear algebra applications designed to be efficient on complex heterogeneous hardware. Our programming model derives its productivity from a design that is based on the task-superscalar paradigm, allowing us to write serial-like code. Then, a task-superscalar runtime executes the code adaptively, scalably, and efficiently on the available heterogenous resources.

II. Background and Related Work

Dense linear algebra, as available in LAPACK and ScaLAPACK, has been modernized for new architectures in libraries like PLASMA and MAGMA [2], as well as in vendor libraries. Key challanges for the development have included programming models and algorithm designs to handle parallelism, heterogeneity, and communication. In MAGMA, similarly to vendor libraries, these challanges have been targeted for specific hardware configurations, e.g., multicore plus GPUs [22], [8], [11], GPUs only [13], embedded systems [16], multicore plus Xeon Phi coprocessors [14], etc. Recently, a unified framework was developed that can handle a mix of multicore CPUs, GPU accelerators, and Xeon Phi coprocessors (KNC) [12], [17], [15] in a single heterogeneous system. In contrast, here we further advance this current state-of-the-art LU, QR, and Cholesky factorization techniques by adding efficient support for KNL, the next generation of Xeon Phi processors. While previous Xeon Phi work concentrated on hybrid factorizations models that use multicore CPUs for the panel factorizations and KNC for the trailing matrix updates, the KNL self-boot packaging requires that the entire computation be done on the KNL, including the communication intensive panels.

Related to parallel programming models, the tasking approach and the use of BLAS are well established for dense linear algebra [1]. To provide parallelism, algorithms are split into computational tasks, which in the context of LAPACK algorithms translates to splitting BLAS calls into tasks. The resulting algorithms can be viewed as DAGs that can be scheduled for execution on the underlying hardware in various ways. Of particular interest are task-superscalar approaches that take serial code as input and result in a parallel execution, inferring the data dependencies between the tasks at runtime, generally using task superscalar techniques, for example as in Jade [20], Cilk [5], OpenMP 4.0 [7], Sequoia [9], SuperMatrix [6], OmpSS [19], Habanero [4], StarPU [3], QUARK [23], or the DepSpawn [10] project.

III. Hardware Description and Intel Library

We conducted our experiments on three different systems, denoted A, B, and C. They are representative of a vast class of servers and workstations commonly used for computationally intensive workloads.

- System A is equipped with two 8-core Intel Xeon E5-2670 (Sandy Bridge) processors, and two Intel Xeon Phi 7120 (KNC) cards with 15.8 GB per card, running at 1.23 GHz, and achieving a double precision theoretical peak of 1180 Gflop/s. Cards are connected via two PCIe I/O hubs at 6 GB/s bandwidth.
- System B has two 18-core Intel Xeon E5-2697 (Broadwell) processors, running at 2.6 GHz. Each socket has 35 MiB of shared L3 cache, and each core has a private 3.5 MiB L2 and 448 KB L1 cache. The system is equipped with 52 GB of memory and the theoretical peak in double precision is 20.8 Gflop/s per core.
- System C is the selfhosted preproduction Intel Xeon Phi Knights Landing (KNL) card, with 16 GB of MCDRAM used in *flat* mode, running at 1.3 GHz and achieving a double precision the theoretical peak of 2662 Gflop/s. It consists of 64 cores with 4 hyper-threads each. There are also card that came with 68 or 72 cores but are not included into our experiments.

We used the Intel MKL (Math Kernel Library) [18] (2017b1_20160506) , and Intel C compiler in the Intel Composer XE 2016 suite. Intel MKL is optimized for all Intel Xeon and Xeon Phi architectures. The Intel MKL team puts additional effort into optimizing the Level 3 BLAS and the LAPACK LU, Cholesky and QR factorizations as these are some of the most commonly used routine in the library.

IV. Flexible and Portable Heterogeneous Programming Model

Making an algorithm work with heterogeneous hardware components can be challenging. We discuss a programming model that provides high-level abstractions for programming multi-way heterogeneous resources. This allows us to have a unified approach and a seamless porting of the algorithms across a wide range of hardware. The use of 1) algorithmic blocking techniques, 2) $1-D$ *block-column* data distribution guided by hardware-capabilities-weight, and 3) optimized kernels, also enables an ease of programming (algorithms can be expressed through building blocks) as well as efficiency in using a wide range of recent architectures. To be fast, reliable, and efficient, we take advantage of a runtime system that allows us to write serial code while extracting parallelism and enabling adaptive execution on the available resources. This work builds on our earlier work on programming for multi-way heterogeneous architectures [12], but differs here by extending it to a native computational mode, as well as extending compatibility to handle new architectures, including the KNL processors.

A. Algorithmic Advancements

Here we present the linear algebra aspects of our generic solution for development of the one-sided factorizations: Cholesky (Chol), Gaussian (LU), and the Householder QR. Algorithmically, as presented in Algorithm 1 and illustrated in Figure 1, these factorizations can be viewed as a sequence of steps with two distinct phases per step: 1) a panel factorization that affects the data depicted by the blue portion of Figure 1, and, 2) a trailing matrix update that updates data represented by the magenta and green color in Figure 1. Table I shows the BLAS and LAPACK routines that must be substituted for the generic routines named in the algorithm. From a hardware

```
for $P_i \in \{P_1, P_2, \ldots, P_n\}$ do
    PanelFactorize($P_i$)
    TrailingMatrixUpdate($A^{(i)}$)
```

Algorithm 1: Two-phase matrix factorization.

	Cholesky	Householder	Gaussian
`PanelFactorize`	xPOTF2 xTRSM	xGEQF2 xLARFT	xGETF2 xLASWP
`TrailingMatrixUpdate`	xSYRK xGEMM	xLARFB	xTRSM xGEMM

TABLE I. Panel factorization and trailing matrix update routines.

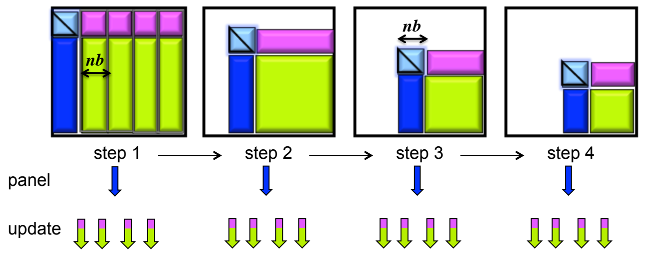

Fig. 1. Two-phase implementation of a one-sided factorization.

standpoint, the increased computational capability requires an incredible increase in the amount of concurrency that a software must be able to utilize. This, which in turn, requires the hardware/software not only to exploit unprecedented amounts of parallelism at the algorithm level, but also Multi-way Heterogeneous Programing through the Master-Devices approach that we propose and describe in the next section. From a software point of view, we know that from the routines described in Table I, the `PanelFactorize` is memory-bound, while the `TrailingMatrixUpdate` is compute-bound. Thus, one can expect inefficiency of the simple loop of Algorithm 1 due to the nature of the `PanelFactorize` phase. As a consequence, the algorithm must be modified further in order to overcome this issue and to achieve closer to optimal performance. The first optimization is to hide the inefficiency of the memory-bound task (e.g., the `PanelFactorize` phase). A common technique to achieve this is to use *lookahead* [21]. In other words, the idea is to split the update phase of step i into an update of the next panel $UnextP$ (the first green block from the left of Figure 1), and an update of the rest of the trailing matrix $Urest$ (the green blocks to the right). Thus, once the update of the next panel is done, its `PanelFactorize` phase of step $i+1$ can start in parallel with the update of the rest of the trailing matrix $Urest$, hiding its memory-bound behavior. The Multi-way Heterogeneous Programing model described in the next subsection provide an easy way to perform these two operations in parallel. Figure 3 shows the execution trace of the LU factorization on the KNL where we can see that the panel factorization, illustrated in red, is computed in parallel, with the trailing matrix update depicted in green.

The efficient use of multiple computational devices for Algorithm 1 strongly depends on the data distribution of the matrix. The `PanelFactorize` phase operates on a *block-column* of data (cf., the blue blocks of Figure 1), and thus it is preferable for its data to be located in the same memory space in order to avoid communications, and to increase data locality, and cache reuse. The `TrailingMatrixUpdate` phase requires data from the panel and the top block of *nb* rows, and in the QR case a summation over the columns of the trailing matrix, which makes a *block-row* data distribution a bad candidate. For example, the xGEMM routine of the LU update (green blocks of Figure 1) requires data from the output of the xTRSM magenta blocks of Figure 1. We easily deduce that a *block-column* data distribution is preferable for both phases. Thus, based on the analysis of all the routines described in Table I, we concluded that an optimal distribution that minimizes the communications is a $1D$ *block-column* data distribution. Note that in contrast to the $2D$ data distribution, well known from the distributed memory ScaLAPACK, here we are in shared memory, and the number of targeted devices is in general less than 10. It turns out that the benefits of a $2D$ distribution (to keep load balance throughout the computation) cannot overcome the overheads of the extra communications and synchronizations associated with it on shared memory systems. This is illustrated by the performance experiments in Figures 4, 5, and 6. We see that two KNCs (purple curves) reach twice the performance of one KNC (cyan curve), i.e., a perfect scalability, and performance is close to the theoretical peak. This shows that the optimization techniques cited above are well implemented, and that the panel computation and the CPU-KNC communications are overlapped with the trailing matrix update phase. More details are provided in Section VI. We propose a $1D$ hardware-capabilities-weight *block-column* distribution, which distributes the data across the devices based on their computing capabilities (in a $1D$ *block-column* fashion). For more details about the hardware-capabilities-weight distribution, we refer to [12]. While the main purpose of this paper is a self-boot single KNL, we mentioned the importance of the data distribution since, as described next, a single device can be viewed as many, and thus the data distribution analysis still holds, where communication becomes now between cores on the same ship and reflect data movements between low level memory such as L2 cache.

B. Programming Multi-way Heterogeneous Resources

Developing software that properly maps algorithmic requirements to the specific strengths of the hardware components requires the development of heterogeneous algorithms. Xeon Phi and GPUs have high computational peaks compared to multicore CPUs. The difference in capabilities makes it challenging to develop a portable algorithm that can achieve high performance, reach good scalability in a multi-way heterogeneous environment, while also being easy to use, modify, and optimize. For example, computations on the critical path of an algorithm (mainly memory-bound operations like the `PanelFactorize` phase) may be more suitable to run on a small number of cores — the well known way is to run it on multicore CPUs– than on high-throughput computing devices such as GPUs or Xeon Phis, which are more suitable for highly-parallel computations as in the `TrailingMatrixUpdate` phase. This is what we call hybrid mode — where CPUs work together with accelerators. We have demonstrated the methodology of developing these types of hybrid algorithms in the MAGMA Library [12], which is widely used and referenced by industry (NVIDIA, Intel, AMD, and MathWorks) and application developers from the scientific community.

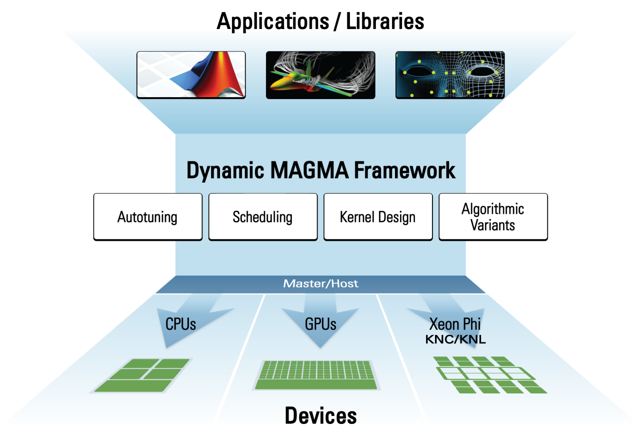

Fig. 2. The design of our Unified Programming Model.

We build on the same approach by: 1) extending it to native mode (where an algorithm runs on the same type of hardware, e.g., the KNL), and 2) supporting more hardware, such as multicore CPUs and the recent self-hosted KNL. Using the whole accelerator to compute a memory-bound task that is on the critical path of an algorithm is not a good idea, as presented in the previous section, since this kind of task does not exhibit enough parallelism. Therefore, we propose a programming model based on what we call a virtual view of the hardware. The hardware (GPUs, Xeon Phi KNC, Xeon Phi KNL, and multicore CPUs) can be viewed/modeled as a Master or Host computing unit with low capability that is responsible for tasks that are memory-bound, and other Device units or workers that are for compute-intensive tasks, as illustrated in Figure 2. This way, we can represent any native mode as a virtual hybrid mode even within the same hardware. For example, a KNL with 64 cores can be viewed as a Master using 4 cores, and one Device using 60 cores. Another possible configuration is a Master using 4 cores along with 6 Devices using 10 cores each, etc. Hence, the techniques proposed in Section IV-A can be easily developed now using this Master-Device design. For the rest of the paper, we just have two computing types, a **Master** unit and one or many **Device** units, independently from the associated hardware.

```
Task_Flags panel_flags = Task_Flags_Initializer
Task_Flags update_flags = Task_Flags_Initializer
Set the priority of the panel task
TaskFlagSet(&panel_flags, PRIORITY, 10)
Panel is memory-bound → locked to Master and disable task stealing
TaskFlagSet(&panel_flags, OPTYPE, BLAS2, DEVTYPE, Master)
Update is compute-intensive → preference to Device
TaskFlagSet(&update_flags, OPTYPE, BLAS3, DEVTYPE, Device)
for $k \in \{0, nb, 2 \times nb, \ldots, n\}$ do
    Factorization of the panel A(k:n,k:k+nb)
    TASK: getf2(A(k:n,k:k+nb))
    Swap the rows to the left and the right of the panel
    TASK: laswp(A(k:n,1:k))
    TASK: laswp(A(k:n,k+nb:n))
    for $j \in \{k+nb, k+2nb, \ldots, n-nb\}$ do
        if $j = k+nb$ then
            TaskFlagSet(&update_flags, PRIORITY, 10)
        TASK: trsm(A(k:k+nb,k:k+nb) → A(k:k+nb,j:j+nb)
        if $panel\_m > panel\_n$ then
            TASK: gemm( A(j:n,k:k+nb) ×
            A(k:k+nb,j,j+nb) → A(j:n,j:j+nb) )
```

Algorithm 2: LU implementation for multiple devices.

The key features taken into account by our model are the capabilities of the computational resources, the memory access, and the communication cost. We have developed a strategy that prioritizes the data-intensive operations to be executed by the Devices and the memory-bound ones by the Master. Moreover, we redesigned the kernels and implemented dynamically guided data distribution to exploit parallelism in order to keep the Devices busy. From a programming model point of view, each algorithm is converted into a Master part and a Device part. The routines destined to execute on the Devices must be extracted into a separate hardware-specific kernel function. The kernels may need to be optimized for the Device, e.g., including unrolling loops, replacing some memory-bound operations with compute-intensive ones even if it has a marginal extra cost, and also arranging operations to use the Device memory efficiently. The Master must manage the Device memory allocation, the Master-Device data movement, and the kernel invocation. We used a runtime engine in order to present a much easier programming environment and to simplify scheduling. This often allows us to maintain a single source version that handles different types of hardware either independently, or mixed together. Our goal is to abstract hardware details, while still maintaining fine levels of control.

Algorithm 2 shows the pseudocode for the LU factorization. It consists of a sequential code that is simple to comprehend and is independent of the architecture. Each call represents a task that is inserted into the scheduler, which stores it to be executed when all of its dependencies are satisfied. Each task by itself consists of a call to a kernel function that could either be a Master or a Device function. We tried to hide the differences between hardware and to allow the scheduler engine to handle the transfer of data asynchronously and automatically, when needed (meaning when Master and Device do not share the same memory). We have proposed a set of scheduling directives (such as DEVTYPE, PRIORITY, and OPTYPE flags) that are evaluated at runtime in order to fully map the algorithm to the hardware, and to run close to the peak performance of the system. DEVTYPE specify the type of of the device, OPTYPE specify the operation type (e.g., memory-bound BLAS2 or compute-intensive BLAS3) while PRIORITY specify the priority of the task. Using these strategies, we can easily develop simple and portable code that can run on different heterogeneous architectures, letting the scheduling and execution engine do the task dependency analysis, resource scheduling, and finally, the task execution. A simple example of these functionalities is the implementation of the lookahead technique that does not requires any extra programming effort. The first task of the trailing matrix update phase (trsm and gemm) consists of the update of the next panel. Since it is a priority task, the scheduling engine ensures that the scheduler places it at the top of the queue as a priority task (since it is on the critical path), tracks its dependencies, and once finished, sends it to the Master in order to perform the panel factorization of the next step, while the accelerator Device continues the update of the trailing matrix of step k. This technique is called lookahead, and is hidden here by the scheduler without any extra lines of code. Figure 3 shows the execution trace of the LU factorization on the KNL. We see that the panel factorization task of step $i+1$ runs in parallel with the update of the rest of the trailing matrix of step i, allowing lookahead.

C. Adaptive Multi-grain Scheduling

In this section we describe the techniques that we used to provide an adaptive, scalable, high performance execution in a multi-way heterogeneous environment. Further details and experiments can be found in our earlier work [12]. There are not many restrictions on the user code, and a sample is presented in Algorithm 2. The user is responsible for providing computational fragments as tasks, defining dataflow dependences between tasks, and indicating task priority as well as its type: memory- or compute-bound (BLAS2 or BLAS3, respectively). The scheduler will take over the execution of tasks by first placing them in the appropriate queue (according to task type and available resources). For an efficient execution, the tasks need to be assigned to the computational resources, taking into account the varying computational differences between the resources. The selection of the right queue takes into account the length of each queue, which reflects the current and future load of the Device, and the computational capacity of the Device. This is achieved by assigning tasks to Device bins with a greedy heuristic. In order to keep a measure of the difference between the resources, for each Device i and each kernel type k, we maintain an α_{ik} parameter which corresponds to the effective performance rate that can be achieved on that Device. This α_{ik}, also referred to as a **resource capability weight for the task**, can be provided by the user via a task-flag, or could potentially be estimated by the runtime environment. As an example, the capability-weights for the update operation (a Level 3 BLAS) is around 1 : 10, which means that the GPU can execute 10 times as many update tasks as the CPU. The tasks are scheduled using **adaptive scheduling with capability weights**. As a task is inserted into the runtime, it is assigned to the resource with the largest remaining capability-weights. This greedy heuristic takes into account the capability-weights of the resource as well as the current number of waiting tasks *preferred* to be executed by this resource. For example, the panel tasks are memory-bound and thus are preferentially executed on the Master. The adaptive heuristic tries to maintain the ratios of the capability-weights across all the resources to maintain a balanced execution time. Finally, the task is inserted into the appropriate queue according to its priority to allow for progress along the critical path of the task graph. The priorities are user-defined and are optional, but in practice, they can give a performance advantage for some workloads. Similarly, appropriate allocation of hardware resources to the scheduler devices can be regarded as a tuning option. Once the tasks are scheduled for execution, we provide **transparent data movement** through the runtime. If the runtime detects that the data required for a task is not available at the location that the task is scheduled, it manages the data transfer. The advantage of such a strategy is not only to hide the data transfer cost between the Master and Devices (since it is overlapped with the Devices computation), but also to keep the Devices busy by providing enough tasks to execute.

V. Flexible Design and Robust Tuning Process

The proposed programing model also allows us to have a flexible resource management and robust tuning process. Thus, keeping a consistent interface that remains the same for users, independent of scale and processor heterogeneity, but which achieves good performance and efficiency by binding to different underlying code, depending on the configuration. For example, consider a KNL in native mode and construct two configurations – one with a 4 core Master and a 60 core Device, and a second one with 4 Devices with 15 cores each. Figure 3 shows the execution trace of the LU factorization for both configurations. Note that performance is similar, with less than a 5% difference. This attractive observation shows that when the kernels of the update routine are optimized to perform well on large number of cores, we do not need to split the hardware over many Devices; only one Device is enough to achieve very good performance and reach close to the peak. This scenario simplifies the tuning process (reducing the number of configurations), minimizes the effort required for the hardware-specific kernel optimizations (since update kernels need to be optimized for a large number of cores), decreases the scheduler work, and also makes the execution trace easy to understand, which in turn simplifies the performance analysis and the debugging process.

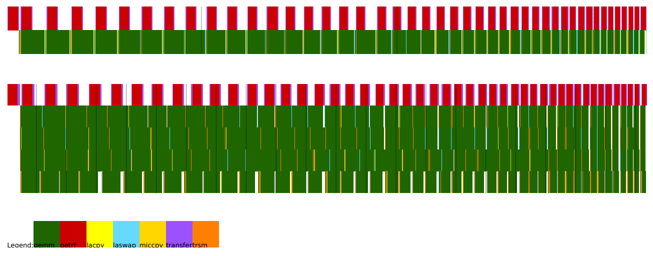

Fig. 3. Execution traces for the LU factorization on KNL (64 cores) viewed as a Master using 4 cores and either one Device of 60 cores (top) or four Devices of 15 cores each (bottom). The panel factorization is represented in red (Master tasks) and the update by the green color (Devices tasks).

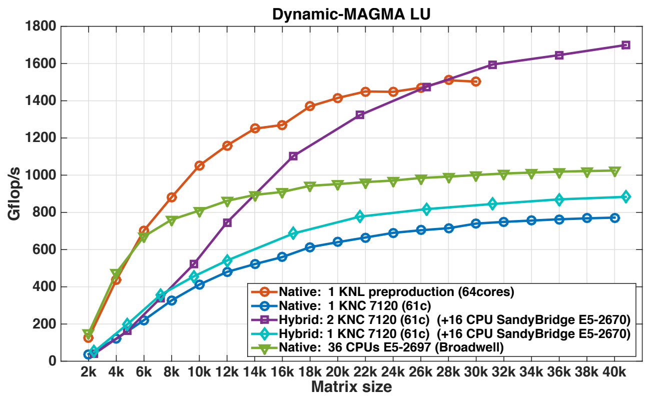

Fig. 4. Performance of LU across five hardware configurations.

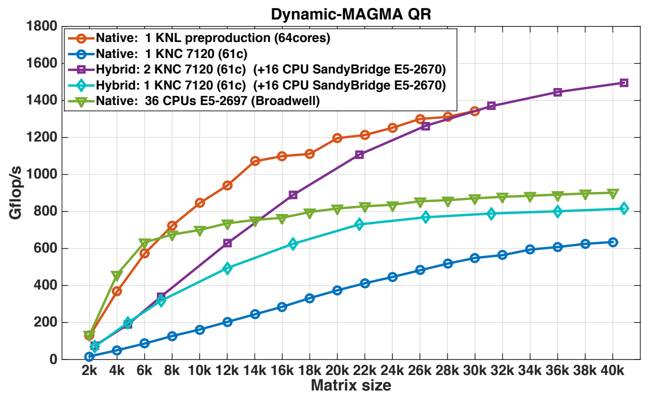

Fig. 5. Performance of QR across five hardware configurations.

VI. Performance Results

Figures 4, 5, and 6, already mentioned in the previous sections, illustrate the performance results in double precision (DP) arithmetic for the LU, QR, and Cholesky factorizations, respectively, for three types of hardware and in both hybrid and native configurations. We use the same code to show its portability, sustainability, and ability to provide close to peak performance when used in native mode, on a single KNC (the blue curve), self-hosted preproduction single KNL (the red curve), 36 Broadwell CPUs only (the green curve), as well as in hybrid mode on 16 Sandy Bridge E5-2670 CPU cores with either one KNC (cyan curve) or two KNCs (purple curve). In addition to the portability, note that the results confirm the following observations. Our heterogeneous multi-device implementation achieves perfect scalability for large matrix sizes. In order to evaluate the performance of an algorithm we

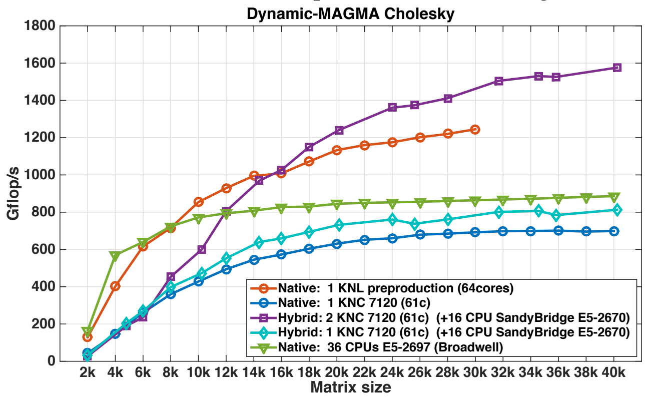

Fig. 6. Performance of Cholesky across five hardware configurations.

rate its performance compared to what we refer to practical peak which is the peak of the most compute-intensive and the most optimized Level 3 BLAS routine, the dgemm routine. The peak performance of the MKL square dgemm on 36 cores Broadwell E5-2697 is about 1,140 Gflop/s, on one KNC is 940 Gflop/s, and around 2,000 Gflop/s for the KNL. The operands of the update phases have rectangular shapes reducing the update's performance to about 90% of the square gemm peak mentioned above. The performance obtained by the hybrid LU factorization illustrated in Figure 4 is about 900 Gflop/s on a single KNC and about 1,700 Gflop/s on two KNCs, which demonstrates scalability and performance close to the peak. This also indicates that the panel factorization phase running on the CPUs is fully overlapped with the trailing matrix update running on the KNCs, and for that, the overall factorization performance reaches the Level 3 BLAS gemm performance. More attractive are the native performance results. We obtained about 772 Gflop/s on the KNC and about 1,500 Gflop/s on the KNL, which is considered efficient and high performance. Note that when running in native mode for any hardware (CPUs, KNC, or KNL), the hardware is split over Master and Devices. The master is assigned to a small number of the hardware cores (in our experiments, about 10%), and the remaining cores are the ones that contribute to the trailing matrix update. As a consequence, in order to evaluate our algorithm, the peak now is the performance of the gemm on the remaining number of cores. The native codes are within 90+% of the hybrid ones, i.e., within 90+% of running just the Level 3 BLAS flops of the factorizations. A similar trend was observed for the QR and Cholesky factorizations.

Figures 7, 8, and 9 show a performance comparison of our results *vs.* MKL for the LU, QR, and Cholesky factorizations, respectively, on the KNL processor. MKL is optimized for all Intel Xeon and Xeon Phi architectures. The Intel MKL team puts additional effort into optimizing the Level 3 BLAS routines and the LAPACK LU, Cholesky, and QR factorizations, as these are some of the most commonly used routines in the library. The MKL LU factorization is the most highly optimized of the three factorizations. The MKL LU performance results look very attractive, reaching up to 1,700 Gflop/s, which is very close to the peak performance of the dgemm routine (*vs.* 1,320 for Cholesky and 800 for QR). In comparison, the dynamic MAGMA LU achieves up to 1,500 Gflop/s. For the QR factorization, as shown in Figure 8, our implementation significantly outperforms the Intel MKL. This is due mostly to the algorithmic and kernel optimizations outlined in the previous sections. The Cholesky factorizations are about the same performance with Intel MKL, slightly outperforming the dynamic MAGMA version.

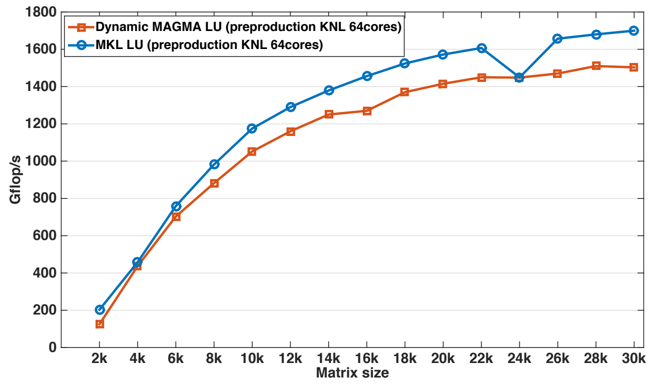

Fig. 7. Dynamic MAGMA *vs.* MKL LU on the preproduction KNL in DP.

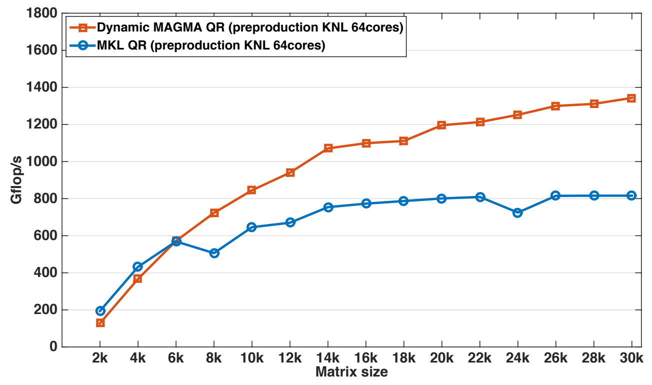

Fig. 8. Dynamic MAGMA *vs.* MKL QR on the preproduction KNL in DP.

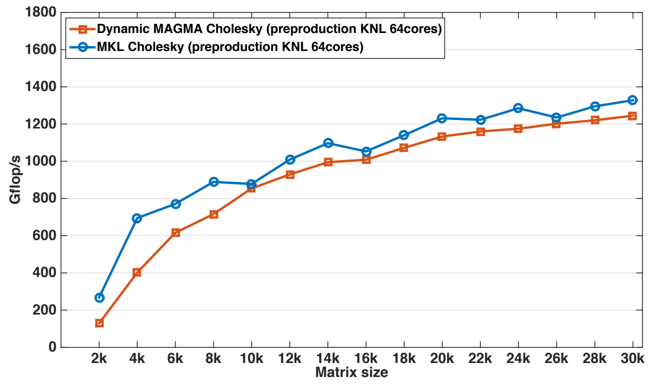

Fig. 9. Dynamic MAGMA *vs.* MKL DP Cholesky on the preproduction KNL

VII. Conclusion and Future Work

While heterogeneous compute nodes have become ubiquitous, the need for sustainable numerical libraries and an easy programming paradigm, capable of delivering correct results and providing portability and efficiency across a large range of hybrid environments, became ever critical.

We designed a programing model and developed a number of optimization techniques for the LU, QR, and Cholesky factorizations on many-core systems. In particular, the techniques presented advanced the current state-of-the-art for these factorizations, providing efficient support for KNL, Intel's next generation of Xeon Phi processors. We also showed how judicious modifications to superscalar task scheduling were used to meet two competing goals: (1) obtain portable high-fraction of the peak performance for heterogeneous systems, and (2) employ a unified programming model that simplifies the development. Our performance analysis unequivocally demonstrates that our approach improves the performance of heterogeneous platforms by using adaptive scheduling techniques and also enhances the scalability of the underlying algorithms by providing a set of features capable of mapping the algorithm and its data to all potential computing resources. This principle can be extended to many other algorithms such as the eigenvalue and singular value methods or even sparse solvers. Future work will include releasing a dynamic MAGMA library that merges CUDA, OpenCL, and Intel Xeon Phi development branches into a single software package using our new programming model.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant ACI-1339822, the Department of Energy, and Intel. The results were obtained in part with the financial support of the Russian Scientific Fund, Agreement N14-11-00190.

References

[1] M. Abalenkovs, A. Abdelfattah, J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, and A. YarKhan. Parallel programming models for dense linear algebra on heterogeneous systems. *Supercomputing Frontiers and Innovations*, 2(4), 10-2015 2015.

[2] 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.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 863–874, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] R. Barik, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar. The Habanero Multicore Software Research Project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 735–736, New York, NY, USA, 2009. ACM.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.

[6] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *Proceedings of the nineteenth annual ACM symposium on parallel algorithms and architectures*, SPAA '07, pages 116–125, New York, NY, USA, 2007. ACM.

[7] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, 1998.

[8] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki. Accelerating numerical dense linear algebra calculations with gpus. *Numerical Computations with GPUs*, pages 1–26, 2014.

[9] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[10] C. H. González and B. B. Fraguela. A framework for argument-based task synchronization with automatic detection of dependencies. *Parallel Computing*, 39(9):475 – 489, 2013. Novel On-Chip Parallel Architectures and Software Support.

[11] A. Haidar, C. Cao, I. Yamazaki, J. Dongarra, M. Gates, P. Luszczek, and S. Tomov. Performance and Portability with OpenCL for Throughput-Oriented HPC Workloads Across Accelerators, Coprocessors, and Multicore Processors. In *5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA 14)*, New Orleans, LA, 11-2014 2014. IEEE.

[12] A. Haidar, C. Cao, A. Yarkhan, P. Luszczek, S. Tomov, K. Kabir, and J. Dongarra. Unified development for mixed multi-gpu and multi-coprocessor environments using a lightweight runtime environment. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 491–500, Washington, DC, USA, 2014. IEEE Computer Society.

[13] A. Haidar, T. Dong, S. Tomov, P. Luszczek, and J. Dongarra. Framework for batched and gpu-resident factorization algorithms to block householder transformations. In *ISC High Performance*, Frankfurt, Germany, 07-2015 2015. Springer, Springer.

[14] A. Haidar, J. Dongarra, K. Kabir, M. Gates, P. Luszczek, S. Tomov, and Y. Jia. Hpc programming on intel many-integrated-core hardware with magma port to xeon phi. *Scientific Programming*, 23, 01-2015 2015.

[15] A. Haidar, Y. Jia, P. Luszczek, S. Tomov, A. YarKhan, and J. Dongarra. Weighted dynamic scheduling with many parallelism grains for offloading of numerical workloads to multiple varied accelerators. In *Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '15, pages 5:1–5:8, New York, NY, USA, 2015. ACM.

[16] A. Haidar, S. Tomov, P. Luszczek, and J. Dongarra. Magma embedded: Towards a dense linear algebra library for energy efficient extreme computing. In *2015 IEEE High Performance Extreme Computing Conference (HPEC 15), (Best Paper Award)*, Waltham, MA, 09-2015 2015. IEEE, IEEE.

[17] A. Haidar, A. YarKhan, C. Cao, P. Luszczek, S. Tomov, and J. Dongarra. Flexible linear algebra development and scheduling with cholesky factorization. In *17th IEEE International Conference on High Performance Computing and Communications*, Newark, NJ, 08-2015 2015.

[18] Intel. Math kernel library. https://software.intel.com/en-us/en-us/intel-mkl/.

[19] J. M. Pérez, R. M. Badia, and J. 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.

[20] 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.

[21] P. E. Strazdins. Lookahead and algorithmic blocking techniques compared for parallel matrix factorization. In *10th International Conference on Parallel and Distributed Computing and Systems, IASTED*, Las Vegas, USA, 1998.

[22] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parellel Comput. Syst. Appl.*, 36(5-6):232–240, 2010. DOI: 10.1016/j.parco.2009.12.005.

[23] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK Users' Guide: QUeueing And Runtime for Kernels. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011.