Dense Linear Algebra Solvers for Multicore with GPU Accelerators

Stanimire Tomov[1], Rajib Nath[1], Hatem Ltaief[1], and Jack Dongarra[1,2,3]
[1]*Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville*
[2]*Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee*
[3]*School of Mathematics & School of Computer Science, University of Manchester*
tomov, rnath1, ltaief, dongarra@eecs.utk.edu

***Abstract*—Solving dense linear systems of equations is a fundamental problem in scientific computing. Numerical simulations involving complex systems represented in terms of unknown variables and relations between them often lead to linear systems of equations that must be solved as fast as possible. We describe current efforts toward the development of these critical solvers in the area of dense linear algebra (DLA) for multicore with GPU accelerators. We describe how to code/develop solvers to effectively use the high computing power available in these new and emerging hybrid architectures. The approach taken is based on *hybridization techniques* in the context of Cholesky, LU, and QR factorizations. We use a high-level parallel programming model and leverage existing software infrastructure, e.g. optimized BLAS for CPU and GPU, and LAPACK for sequential CPU processing. Included also are architecture and algorithm-specific optimizations for standard solvers as well as mixed-precision iterative refinement solvers. The new algorithms, depending on the hardware configuration and routine parameters, can lead to orders of magnitude acceleration when compared to the same algorithms on standard multicore architectures that do not contain GPU accelerators. The newly developed DLA solvers are integrated and freely available through the MAGMA library.**

***Keywords*-Dense Linear Algebra Solvers, GPU Accelerators, Multicore, MAGMA, Hybrid Algorithms.**

I. Introduction

Since the introduction of multicore architectures, hardware designs are going through a renaissance due to the need for new approaches to manage the exponentially increasing:

1) Appetite for power, and
2) Huge gap between compute and communication speeds.

Hybrid GPU-based multicore platforms, composed of both homogeneous multicores and GPUs, stand out among a confluence of current hardware trends as they provide an effective solution to these two challenges. Indeed, as power consumption is typically proportional to the cube of the frequency, GPUs have a clear advantage against current homogeneous multicores, as GPUs' compute power is derived from many cores that are of low frequency. Furthermore, initial GPU experiences across academia, industry, and national research laboratories have provided a long list of success stories for specific applications and algorithms, often reporting speedups of order 10 to 100× compared to current x86-based homogeneous multicore systems [1], [2].

A. Dense Linear Algebra – Enabling New Architectures

Despite the current success stories involving hybrid GPU-based systems, the large scale enabling of those architectures for computational science would still depend on the successful integration and deployment of fundamental numerical libraries. Major issues in terms of developing new algorithms, programmability, reliability, and user productivity must be addressed. This paper describes some of the current efforts on the development of these fundamental libraries, and in particular, libraries in the area of dense linear algebra (DLA).

Historically, DLA has been in the vanguard of efforts to enable new architectures for computational science for good strategic reasons. First, a very wide range of science and engineering applications depend on linear algebra; these applications will not perform well unless DLA libraries perform well. Second, dense linear algebra has a rich and well understood structure for software developers, so these libraries represent a critical starting point for the effort to bridge the yawning software gap that has opened up today within the HPC community.

B. MAGMA – DLA Libraries for Hybrid Architectures

The Matrix Algebra on GPU and Multicore Architectures (MAGMA) project as well as the libraries [3] stemming from it, are used to demonstrate the algorithmic techniques and their effect on performance and portability across hardware systems. Designed to be similar to LAPACK in functionality, data storage, and interface, the MAGMA libraries will allow scientists to effortlessly port their LAPACK-relying software components and to take advantage of each component of the new hybrid architectures. Current work targets GPU-based systems, and the efforts are supported by both government and private industry, including NVIDIA, who recently recognized the University of Tennessee, Knoxville's (UTKs) Innovative Computing Laboratory (ICL) as a CUDA Center of Excellence. This is to further promote, expand, and support ICL's commitment toward developing **DLA Libraries for Hybrid Architectures**.

Against this background, the main focus of this paper will be to provide some high-level insight on **how to code/develop DLA for multicore with GPU accelerators**. The approach described here is based on the idea that in order to deal with the complex challenges stemming

from the heterogeneity of the current GPU-based systems, optimal software solutions will themselves have to hybridize, combining the strengths of the system's hybrid components. In other words, *hybrid algorithms* that match algorithmic requirements to the architectural strengths of the system's hybrid components must be developed. It has been shown that properly designed numerical algorithms for hybrid GPU-based multicore platforms lead to orders of magnitude acceleration.

The paper is organized as follows. Section II describes our approach to make the standard one-sided factorizations, i.e., Cholesky, QR and LU, efficiently run on systems of multicores with GPU accelerators. Section III presents DLA solvers based on correspondingly each of these three factorizations and using efficient triangular solvers as well as mixed-precision iterative refinement techniques. Section IV illustrates the performance results of the different solvers and Section V concludes this paper.

II. Hybrid DLA Algorithms

The development of high performance DLA algorithms for homogeneous multicores has been successful in some cases, like the one-sided factorizations [4], and difficult for others, like the two-sided factorizations [5]. The situation is similar for GPUs - some algorithms map well, others are more challenging. Developing algorithms for a combination of these two architectures (to use both multicore and GPUs) can be potentially beneficial and should be exploited, especially since in many situations, the computational bottlenecks for one of the components (of this hybrid system) may not be the case for the other. Thus, developing **hybrid algorithms** that properly split and schedule the computation over different hardware components may lead to very efficient algorithms. The goal is to develop these new hybrid DLA algorithms that

- Leverage prior DLA developments, and
- Overcome bottlenecks that would not be possible otherwise by just using one of the hybrid component, i.e., homogeneous multicores or GPU accelerators.

A. How to Code DLA for GPUs?

The question of how to code for any architecture, including GPUs, is complex in the sense that issues such as choosing a language, programming model, developing new kernels, programmability, reliability, and user productivity are involved. Nevertheless, it is possible to identify a solid roadmap that has already shown promising results:

1) **Use CUDA / OpenCL:** CUDA is currently the language of choice for programming GPUs. It facilitates a data-based parallel programming model that has turned out to be a remarkable fit for many applications. Moreover, current results show its programming model allows applications to scale on many cores [1]. DLA is no exception as algorithms can be represented in terms of Level 2 and 3 BLAS – essentially a data parallel set of operations that are scaling on current GPUs. The approach described here also shows how the BLAS scalability is in fact translated into scalability on higher level routines (LAPACK). Similar to CUDA, OpenCL takes its roots in the data-based parallelism (now both moving to support task-based parallelism). OpenCL is still yet to be established, but the fact that it is based on a programming model with already recognized potential and the idea of providing portability – across heterogeneous platforms consisting of CPUs, GPUs, and other processors – makes it an excellent candidate for coding hybrid algorithms.
2) **Use GPU BLAS:** Performance of DLA critically depends on the availability of fast BLAS, especially on the most compute intensive kernel, i.e., the Level 3 BLAS matrix-matrix multiplication. Older generation GPUs did not have memory hierarchy and their performance exclusively relied on high bandwidth. Therefore, although there has been some work in the field, the use of older GPUs has not led to significantly accelerated DLA algorithms. For example, Fatahalian et al. studied SGEMM and their conclusion was that CPU implementations outperform most GPU implementations. Similar results were produced by Galoppo et al. on LU factorization. However, the introduction of **memory hierarchy** in current GPUs has drastically changed the situation. Indeed, by having memory hierarchy, GPUs can be programmed for memory reuse and hence not rely exclusively on their high bandwidth. An illustration of this fact is given in Figure 1, showing the performance of a compute-bound (matrix-matrix multiplication on the top) and a memory-bound kernel (matrix-vector multiplication on the bottom). Implementing fast BLAS is a paramount key because algorithms for GPUs can now leverage prior DLA developments, which have traditionally relied on fast BLAS. Of course there are GPU specific optimizations, like trading extra-operations for performance, or interleaving BLAS calls, etc, but the important fact is, high performance algorithms can be coded at a high level, just using BLAS, often abstracting the developer from the need of low-level GPU specific codes.
3) **Use Hybrid Algorithms:** Current GPUs feature massive parallelism but serial kernel execution. For example, the NVIDIA's GTX280 has 30 multiprocessors, each multiprocessor having eight SIMD functional units, each unit capable of executing up to three (single floating point) operations per cycle. At the same time, kernels are executed serially; only one kernel is allowed to run at a time using the entire GPU. This means that only large, highly parallelizable kernels can run efficiently on GPUs. The idea of using hybrid algorithms presents an opportunity to

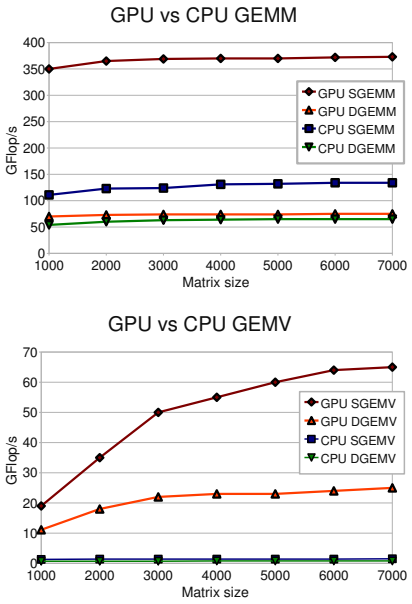

Figure 1. BLAS on GPU (GTX 280) *vs* CPU (8× Intel Xeon 2.33GHz).

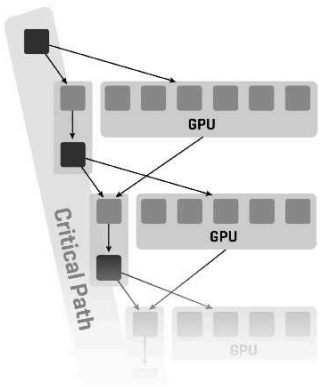

Figure 2. Algorithms as a collection of BLAS-based tasks and dependencies among them (DAGs) for hybrid GPU-based computing

remedy this situation and therefore enable the efficient use of GPUs well beyond the case of data-parallel applications. Namely, the solution and advice to developers is to use a hybrid coding approach, where small, non-parallelizable kernels would be executed on the CPU, and only large, data-parallel kernels on the GPU. Although GPUs move towards supporting task-based parallelism as well (e.g., advertised for the next generation NVIDIA GPUs, code named "Fermi" [6]), small tasks that arise in DLA would still make sense to be executed on the CPU, reusing existing software infrastructure (in particular LAPACK).

B. The Approach – Hybridization of DLA Algorithms

The above considerations are incorporated in the following *Hybridization of DLA Algorithms* approach:

- Represent DLA algorithms as a collection of BLAS-based tasks and dependencies among them (see the illustration in Figure 2):
  - Use parametrized task granularity to facilitate auto-tuning frameworks;
  - Use performance models to facilitate the task splitting/mapping.
- Schedule the execution of the BLAS-based tasks over the multicore and the GPU:
  - Schedule small, non-parallelizable task on the CPU and large, parallelizable on the GPU;
  - Define the algorithm's *critical path* and prioritize its execution/scheduling.

The splitting of the algorithms into tasks is in general easy, as it is based on the splitting of large BLAS into smaller ones. More challenging is choosing the granularity and shape of the splitting and the subsequent scheduling of the sub-tasks. There are two main guiding directions on how to design the splitting and scheduling of tasks. **First**, the splitting and scheduling should allow for asynchronous execution and load balance among the hybrid components. **Second**, it should harness the strengths of the components of a hybrid architecture by properly matching them to algorithmic/task requirements. Examples demonstrating these general directions are given in the next two sections.

Next, choosing the task granularity, can be done by parametrizing the tasks' sizes in the implementations and tuning them empirically [7]. The process can be automated, often referred to as **auto-tuning**. Auto-tuning is crucial for the performance and the maintenance of modern numerical libraries, especially for algorithms designed for hybrid architectures. Figuratively speaking, it can be regarded as both *the Beauty and the Beast* behind hybrid DLA libraries (e.g., MAGMA) as it is an elegant and very practical solution for easy maintenance and performance portability, while often being a brute force, empirically-based exhaustive search that would find and set automatically the best performing algorithms/kernels for a specific hardware configuration. The "exhaustive" search is often relaxed by applying various performance models.

Finally, the problem of scheduling is of crucial importance for the efficient execution of an algorithm. In general, the execution of the critical path of an algorithm should be scheduled as soon as possible. This often remedies the problem of synchronizations introduced by small, non-parallelizable tasks (often on the critical path; scheduled on the CPU) by overlapping their execution with the execution of larger more parallelizable ones (often Level 3 BLAS; scheduled on the GPU).

These principles are general enough to be applied in areas well beyond DLA. Usually they come with specifics, induced by the architecture and the algorithms considered. The following two sections present some of these specifics for the LU, QR, and Cholesky factorizations, and the direct as well as mixed-precision iterative refinement solvers based on them.

C. One-sided Factorizations

This section describes the hybridization of LAPACK's one-sided factorizations – LU, QR, and Cholesky – on dense matrices. LAPACK uses block-partitioned algorithms, and the corresponding hybrid algorithms are based on them. The one-sided factorizations are the first of two steps in solving a dense linear system of equations. It represents the bulk of the computation ($O(N^3)$ floating point operations in the first step *vs* $O(N^2)$ in the second step) and therefore has to be highly optimized. The second step involves triangular solvers (or multiplication with orthogonal matrices, e.g., in the least squares solvers based on the QR/LQ factorizations) and is described in Section III-A.

The opportunity for acceleration using hybrid approaches (CPU and GPU) has been noticed before in the context of one-sided factorizations. In particular, while developing algorithms for GPUs, several groups observed that panel factorizations are often faster on the CPU than on the GPU, which led to the development of highly efficient, one-sided hybrid factorizations for a single CPU core and a GPU [8], [9], multiple GPUs [9], [10], and multicore+GPU systems [11]. M. Fatica [12] developed hybrid DGEMM and DTRSM for GPU-enhanced clusters and used them to accelerate the Linpack benchmark. This approach, mostly based on BLAS level parallelism, results only in minor or no modifications to the original source code.

The performance results showed in this section have all been performed using the NVIDIAs GeForce GTX 280 GPU and its multicore host, a dual socket quad-core Intel Xeon running at 2.33 GHz.

Cholesky Factorization MAGMA uses the left-looking version of the Cholesky factorization. Figure 3 shows how the standard Cholesky algorithm in MATLAB style can be written in LAPACK style and can easily be translated to hybrid implementation. Indeed, note the simplicity and the similarity of the hybrid code with the LAPACK code. The only difference is the two CUDA calls needed to offload data back and forth from the CPU to the GPU. Also, note that steps (2) and (3) are independent and can be overlapped – (2) is scheduled on the CPU and (3) on the GPU, yet another illustration of the general guidelines mentioned in the previous two sections. The performance of this algorithm is given on Figure 4. The hybrid MAGMA Cholesky factorization runs asymptotically at 300 Gflop/s in single and almost 70 Gflop/s in double precision arithmetic.

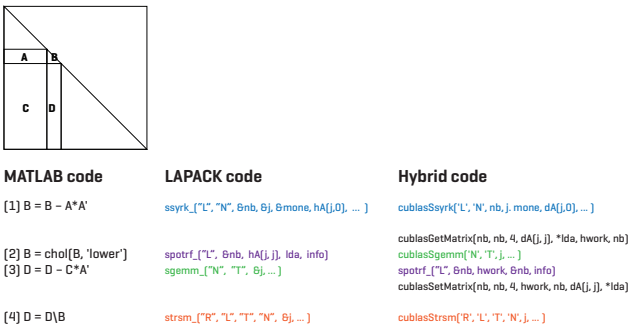

Figure 3. Pseudo-code implementation of the hybrid Cholesky. `hA` and `dA` are pointer to the matrix to be factored correspondingly on the host (CPU) and the device (GPU).

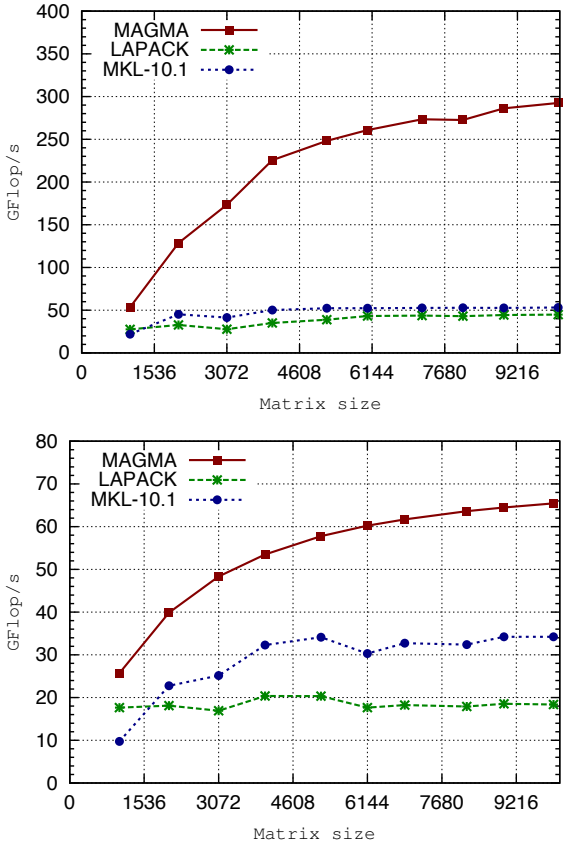

Figure 4. Performance of MAGMA's hybrid Cholesky in single (top) and double precision (bottom) on GTX 280 *vs* MKL 10.1 and LAPACK (with multi-threaded BLAS) on Intel Xeon dual socket quad-core 2.33GHz.

QR Factorization Currently, we use static scheduling and a right looking version of the block QR factorization. The panel factorizations are scheduled on the CPU using calls to LAPACK, and the Level 3 BLAS updates on the trailing sub-matrices are scheduled on the GPU. The trailing matrix updates are split into two parts - one that updates just the next panel and a second one updating the rest. The next

panel update is done first, sent to the CPU, and the panel factorization on the CPU is overlapped with the second part of the trailing matrix. This technique is known as *look-ahead*, e.g., used before in the Linpack benchmark. The performance of this algorithm is given on Figure 5. The hybrid MAGMA QR factorization runs asymptotically almost at 290 Gflop/s in single and almost 68 Gflop/s in double precision arithmetic.

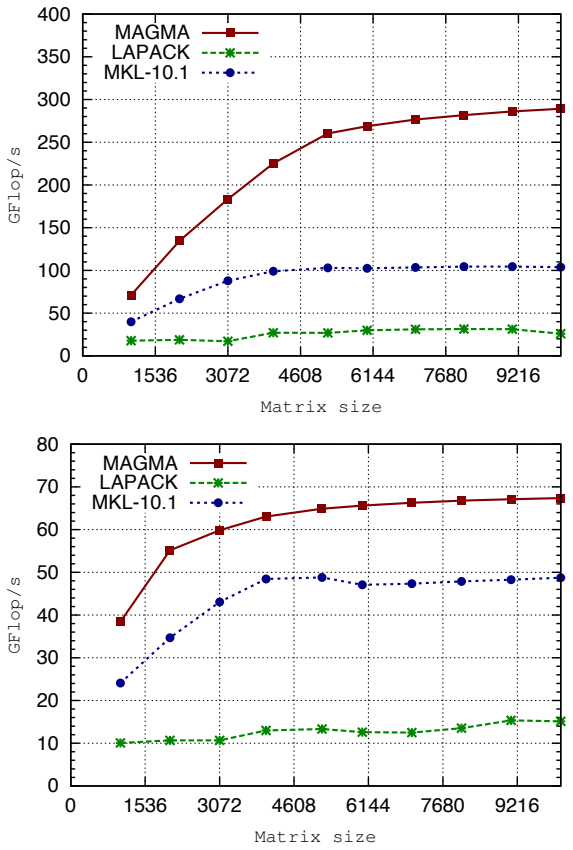

Figure 5. Performance of MAGMA's hybrid QR in single (top) and double precision (bottom) arithmetic on GTX 280 *vs* MKL 10.1 and LAPACK (with multi-threaded BLAS) on Intel Xeon dual socket quad-core 2.33GHz

LU Factorization Similarly to QR, MAGMA uses a right-looking version of the LU factorization. The scheduling is static using the look-ahead technique. Interchanging rows of a matrix stored in column major format, needed in the pivoting process, can not be done efficiently on current GPUs. We use the LU factorization algorithm by V. Volkov and J. Demmel [9] that removes the above mentioned bottleneck. The idea behind it is to transpose the matrix in the GPU memory (once at the beginning of the factorization) so that row elements are contiguous in memory, i.e. equivalent to changing the storage format to row major. Row interchanges now can be done efficiently using coalescent memory accesses on the GPU (*vs* strided memory accesses for a matrix in column major format). The panels are being transposed before being sent to the CPU for factorization, i.e., moved back to the standard for LAPACK column major format. Compared to the non-transposed version, this algorithm runs approximately 50% faster on current NVIDIA GPUs, e.g., GTX 280. The performance of the LU factorization in MAGMA is shown in Figure 6. The hybrid MAGMA LU factorization runs asymptotically almost at 320 Gflop/s in single and almost 70 Gflop/s in double precision arithmetic.

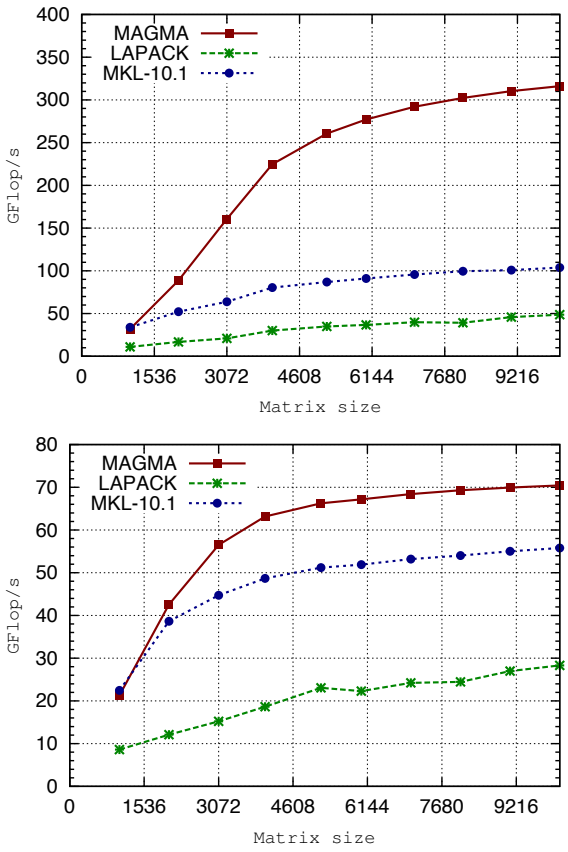

Figure 6. Performance of MAGMA's hybrid LU in single (top) and double precision (bottom) arithmetic on GTX 280 *vs* MKL 10.1 and LAPACK (with multi-threaded BLAS) on Intel Xeon dual socket quad-core 2.33GHz

D. Extension to multiple GPUs

As mentioned earlier, the challenges in developing scalable high performance algorithms for multicore with GPU accelerators systems stem from their heterogeneity, massive parallelism, and the huge gap between the GPUs' compute power *vs* the CPU-GPU communication speed. We show an approach that is largely based on software infrastructures that have already been developed – namely, the *Parallel Linear Algebra for Scalable Multicore Architectures* (PLASMA) [13] and MAGMA libraries. On one hand, the tile algorithm concepts from PLASMA allow the computation to be split into tiles along with a static scheduling mechanism to efficiently balance the work-load between

GPUs. On the other hand, MAGMA kernels are used to efficiently handle heterogeneity and parallelism on a single tile. Thus, the new algorithm features two levels of nested parallelism. A coarse-grained parallelism is provided by splitting the computation into tiles for concurrent execution between GPUs (following PLASMA's framework). A fine-grained parallelism is further provided by splitting the workload within a tile for high efficiency computing on GPUs but also, in certain cases, to benefit from hybrid computations by using both GPUs and CPUs (following MAGMA's framework). Furthermore, to address the challenges related to the huge gap between the GPUs' compute power *vs* the CPU-GPU communication speed, we developed a mechanism to minimize the communications overhead by trading off the amount of memory allocated on GPUs. This is crucial for obtaining high performance and scalability on multicore with GPU accelerators systems.

The experiments shown in Figure 7 have been performed on a dual-socket dual-core host machine based on an AMD Opteron processor operating at 1.8 GHz. The NVIDIA S1070 graphical card is connected to the host via PCI Express 16x adapter cards (3.1 GB/s of CPU-to-GPU and 2.8 GB/s GPU-to-CPU bandwidth). It is composed of four GPUs C1060 with two PCI Express connectors driving two GPUs each. Each GPU has 1.5 GB GDDR-3 of memory and 30 processing cores each, operating at 1.44 GHz. As a result, by reusing the core concepts of our existing software infrastructures along with data persistence optimizations, the new hybrid Cholesky factorization not only achieves unprecedented high performance but also, scales while the number of GPUs increases. The performance reaches up to 1.163 TFlop/s in single and up to 275 GFlop/s in double precision arithmetic. Compared with the performance of the embarrassingly parallel xGEMM over four GPUs, where no communication between GPUs are involved, our algorithm still runs at 73% and 84% for single and double precision arithmetic respectively.

As shown in [14], the static scheduler is very efficient to handle the distribution of tasks on multicore architectures. It still conveniently performs in hybrid environments as presented in Figure 8 with four GPUs. Each task row corresponds to a particular GPU trace execution. The different kernels are clearly identified by their colors. There are almost no gaps between the scheduling of the four different kernels. There is a slight load imbalance phenomenon at the end of the trace mainly because GPUs naturally run out of work as they approach the end of the factorization.

III. Dense Linear System Solvers

We have implemented in MAGMA solvers in real arithmetic, both single and double precision, based on the LU, QR, and Cholesky factorizations. To solve

$$Ax = b$$

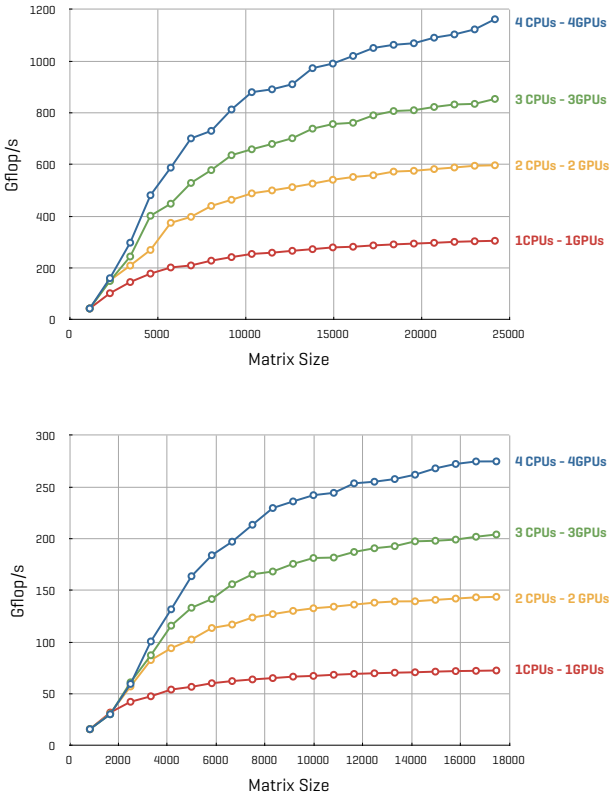

Figure 7. Speed up of the tile hybrid Cholesky factorization in single (top) and double precision (bottom) arithmetic on four NVIDIA C1060 GPUs.

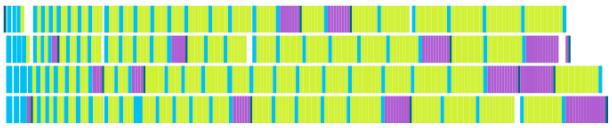

Figure 8. Execution trace of the hybrid tile Cholesky on four GPUs. Dark blue denotes POTRF, light blue SYRK, purple TRSM, and green GEMM.

the matrix A is first factored, and second, the resulting factors are used in solving the original problem. A general recommendation is to use LU for general $n \times n$ matrices, Cholesky for symmetric and positive definite $n \times n$ matrices, and QR for solving least squares problems

$$min\ ||Ax - b||$$

for general $m \times n$, $m \geq n$ matrices. The first step was already discussed in the previous section. Here we comment on the second step, namely the triangular solvers (next).

In addition to having the standard solvers, where both the factorization and the subsequent triangular solves are done in the working precision, we have implemented mixed precision solvers, where a factorization is done in single precision, followed by iterative steps in double precision to increase the accuracy (see Section III-B).

A. Triangular Solvers

Although the solution step has $O(n)\times$ less floating point operations than the factorization, it is still very important to optimize the triangular solver step. Indeed, solving a triangular system of equations can be very slow.Various approaches have been proposed in the past. We use an approach where diagonal blocks of A are explicitly inverted and used in a block algorithm. This results in a numerically stable algorithm, especially when used with triangular matrices coming from numerically stable factorization algorithms (e.g. as in LAPACK and as implemented here in MAGMA), of high performance, e.g., often exceeding $50\times$ the performance of the corresponding CUBLAS implementations (depending on matrix size, number of right-hand-sides and hardware).

B. Mixed-precision Iterative Refinement Solvers

To take advantage of the fact that GPU's single precision is currently of much higher performance than the double precision (theoretically $\approx 10\times$), MAGMA version 0.2 provides a second set of solvers, based on the mixed precision iterative refinement technique. The solvers are based again on correspondingly the LU, QR, and Cholesky factorizations, and are designed to solve linear problems in double precision accuracy but at a speed that is characteristic for the much faster single precision computations. The idea is to use single precision for the bulk of the computation, namely the factorization step, and then use that factorization as a preconditioner in a simple iterative refinement process in double precision arithmetic. This often results in the desired high performance and high accuracy solvers (the limiting factor is the conditioning of the linear system).

IV. Performance Results

This section shows the performance of the linear solvers developed using single and multiple GPUs (up to four). The number of right hand sides has been set to one.The characteristics of the single GPU are described in II-C and the ones for the multiGPU card are mentioned in II-D.

Figure 9 presents the performance of the triangular solvers in single and double precision arithmetic. The speed up is considerable compared to the CUBLAS library.

The performance of the standard linear solvers (factorization and triangular solves in working precision) and the mixed precision iterative refinement solvers is presented in Figure 10. The performance is for a single GPU and is given for the solvers based on each of the three one-sided factorizations, i.e., Cholesky, QR and LU. The experiment is for randomly generated matrices (with adjustment for the symmetric case to be positive definite). The iterative refinement solutions have double precision norm-wise backward error. The results illustrate the high benefit to be expected from these type of solvers, namely, to get a solution in double precision accuracy while running close to the single precision execution rate.

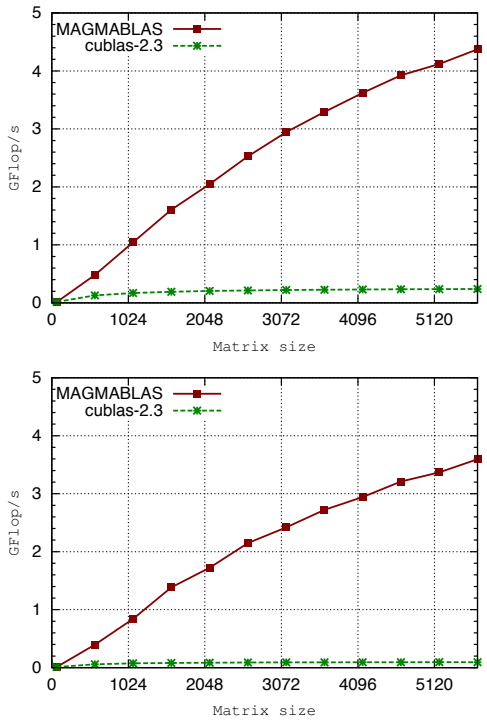

Figure 9. Triangular solvers with one right-hand side in single (top) and double precision (bottom) arithmetic on one NVIDIA GTX 280 GPU.

Figure 11 gives the performance and strong scalability of a mixed precision iterative refinement solver using up to four CPU-GPU couples. The solver is for symmetric and positive definite matrices, using the multiGPU Cholesky factorization from Section II-D. The factorization is done across up to four GPUs and then, the iterations to converge to the solution with enhanced accuracy is performed on a single GPU. We note that the performance curves of the Cholesky solver are close to the Cholesky factorization in single precision, and fairly scales while the number of GPUs increases. The performance with four CPU cores with four GPUs is up to $100\times$ higher than the performance with just the four CPU cores of the host (dual-socket dual-core AMD Opteron 1.8 GHz, solving in double precision arithmetic at 9 GFlop/s).

V. Conclusion

We described a set of techniques on how to develop algorithms that efficiently use hybrid systems of multicores with GPU accelerators to solve dense linear systems. The techniques described are incorporated into the MAGMA library. MAGMA is designed to be similar to LAPACK in functionality, data storage, and interface, to allow scientists to effortlessly port their LAPACK-relying software components and to take advantage of each component of the new hybrid architectures. Current results show the approach is scalable. We used a high-level parallel programming model and leveraged prior advances in the field to develop a

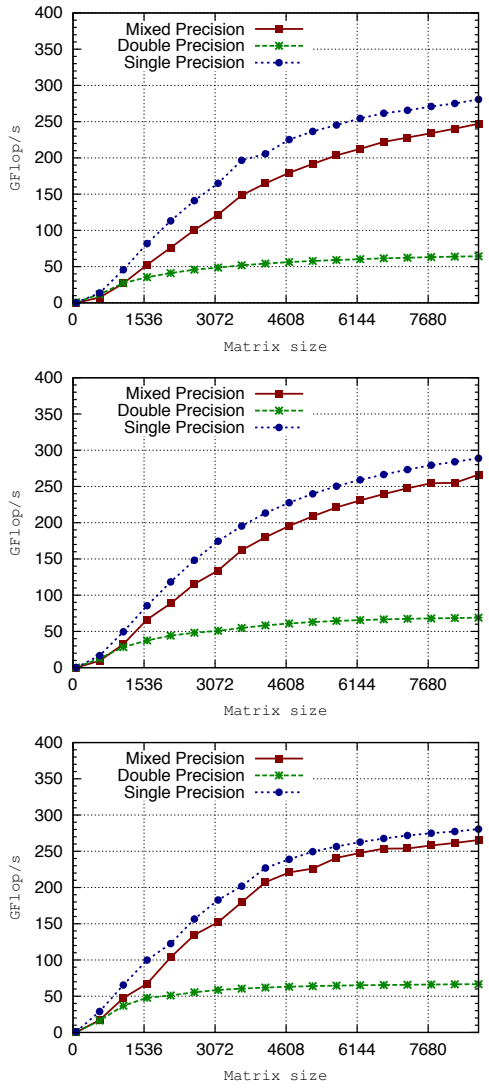

Figure 10. Performance of solvers using Cholesky (top), LU (middle), and QR (bottom) on one NVIDIA GTX 280 GPU.

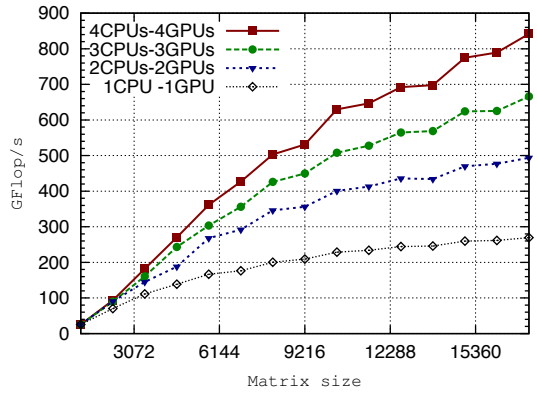

Figure 11. Performance of **double precision solvers** using the **mixed precision iterative refinement** technique on **multiple GPUs**. Shown is the performance of solvers using single precision Cholesky factorization and two double precision iterative refinement steps on up to four NVIDIA C1060 1.44GHz GPUs and four Opteron 1.8GHz CPUs.

scientific computing software tool enabling the efficient use of hybrid GPU accelerated multicore systems.

ACKNOWLEDGMENT

The authors would like to thank the National Science Foundation, Microsoft Research, and NVIDIA for supporting this research effort.

REFERENCES

[1] NVIDIA CUDA ZONE. http://www.nvidia.com/object/cuda_home.html.

[2] General-purpose computation using graphics hardware. http://www.gpgpu.org.

[3] S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 User Guide. http://icl.cs.utk.edu/magma, 11/2009.

[4] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[5] H. Ltaief, J. Kurzak, and J. Dongarra. Parallel band two-sided matrix bidiagonalization for multicore architectures. *Accepted for publication at TPDS*, 2009.

[6] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/object/fermi_architecture.html, 2009.

[7] Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In *ICCS '09*, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] M. Baboulin, J. Dongarra, and S. Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. Lapack working note 200, May 2008.

[9] V. Volkov and J. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[10] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra. A scalable high performant Cholesky factorization for multicore with GPU accelerators. Lawn 223, November 2009.

[11] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. Lawn 210, October 2008.

[12] M. Fatica. Accelerating Linpack with CUDA on heterogenous clusters. In *GPGPU-2*, pages 46–51, New York, NY, USA, 2009. ACM.

[13] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. PLASMA version 2.0 user guide. http://icl.cs.utk.edu/plasma, 2009.

[14] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC '09*, 2009.