Heterogenous Acceleration for Linear Algebra in Mulit-Coprocessor Environments*

Azzam Haidar[1], Piotr Luszczek[1], Stanimire Tomov[1], and Jack Dongarra[1,2,3]

[1] University of Tennessee Knoxville, USA
[2] Oak Ridge National Laboratory, USA
[3] University of Manchester, Manchester M13 9PL, UK

Abstract. We present an efficient and scalable programming model for the development of linear algebra in heterogeneous multi-coprocessor environments. The model incorporates some of the current best design and implementation practices for the heterogeneous acceleration of dense linear algebra (DLA). Examples are given as the basis for solving linear systems' algorithms – the LU, QR, and Cholesky factorizations. To generate the extreme level of parallelism needed for the efficient use of coprocessors, algorithms of interest are redesigned and then split into well-chosen computational tasks. The tasks execution is scheduled over the computational components of a hybrid system of multi-core CPUs and coprocessors using a light-weight runtime system. The use of light-weight runtime systems keeps scheduling overhead low, while enabling the expression of parallelism through otherwise sequential code. This simplifies the development efforts and allows the exploration of the unique strengths of the various hardware components.

1 Programming Models for the Off-Load Mode

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, sometimes known under an alternative name of Many Integrated Cores (MIC). The common mode of operation for the device is called off-load but the stand-alone and reverse off-load are also 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 (commonly an Intel CPU such as Sandy Bridge or Ivy Bridge) being involved. 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 compute-capable GPUs and 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.

* This research was partially supported by the National Science Foundation under Grants OCI-1032815, ACI-1339822, and Subcontract RA241-G1 on NSF Prime Grant OCI-0910735, DOE under Grants DE-SC0004983 and DE-SC0010042, and Intel.

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.

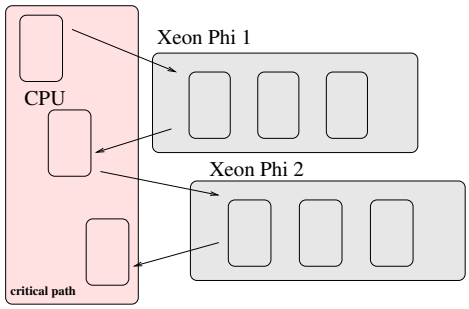

Fig. 1. DLA algorithm as a collection of BLAS-based tasks represented as rectangles and their dependences represented as arrows. The algorithm's critical path is, in general, scheduled on the CPUs, and large data-parallel tasks on the Xeon Phi coprocessors.

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 [3]. 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.

For many scientific applications, the offload model offered by the Intel compiler and OpenMP 4 is sufficient. Until recently, this was not the case for a port of the MAGMA library 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. The motivation to use SCIF is to take advantage of the capability of asynchronous events that allows the user for low-latency messaging between the host and the device as well as to notify about completion of kernels on Xeon Phi. This enabled the possibility of hiding 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 transfers to be page-aligned and pinned to guarantee their fixed location in the physical memory.

With the continuous improvements in the APIs that conceptually reside above SCIF, the overheads and functionality afforded by SCIF is no longer exclusive, and we are able to achieve very much comparable performance and asynchronous interface using higher-level APIs, while gaining portability as an important added bonus.

2 Efficient and Scalable Programming Model Across Multiple Devices

In this section, we describe a programming model that raises the level of abstraction above the hardware specifics while still allowing us to capture the strengths of the various hardare components in a heterogeneous system and develop highly efficient algorithms. We present the accompanying software stack and the techniques developed for the effective use of both single and multi Xeon Phi coprocessors. Our proposed techniques consider both the higher ratio of execution and the hierarchical memory model of the new emerging coprocessors.

2.1 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 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 [10, 9].

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.

2.2 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.

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, ... 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}, ...) ;
```

The use of multiple coprocessors complicates the development using static scheduling. Instead, the use of a light-weight 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 [11] superscalar execution environment that has been originally used with great success for linear algebra software on just multicore platforms [6]. The conceptual work though could be replicated within other models such as StarPU [1], OmpSS [7], Cilk [2], and Jade [8], to just mention a few.

2.3 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 [5] 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%.

2.4 Data Distribution to Minimize Communication

Data distribution formats for multi-device computations can drastically affect the performance. In particular, swapping rows (the dlaswp routine) in LU, or the dlarfb trailing matrix update routine in QR, may require unnecessary data movements in certain data formats. Therefore, to minimize the amount of communication between devices, our implementation uses a $1D$ block cyclic distribution. Indeed, using the well known $2D$ block cyclic distribution among multi-devices will enforce an extra amount of communication between them in order to perform the dlaswp in LU, while using $1D$ block cyclic distribution will not need any of these communication. Another example is the dlarfb routine used in the QR factorization to perform $(I - VT^TV^T)\tilde{A}$. Here V holds the Householder reflectors generated during the panel factorization at step k, and $\tilde{A}$ is the trailing matrix at step k. A $2D$ block cyclic distribution will require a sum between the devices in order to compute $V^T\tilde{A}$, while a $1D$ block cyclic distribution will again not need any communications. Note that the overall workload is well spread among the multi-coprocessors when using $1D$ block cyclic distribution.

2.5 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.

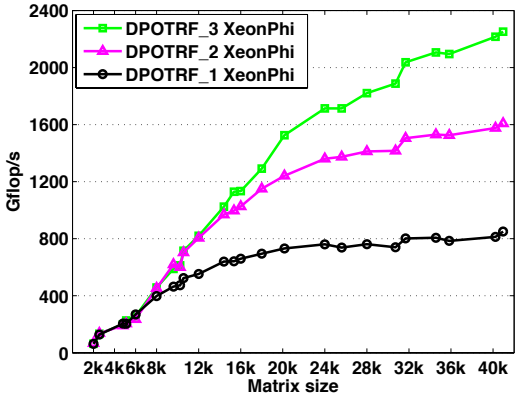

Fig. 2. Performance of DPOTRF (Cholesky factorization) on up to 3 Xeon Phi cards.

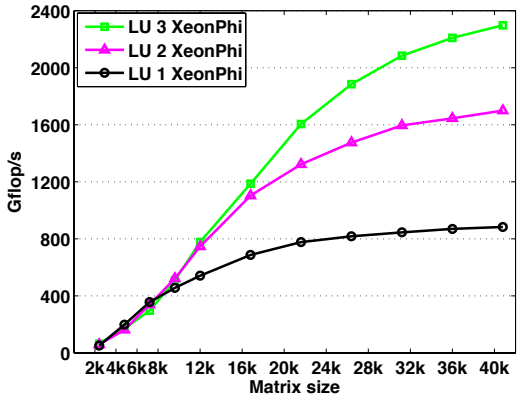

Fig. 3. Performance of DGETRF (LU factorization) on up to 3 Xeon Phi cards.

3 Experimental Results

We present performance results on an Intel dual-socket multicore system with three Intel Xeon Phi cards. Each CPU processor is eight-core Intel Xeon E5-2670 (Sandy Bridge), running at 2.6 GHz, and has a 24 MB shared Level 3 cache. Each core has a private 256 KB Level 2 and 64 KB Level 1 caches. The system is equipped with 52 GB of memory. Its theoretical peak in double precision is 332 Gflop/s. The Intel Xeon Phi cards have 15 GB memory each, running at 1.09 GHz, and yielding a double precision theoretical peak of $1,046$ Gflops.

On the CPU side we use the MKL (Math Kernel Library) [4], version 11.00.03. The Intel Xeon Phi is running the MPSS 2.1.5889-16 software stack and the icc 13.1.1 20130313 compiler. These come with the composer_xe_2013.3.163 suite.

Figures 2, 3, and 4 show the performance results for the Cholesky, LU, and QR factorizations respectively. The figures show a scalability study for up to 3 Xeon Phi devices. The first observation is the large matrix sizes (beyond 5000) required to take advantage of the benefits that the devices offer and, consequently, outperform the peak performance of the CPU. Similarly, adding the second Xeon Phi is beneficial for matrix sizes larger than $10,000$ for Cholesky, $12,000$ for LU, and QR factorizations. Finally,

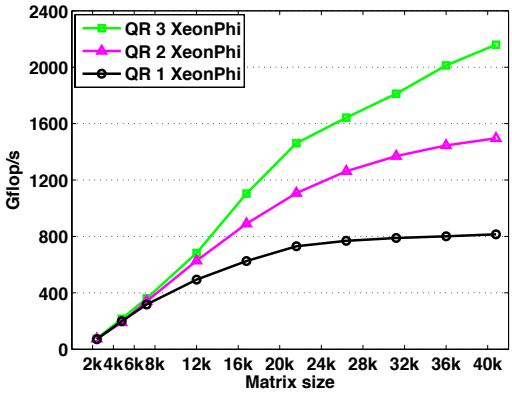

Fig. 4. Performance of DGEQRF (QR factorization) on up to 3 Xeon Phi cards.

the addition of the third Xeon Phi benefits all three factorizations only beyond matrices of size $16,000$. This behavior is to be expected from a compute-oriented device that is connected to the CPU through a high-latency, low-bandwidth bus such as the PCI Express. Each matrix panel is factorized on the CPU and for that must make its way from the Xeon Phi device to the CPU and back, thus suffering the communication penalty twice. While the dynamic scheduling allows us to hide this overhead at the beginning of the factorization when the trailing matrix updates carry enough of a computational load, the final steps are squarely dominated by the panel computation and very little can be done about it since the Xeon Phi is a throughput oriented device and in our attempts delivered low performance for latency-bound workloads such as the panel factorization.

As far as scaling and parallel efficiency are concerned, Figures 2, 3, and 4 show that once the matrix sizes grow beyond the aforementioned threshold, the scaling from one to two and from two to three Xeon Phi devices remains steady and progresses as expected.

Another important aspect of the performance behavior that we observed on our Xeon Phi cards can be seen in Figure 2. The figure shows extra data points to underscore the variability of the performance with respect to the problem size. In particular, when the matrix sizes are not divisible by a particular value, a blocking factor for the underlying BLAS library, the resulting performance might not follow a smooth path and experience variations. However, we are finding this behavior to continuously become less of a burden with every new release of the software stack for the Xeon Phi card.

4 Conclusions and Future Work

We designed algorithms and a programing model for developing high-performance dense linear algebra in co-processors environments. Further, despite the complexity of the hardware, acceleration was achieved at a surprisingly low software development effort using a high-level methodology of developing hybrid algorithms. In particular, we obtained high fraction of the peak performance for the entire heterogeneous system. The promise shown so far motivates and opens opportunities for future research and extensions, e.g., tackling more complex algorithms and hybrid hardware. When a complex algorithm needs to be executed on a complex heterogeneous system, scheduling

decisions have a dramatic impact on performance. Therefore, new scheduling strategies must be designed to fully benefit from the potential of future large-scale machines.

References

1. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.
2. 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.
3. Intel. Intel xeon phi coprocessor system software developers guide. http://software.intel.com/en-us/articles/.
4. Intel. Math Kernel Library. http://software.intel.com/intel-mkl/.
5. J. Jeffers and J. Reinders. *Intel® Xeon Phi™ Coprocessor High-Performance Programming*. Morgan Kaufmann Publishers, 2013.
6. 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.
7. 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.
8. 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.
9. L. G. Valiant. Bulk-synchronous parallel computers. In M. Reeve, editor, *Parallel Processing and Artificial Intelligence*, pages 15–22. John Wiley & Sons, 1989.
10. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), Aug. 1990. DOI 10.1145/79173.79181.
11. A. YarKhan. *Dynamic Task Execution on Shared and Distributed Memory Architectures*. PhD thesis, University of Tennessee, December 2012.