Flexible Linear Algebra Development and Scheduling with Cholesky Factorization

Azzam Haidar
University of Tennessee, Knoxville, USA
haidar@eecs.utk.edu

Asim YarKhan
University of Tennessee, Knoxville, USA
yarkhan@eecs.utk.edu

Chongxiao Cao
University of Tennessee, Knoxville, USA
ccao1@vols.utk.edu

Piotr Luszczek
University of Tennessee, Knoxville, USA
luszczek@eecs.utk.edu

Stanimire Tomov
University of Tennessee, Knoxville, USA
tomov@eecs.utk.edu

Jack Dongarra
University of Tennessee, Knoxville, USA
Oak Ridge National Laboratory, USA
University of Manchester, UK
dongarra@eecs.utk.edu

Abstract

Modern high performance computing environments are composed of networks of compute nodes that often contain a variety of heterogeneous compute resources, such as multicore-CPUs, GPUs, and coprocessors. One challenge faced by domain scientists is how to efficiently use all these distributed, heterogeneous resources. In order to use the GPUs effectively, the workload parallelism needs to be much greater than the parallelism for a multicore-CPU. On the other hand, a Xeon Phi coprocessor will work most effectively with degree of parallelism between GPUs and multicore-CPUs. Additionally, effectively using distributed memory nodes brings out another level of complexity where the workload must be carefully partitioned over the nodes. In this work we are using a lightweight runtime environment to handle many of the complexities in such distributed, heterogeneous systems. The runtime environment uses task-superscalar concepts to enable the developer to write serial code while providing parallel execution. The task-programming model allows the developer to write resource-specialization code, so that each resource gets the appropriate sized workload-grain. Our task-programming abstraction enables the developer to write a single algorithm that will execute efficiently across the distributed heterogeneous machine. We demonstrate the effectiveness of our approach with performance results for dense linear algebra applications, specifically the Cholesky factorization.

1. Introduction

The scientific, technical, and HPC communities are increasingly turning to computers containing a combination of heterogeneous hardware resources. Modern compute platforms augment their multicore CPUs with one-or-more GPUs or coprocessors like the Intel Xeon Phi (MIC - Many Integrated Cores). We believe that the compute nodes of large-scale machines will contain a mixed-core approach to hardware, combining multicores and GPUs or coprocessors, each of which appropriate for various work granularities. These mixed-core nodes will be connected together via high-speed networks to create large powerful distributed memory installations. However, it remains an open question of how to use these mixed-core compute resources efficiently. No current approach is designed to adapt to the available resources in a transparent manner. We examine an approach that builds on the strengths of all the hardware resources, combining them when appropriate, and using them separately when the appropriate task granularity requires it. In order to accomplish this we extend the work described in our earlier paper [17], where we designed a mixed-core approach for a single node.

The extension includes a number of new contributions varying from the way data is stored and moved to the way algorithms are split into tasks and scheduled for execution. Our approach targets the development of high-performance dense linear algebra (DLA), and we demonstrate it for the case of the Cholesky factorization.

2. Background and Related Work

The desire for faster computers and greater immersive experiences has led hardware designers to push beyond the constraints of Moore's Law. Machines are designed with increasing numbers of cores and specialized accelerators that provide enormous performance boosts for the right kind of computation. In response to this, computer scientists are designing algorithms, software and frameworks that can take advantage of these highly parallel resources. Algorithms need to be restructured to allow greater levels of asynchronous parallelism and workloads need to be tailored to match the available hardware, where the hardware can include multicore CPUs, GPUs, and other accelerators or coprocessors.

Dense linear algebra libraries are at the core of a substantial number of scientific codes across science domains. The traditional approach to extracting high linear algebra performance from the hardware resources is by relying on highly tuned, platform-specific optimized libraries, such as the Basic Linear Algebra Subroutines (BLAS). The scientific codes are then constructed as a sequence of calls to an optimized library. However, this can result in a fork-join style execution where highly tuned parallel code is interleaved with code that achieves very low parallelism. Newer alternatives to extracting parallelism from available hardware have been advocating a higher level approach [11]. This approach is a task-based dataflow approach where precedence-constrained tasks are executed asynchronously and in parallel.

Developing linear algebra algorithms for today's heterogeneous machines requires a level of complexity that was not required in homogeneous environments. Each different type of hardware is likely to have a different workload-grain size for optimal performance. For example, for matrix-matrix multiplication, a GPU will require large matrix sizes in order to achieve high performance, whereas a CPU can achieve its best performance at much smaller sizes. Furthermore, in a distributed memory environment, the data needs to be partitioned among the nodes to enable load balance and scalability. If the nodes have varying capabilities, then the algorithm needs to allow different workloads to be allocated to each node.

This paper presents research in designing the algorithms and the programming model for high-performance DLA in distributed-memory heterogeneous environments. The compute nodes in these environments can be composed of a mix of multicore CPUs, GPUs, and MICs, all of which may have varying capabilities and different optimal workload granularity. While the main goal is to obtain as high fraction of the peak performance as possible for the entire system, a competing secondary goal is to propose a programming model that would simplify the development. To this end, we propose and develop a new distributed-memory lightweight runtime environment, and describe the construction of DLA routines based on it. We demonstrate the new heterogeneous runtime environment and its programming model using the Cholesky factorization. The design of this new environment considers our experience [17], as well as other state-of-the-art developments in the area, summarized as follows, to extract and develop the techniques best suited for DLA on distributed heterogeneous systems.

There has been a lot of effort on enabling DLA libraries to run on heterogeneous systems. Vendors such as NVIDIA, Intel, and AMD provide their own numerical libraires, such as cuBLAS/cuSolver/cuSparse [21], MKL [19], and clMath [3], respectively. LAPACK is provided in MAGMA [26] for heterogeneous systems with GPUs (in CUDA and OpenCL) or Xeon Phi coprocessors, and in CULA [18] for Nvidia GPUs. These libraries do not include implementations for distributed-memory systems yet.

Song et al. [23] describe distributed-memory, multi-GPU linear algebra algorithms that use a static multi-level block-cyclic data partitioning. The static data layout allows the distributed nodes to schedule communication events without coordination. The multi-level data scheme enables CPUs and GPUs to partition work to handle the workload imbalance between the resources. This approach does not provide for GPUs of different strengths and for the addition of other resources such as MICs.

Ayguade et al. have created StarSS [7], a programming model that uses compiler directives to annotate code in order to allow task superscaler execution via a specialized runtime. The directives can specify that functions should be executed using specific hardware (e.g. GPU, Cell, SMP) rather than using CPUs. The superscalar execution allows the host CPU

and additional hardware to run in parallel. Some versions of StarSS support distributed execution but the data movement must be explicitly specified by the user [16]. Many of the ideas in StarSS have been incorporated in the implementation of Task Parallelism in the OpenMP 4.0 specification [14], however the OpenMP standard [15] does not include distributed-memory execution.

Augonnet [5] and the INRIA Runtime team have developed StarPU [6], which is a dynamic scheduling runtime that uses superscalar execution methods to run sequential task-based code on parallel resources. StarPU uses a history-based scheduling mechanism to transparently schedule tasks on heterogeneous multicore and GPU resources, with extensions that allow StarPU to execute in distributed-memory environments. StarPU has been used as a runtime in MAGMA to implement the Cholesky, QR, and LU factorizations [2].

The SuperMatrix runtime system for linear algebra [12] was extended to execute on multicore and GPUs in a shared-memory environment [22]. The SuperMatrix approach requires that the task-dependencies be substantially exposed before scheduling and that the GPU take the burden of the computation, not using available multicore CPUs for complex computational tasks.

Bosilca et al. [10] have developed PARSEC, a distributed task-based runtime environment using Parameterized Task Graphs (PTGs). PARSEC has been applied to DLA [9], however it remains challenging to implement complex algorithms using PARSEC.

3. Algorithmic Advancements

We extend the classical LAPACK algorithms [4] into heterogeneous algorithms for distributed systems and give a description for the case of the Cholesky factorization. We designed a two-level block-cyclic distribution method to support the heterogeneous algorithms, as well as an adaptive task scheduling method to determine the splitting of work over the devices.

Algorithm 1 shows the starting point of our algorithmic considerations. The decomposition of the input matrix across both rows and columns is matched by the decomposition in double-nested loop to allow for static mapping to the hardware and flexible scheduling at runtime. This two-fold decomposition in the data domain and the algorithmic domain serves as facility of introducing lookahead [24, 25] to increase efficiency through temporal and spacial overlap of communication, computation, and the mix thereof.

Algorithm 1: Right-looking blocked and tiled Cholesky factorization with a fixed blocking factor n_b.

Input : A—Symmetric positive definite
Input : n_b—Blocking factor
Output : L—Lower triangular
for $A_{i,i} \in \{A_{1,1}, A_{2,2}, A_{3,3}, \ldots A_{*,*}\}$ **do**
  $A_{i,i} \in \mathbf{R}^{n_b \times n_b}$
  $L_{i,i} \leftarrow \texttt{UnblockedCholesky}(A_{i,i})$
  for $A_{j,i} \in \{A_{i+1,i}, A_{i+2,i}, A_{i+3,i}, \ldots A_{*,i}\}$ **do**
    $A_{j,i} \in \mathbf{R}^{n_b \times n_b}$
    $A_{j,i} \leftarrow L_{i,i}^{-1} \times A_{j,i}$
  for $A_{j,k}$ *where* $j,k > i$ **do**
    $A_{j,k} \in \mathbf{R}^{n_b \times n_b}$
    $A_{j,k} \leftarrow A_{j,k} - L_{j,i} \times L_{i,k}$

Through this partitioning, we can take this concept beyond its inception and apply it in both domains (across matrix dimensions and loop nests) simultaneously. The proper tracking of these, admittedly more complex, dependences is offloaded to the runtime and thus only a minor burden is left to the algorithm developer – the custodial task of invoking the runtime and informaing it about the dataflow structure.

3.1. Data Distribution

We use a multi-level hierarchy of data blocking rather than fixed blocking across nodes, cores, and devices. At the coarsest (global distributed) level we employ a 2D block cyclic distribution [13], the main reasons being scalability and load balance, both of which are of concern at the level of parallelism and hardware size that we target. Within a single node, the amount of concurrency can still be staggering, especially when we count CUDA cores, floating-point ALUs, and hyper-threading contexts. More appropriate, however, is modeling the single node hardware unit as a moderately parallel entity with at most tens of computational units, be it GPU cores (NVIDIA's SMX or AMD compute units) or CPU cores (often multi-way hyper-threaded). For such a hardware model, a 1D cyclic distribution is adequate to balance the load while still scaling efficiently. This 1D distribution has some additional benefits for matching the data layout to the panel-update style linear algebra algorithm.

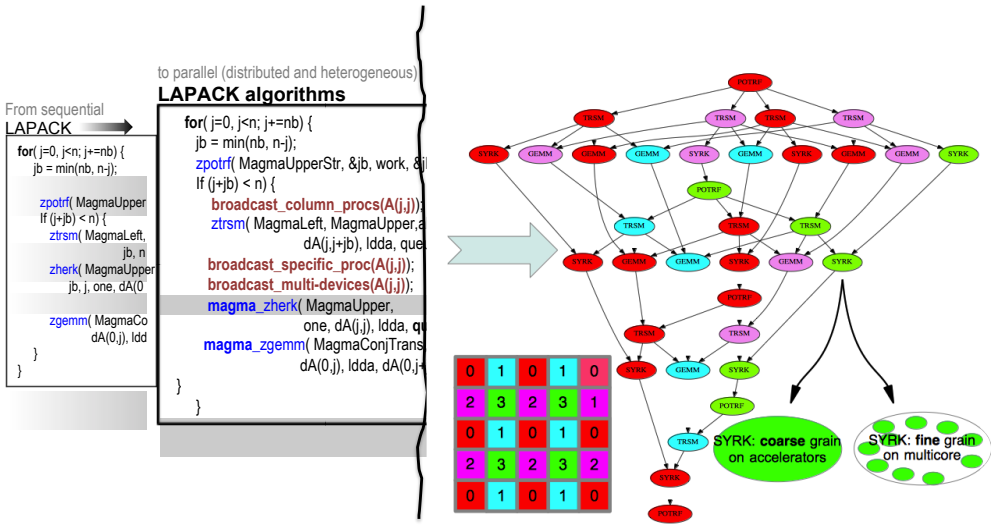

Figure 1. Algorithms look like LAPACK (left), while a task superscalar runtime executes the underlying distributed algorithm (right). The execution can be viewed as a DAG with the tasks executed on nodes where the 2D block-cyclic data is located. A matrix consisting of 5x5 block-cyclic distributed tiles is executed on four distributed nodes, marked by different colors. MPI communication tasks, not shown for simplicity, are between nodes of different colors. One SYRK task is shown having adaptive grain sizes, depending on the hardware that the task is assigned to (CPU, GPU, Phi).

3.2. Two Communication Layers

Our goal is to provide a level of abstraction that delivers portable performance on many kinds of heterogeneous systems. To that end, we propose a new methodology that avoids the all too common issue of the classical distributed programming model – the "*bulk-synchronous*" [28, 27] lock-step execution, which was used by ScaLapack. This model does not cope productively with the heterogeneity of the current processing units (large core-count manycore and heterogeneous systems), and neither can they overlap the communication nor account for the variability in runtime performance behavior.

In a distributed memory environment, explicit data movement tends to be the source of many parallel, and thus hard to find, bugs. To alleviate the issue and to keep the overall ease of use and consistent notion of task-based programming, we propose encapsulating MPI communication calls inside tasks. This turns the message passing primitives into data sources and sinks, which in turn makes it possible to ease the burden of manual tracking of asynchronous calls throughout the code and ensuring proper progress of the communication protocol. Additionally, the runtime provides basic flow control to limit the number of outstanding asynchronous events, which dovetails the issue of how many such non-blocking calls are acceptable for a given MPI implementation – a purely software engineering limitation that could potentially be hard to accommodate if done manually across a number of open source and vendor MPI libraries.

Our runtime system has already been successfully used for multicore-only workloads from dense linear algebra [20] and as such it enables various advanced scheduling techniques but at its core it enables concurrent execution of tasks across available cores. The situation changes only slightly when one of the cores is devoted to only handle MPI-related activities. On occasion, the communication core might go underutilized due to high computation demand and low communication load but in the overall hardware mix with tens of cores per core, this does not pose an appreciative loss in total achieved performance. On the contrary, at the periods of heavy communication, the thread is either busy queueing new asynchronous sends and/or receives or providing progress opportunity to already executing MPI calls. With this scheme we achieve on-demand communication between nodes from the single message passing thread and shared memory concurrency within the node.

3.3. Task Superscalar Runtime System

The increasing complexity of multicore heterogeneous systems has made problem of efficient work assignment ever more difficult. An effort to address this difficulty has led to the increased use of dynamic run-

time environments to manage the resources. These runtime environments can dynamically assign work when hardware resources complete earlier tasks and become available, enabling a simple form of load balancing.

Task superscalar runtime environments further lighten the burden of parallel programming. In addition to managing the hardware resources, task superscalar environments handle the data hazards between tasks, releasing tasks for execution after all the data dependencies are met.

The net effect of this is that we can program a serial sequence of tasks and transparently obtain parallel (and distributed) execution. Each task has to mark its parameters as read or read/write, and the runtime will enforce any implied data hazards (read after write – RAW, write after read – WAR, and write after write – WAW) between tasks and provide a correct parallel execution.

This enables us to express the Cholesky code as a block-structured code as it is implemented in LAPACK and obtain a parallel execution. Using the block-structured algorithm allows us to express algorithms in the same structure as LAPACK, unlike most other task based linear algebra libraries that required a restructured tile based algorithm working on a tiled data layout [1, 8].

For this work we are utilizing the QUARK task superscalar runtime system [29] which provides several advanced features that make our implementation easier. QUARK has been shown to provide good performance and scalability for PLASMA, a task-based linear algebra library for multicore architectures [20]. However, in essence other task-based superscalar runtime systems could provide similar capabilities for the methodology we are presenting.

3.4. Advanced Scheduling Techniques

When scheduling the tasks on a single shared memory node consisting of multicore CPUs, GPUs, and accelerators, we use the runtime system's ability to do priority-based task scheduling. The algorithm provides priority information with the tasks so that the critical path is processed faster and additional parallelism is exposed earlier. For Cholesky, we prioritize the diagonal tasks to expose the outer loop for the next iteration as soon as possible. This form of lookahead attempts to increase available parallelism and tasks, keeping the computational resources busy.

The runtime system provides task-to-thread locking capabilities, so MPI communication tasks are attached to specific threads. Similarly, GPU control tasks can be locked to a specific thread. Other tasks are scheduled to a specific thread (based on criteria such as locality), but threads that have no scheduled work may steal tasks from other threads.

4. Performance Results

4.1. Hardware Description and Setup

We conducted our experiments on two distributed systems, featuring GPUs and MICs, respectively:
System A has 120 nodes connected with Mellanox InfiniBand QDR. Each node has two Intel Xeon hexa-core X5660 CPUs running at 2.8 GHz, and three NVIDIA Fermi M2090 GPUs.
System B has 48 nodes connected by an FDR InfiniBand interconnect providing 56 Gb/s of bi-directional bandwidth. Each node features two 8-core Intel Xeon E5-2670 CPUs (Sandy Bridge), running at 2.6 GHz, and two Intel Xeon Phi 5110P coprocessors with 8 GiB of GDDR memory each.
A number of software packages were used for the experiments. On the CPU side, we used MKL (Math Kernel Library) [19] with the Intel `icc` compiler version 2013.sp1.2.144 and on the GPU accelerators we used CUDA version 6.0.

4.2. Experimental Data and Discussion

Getting high performance across accelerators remains a challenging problem that we address with the algorithmic and programming techniques described in this paper. These efficient strategies are used to exploit parallelism and hide both inter-nodal and intra-nodal communication. We highlighted this through a set of experiments that we performed on our systems. The data is distributed among the nodes of the cluster using the classical $2D$ fashion and within a node the data is adaptively distributed among the available resources as described in Section 3.1. We performed a set of strong and weak scalability experiments. We use weak scalability to evaluate the capability of our algorithm to solve potentially larger problems when more computing resources are available. In a manner commonly accepted for weak scalability, we increase the input size accordingly when we increase the number of CPU cores and GPUs such a way to have fixed amount of work per node.

We evaluated our unified programming model on two distributed memory machines. Figure 2 illustrates the performance of the Cholesky factorization on System A – distributed platform with GPU accelerators. We plotted the best performance obtained by the

state-of-the-art ScaLapack software as implemented by the Intel MKL, and tuned for the best blocking factor n_b across multiple runs. We also plotted the performance obtained by our algorithm when using only the CPUs. This allowed us to compare fairly with the ScaLapack approach. We can see that our implementation is between 15% to 20% faster than its ScaLapack counterpart and we achieved perfect weak scaling – a result we were expecting. The ScaLapack approach follows the classical "*bulk-synchronous*" technique, meaning that, at every phase of the factorization there is a synchronization. Thus, there is a synchronization between the three phases of the Cholesky algorithm. Cholesky algorithm is quite special since only the factorization of the diagonal tile (xpotrf task) is sequential while the xtrsm and xgemm/xsyrk are naturally parallel. Consequently, the bottleneck of the ScaLapack approach compared to our proposed dynamic technique can be summarized by the following observations:

- during the diagonal tile factorization, only one processor is working in ScaLapack while in our technique, when a processor is performing the diagonal factorization of step i, the other processors are still applying updates from step $i-1$.
- ScaLapack cannot hide the overhead of the communication because it issues only blocking message passing calls, while in our approach, the communication is hidden since it is handled by a separate thread and thus when a communication is in progress, the other threads are busy with computational kernels.
- close to the end of the factorization, there is not enough work to keep the processors fully occupied, this is a bottleneck for the ScaLapack approach, while its effect is minimized for the algorithm we proposed because of the multi-dimensional lookahead technique.

However, since there is only one diagonal tile to be factorized per step, and since the elapsed time to factorize it is very small compared to the time required by the update phase, the scalability of the ScaLapack is expected to be acceptable. Thus, a 20% speedup over ScaLapack is considered to be a viable improvement.

Figure 3 shows the performance obtained per node by each implementation. Note that, a perfect scalability means that the performance per node for certain devices, remains stable when we increase the number of nodes (a flat line indicates perfect scaling). The performance achieved by our algorithm on 100 nodes is about 10.2 Tflop/s which translates to about 102 Gflop/s per node as shown in Figure 3, while the

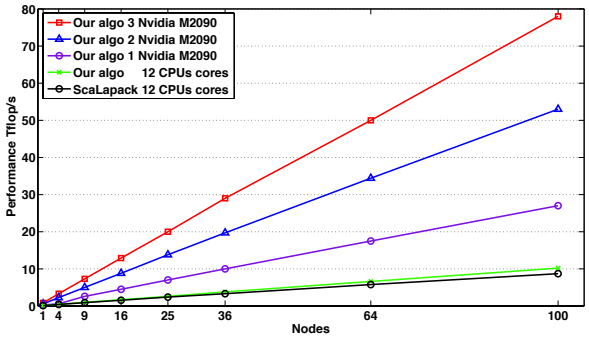

Figure 2. Weak scalability (horizontal reading) strong scalability (vertical reading) of the distributed multi-device Cholesky factorization on System A.

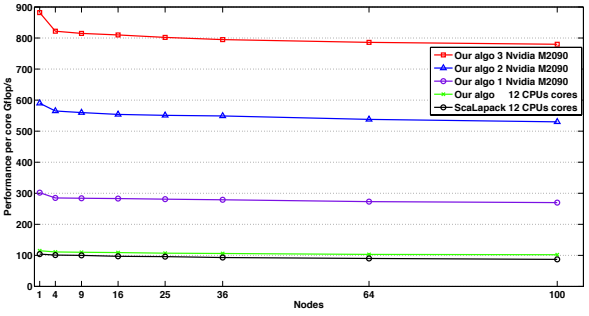

Figure 3. Weak scalability, performance per node on System A.

performance on one node is about 115 Gflop/s meaning that our algorithm can be considered to have provided very good scalability with over 90% efficiency calculated against the performance on a single node multiplied by the number of nodes. The ScaLapack performance per node goes from 104 Gflop/s on one node to about 87 Gflop/s on 64 nodes which is about 73% efficient. The effect of the scalability observed in these results is mostly due to the effect of the communication and the synchronization that does not allow for application of the lookahead technique. The behavior of the CPU performance on System B and depicted in Figure 4 and Figure 5 is similar to the one described for System A. The only difference being that the network available here is not as fast as the one available on System A, which leads to scalability problems for ScaLapack.

The main motivation of our programming model is to have a simple and unified code that can achieve high performance on homogeneous and heterogeneous system alike. To that end, we also conducted experiments on the multi-GPU System A as well as on the multi-Coprocessors System B. Figure 2 depicts the

weak scalability for our algorithm when adding either 1, 2, or 3 GPUs. The experiment demonstrates a good weak scalability when using heterogeneous hardware. Adding one GPU brings up the performance per core (Figure 3) to about 302 Gflop/s on one node and 270 Gflop/s on 100 nodes. The algorithm showed very good scalability even when using a single GPU per node in combination with all of the CPU cores. Enabling more GPUs on each node brought the performance up in a proportionate fashion. The performance obtained on 100 nodes using the 3 NVIDIA GPUs was about 78 Tflop/s for a fixed problem size of 30 000 per node. On 100 nodes when using 3 GPUs, the performance per node is about 780 Gflop/s, as shown in Figure 3. Our implementation showed a very good scalability. Similarly, our experiments on System B illustrates the same behavior. The drop in performance per core when using 2 Xeon Phi is not explicable. We verified using the execution trace that on large number of nodes when using 2 or more Xeon Phi coprocessors, one of them is slowing down on only 1 or 2 nodes – an issue related to overheating and thermal throttling as we later found out. We believe this is related to overheating of these nodes due to malfunctioning ventilation. Previously, we observed such behavior on a single node equipped with 4 Xeon Phi coprocessors, when using all four of devices to solve a single large problem. In the latter case, there was a time period when the Xeon Phi positioned in the middle of the motherboard would slow down due to drastic temperature increase. Overall, our approach exhibit a very good scalability. The performance per node is stable as we increase the number of nodes form 1 to 36.

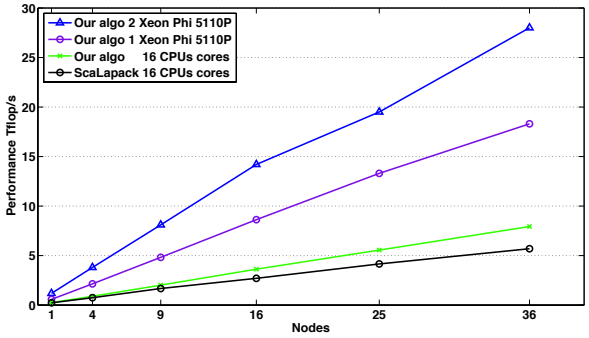

Figure 4. Weak scalability (horizontal reading) strong scalability (vertical reading) of the distributed multi-device Cholesky factorization on System B.

The strong scalability is defined as the speedup that can be achieved to solve a problem of fixed size while increasing the number computing units and

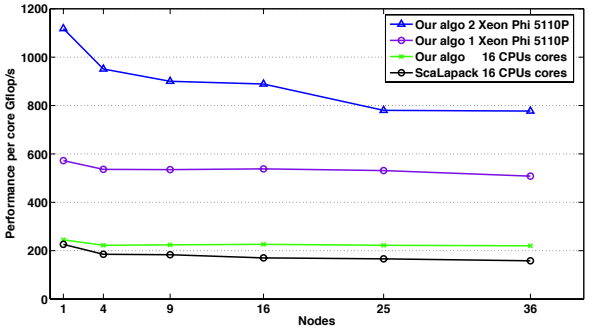

Figure 5. Weak scalability, performance per node on System B.

the associated parallel resources. In our graphs, we can evaluate the strong scalability of our algorithm per node if we read the graph by scanning it along the vertical direction. In other words, given particular nodes configurations, for example 1×1 or 2×2, the vertical reading of the data shows the effect of increasing the computing resources for a fixed-size matrix. Figure 2 and Figure 4 show the strong scaling of the Cholesky factorization on the tested systems when increasing the number of GPUs (Figure 2) or the number of Xeon Phi coprocessors (Figure 4). The obtained strong scaling is nearly ideal. For example. on 1 node, using 3 GPUs is about 3 times faster than running with a single GPU.

5. Conclusions and Future Work

We have designed and implemented a programming model that builds on task-based superscalar runtime environments for implementing dense linear algebra algorithms in distributed memory, multi-way heterogeneous environments. We have presented a methodology for managing the different granularity requirements demanded by the various heterogeneous resources for achieving high performance. Our task superscalar runtime environment allows simple serial algorithmic implementations that are flexible enough to allow high performance execution on our complex distributed, heterogeneous test environments.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grants ACI-1339822 and 1137097, the Department of Energy, and the NVIDIA and Intel Corporations. The results were obtained in part with the financial support of the

Russian Scientific Fund, Agreement N14-11- 00190.

References

[1] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. PLASMA Users Guide. Technical report, ICL, University of Tennessee, 2010.

[2] E. Agullo, J. Dongarra, R. Nath, and S. Tomov. A Fully Empirical Autotuned Dense QR Factorization for Multicore Architectures. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 194–205, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] AMD. clmath libraries: clblas 2.2. https://github.com/clMathLibraries/clBLAS, April 30 2015.

[4] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. SIAM, Philadelphia, PA, USA, 1999.

[5] C. Augonnet. *Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective*. Phd thesis, Universit'e Bordeaux 1, December 2011.

[6] 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.

[7] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An extension of the starss programming model for platforms with multiple gpus. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 851–862. Springer-Verlag, 2009.

[8] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, H. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Distibuted Dense Numerical Linear Algebra Algorithms on Massively Parallel Architectures: DPLASMA. Technical report, Innovative Computing Laboratory, University of Tennessee, apr 2010.

[9] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, H. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Distributed-Memory Task Execution and Dependence Tracking within DAGuE and the DPLASMA Project. Technical Report 232, LAPACK Working Note, Sept. 2010.

[10] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Computing*, 38(1-2):37–51, 2012.

[11] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The Impact of Multicore on Math Software. In B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin / Heidelberg, 2007.

[12] 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.

[13] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. W. Walker, and R. C. Whaley. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. In *Proceedings of the Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science*, PARA '95, pages 107–114, London, UK, UK, 1996. Springer-Verlag.

[14] O. Consortium. OpenMP application programming interface - version 4.0 july 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2013.

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

[16] M. Garcia, J. Corbalan, R. Badia, and J. Labarta. A dynamic load balancing approach with smpsuperscalar and mpi. In R. Keller, D. Kramer, and J.-P. Weiss, editors, *Facing the Multicore - Challenge II*, volume 7174 of *Lecture Notes in Computer Science*, pages 10–23. Springer Berlin / Heidelberg, 2012. 10.1007/978-3-642-30397-5.

[17] 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 *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 491–500, 2014.

[18] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis. Cula: hybrid gpu accelerated linear algebra routines. In *Proc. SPIE*, volume 7705, pages 770502–770502–7, 2010.

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

[20] 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.

[21] Cublas library, 2008. NVIDIA Corporation, Santa Clara, California.

[22] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators.

SIGPLAN Not., 44:121–130, February 2009.

[23] F. Song, S. Tomov, and J. Dongarra. Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 365–376, New York, NY, USA, 2012. ACM.

[24] 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.

[25] P. E. Strazdins. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. *Int. J. Parallel Distrib. Systems Networks*, 4(1):26–35, 2001.

[26] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010.

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

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

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