Hybrid Multicore Cholesky Factorization with Multiple GPU Accelerators

Hatem Ltaief[1], Stanimire Tomov[1], Rajib Nath[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
{ltaief, tomov, rnath1, dongarra}@eecs.utk.edu

Abstract. We present a Cholesky factorization for multicore with GPU accelerators. The challenges in developing scalable high performance algorithms for these emerging 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 for homogeneous multicores and hybrid GPU-based computing. The algorithm features two levels of nested parallelism. A coarse-grained parallelism is provided by splitting the computation into tiles for concurrent execution between GPUs. A fine-grained parallelism is further provided by splitting the work-load 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. Our resulting computational kernels are highly optimized. An efficient task scheduling mechanism ensures a load balanced execution over the entire multicore with GPU accelerators system. Furthermore, the communication overhead is minimized by trading off the amount of memory allocated on GPUs. This results in a scalable hybrid Cholesky factorization of unprecedented performance. In particular, using NVIDIA's Tesla S1070 (4 C1060 GPUs, each with 30 cores @1.44 GHz) connected to a quad-socket quad-core AMD Opteron @ 2.4 GHz processors, we reach up to 1.189 TFlop/s in single and up to 282 GFlop/s in double precision arithmetic. Compared with the performance of the parallel xGEMM over four GPUs, our algorithm still runs at 78% and 89% for single and double precision arithmetic respectively.

1 Introduction

When processor clock speeds flatlined in 2004, after more than fifteen years of exponential increases, the era of routine and near automatic performance improvements that the HPC application community had previously enjoyed came

* Research reported here was partially supported by the National Science Foundation, Microsoft Research, and NVIDIA.

to an abrupt end. CPU designs moved to multicores and are currently going through a renaissance due to the need for new approaches to manage the exponentially increasing (a) appetite for power of conventional system designs, and (b) gap between compute and communication speeds.

Compute Unified Device Architecture (CUDA) [1] based multicore platforms stand out among a confluence of trends because of their low power consumption and, at the same time, high compute power and bandwidth. Indeed, as power consumption is typically proportional to the cube of the frequency, accelerators using GPUs have a clear advantage against current homogeneous multicores, as their compute power is derived from many cores that are of low frequency. 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 on the order of 10 to 100$\times$ compared to current x86-based homogeneous multicore systems [2, 3]. The area of dense linear algebra (DLA) is no exception as evident from previous work on a single core with a single GPU accelerator [4–6], as well as BLAS for GPUs (see the CUBLAS library [7]).

Following those success stories, there is no doubt that the appeal of GPUs for high performance computing will definitely continue to grow. Another clear illustration of this can be seen in the interest of the recently announced *NVIDIA architecture*, code named “Fermi”, poised to deliver “supercomputing features and performance at $1/10^{th}$ the cost and $1/20^{th}$ the power of traditional CPU-only servers” [8].

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 development of fundamental numerical libraries for using the CPU-GPU in a hybrid manner. Major issues in terms of developing new algorithms, programmability, reliability, and user productivity have to be addressed. Our work is a contribution to the development of these libraries in the area of dense linear algebra and will be included in the *Matrix Algebra for GPU and Multicore Architectures* (MAGMA) Library [5]. Designed to be similar to LAPACK in functionality, data storage, and interface, the MAGMA library will allow scientists to effortlessly port their LAPACK-relying software components and to take advantage of the new hybrid architectures.

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) [9] and MAGMA libraries. On one hand, the tile algorithm concepts from PLASMA allow the computation to be split into tiles along with a 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 work-load 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 communication 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. Indeed, although the computing power of order 1 TFlop/s is concentrated in the GPUs, communication between them are still performed using the CPUs as a gateway, which only offers a shared connection on the order of 1 GB/s. 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 paper is organized as follows. Section 2 recalls the basic principles of the Cholesky factorization. Section 3 highlights the fundamental mechanisms of the PLASMA project. Section 4 describes the hybrid computation approach in the MAGMA project. Section 5 introduces the principles of the new technique, which permits the overall algorithm to scale on multiple GPUs. It also gives implementation details about various Cholesky versions using different levels of optimizations. Section 6 presents the performance results of those different versions. Section 7 describes the on-going work in this area and finally, Section 8 summarizes this work.

2 The Cholesky Factorization

The Cholesky factorization (or Cholesky decomposition) is mainly used for the numerical solution of linear equations $Ax = b$, where A is symmetric and positive definite. Such systems arise often in physics applications, where A is positive definite due to the nature of the modeled physical phenomenon. This happens frequently in numerical solutions of partial differential equations.

The Cholesky factorization of an $N \times N$ real symmetric positive definite matrix A has the form

$$A = LL^T,$$

where L is an $N \times N$ real lower triangular matrix with positive diagonal elements. In LAPACK, the algorithm is implemented by the xPOTRF routines. The "x" in front of the routine name refers to the precision arithmetic. A single step of the algorithm is implemented by a sequence of calls to the LAPACK and BLAS routines: xSYRK, xPOTF2, xGEMM, xTRSM. Due to the symmetry, the matrix can be factorized either as upper triangular matrix or as lower triangular matrix. Here the lower triangular case is considered.

The algorithm can be expressed using either the top-looking version, the left-looking version or the right-looking version, the first being the most *lazy* algorithm (depth-first exploration of the task graph) and the last being the most *aggressive* algorithm (breadth-first exploration of the task graph). The left-looking variant is used throughout the paper, and performance numbers will be reported in both single and double precision arithmetic.

3 The PLASMA Framework

The Parallel Linear Algebra for Scalable Multicore Architectures (PLASMA) project aims to address the critical and highly disruptive situation that is facing the Linear Algebra and high performance computing community due to the introduction of multicore architectures.

3.1 Tile Algorithm Concept

PLASMA's ultimate goal is to create a software framework that enable programmers to simplify the process of developing applications that can achieve both high performance and portability across a range of new architectures. In LAPACK, parallelism is obtained through the use of multi-threaded Basic Linear Algebra Subprograms (BLAS) [10]. In PLASMA, parallelism is no longer hidden inside the BLAS but is brought to the fore in order to yield much better performance. Our programming model enforces asynchronous, out of order scheduling of operations. This concept is used as the basis for a scalable yet highly efficient software framework for computational linear algebra applications.

To achieve high performance on multicore architectures, PLASMA relies on tile algorithms, which provide fine granularity parallelism. PLASMA performance strongly depends on tunable execution parameters trading off utilization of different system resources. One of the parameters is obviously the outer block/tile size (NB) which trades off parallelization granularity and scheduling flexibility with single core utilization. The standard linear algebra algorithms can then be represented as a Directed Acyclic Graphs (DAG) where nodes represent tasks and edges represent dependencies among them.

3.2 Runtime Environment

To schedule the tasks represented in the DAG, PLASMA v2.0 uses a static pipeline scheduling, originally implemented for dense matrix factorizations on the IBM CELL processor [11, 12]. This technique is extremely simple yet provides good locality of reference and load balance for a regular computation, like dense matrix operations. In this approach each task is uniquely identified by the m, n, k triple, which determines the type of operation and the location of tiles operated upon. Each core traverses its task space by applying a simple formula to the m, n, k triple, which takes into account the id of the core and the total number of cores in the system.

Task dependencies are tracked by a global progress table, where one element describes progress of computation for one tile of the input matrix. Each core looks up the table before executing each task to check for dependencies and stalls if dependencies are not satisfied. Each core updates the progress table after the completion of each task. Access to the table does not require mutual exclusion (using, e.g., mutexes). The table is declared as volatile. Update is implemented by writing to an element. Dependency stall is implemented by busy-waiting on an element.

The use of a global progress table is a potential scalability bottleneck. However, it does not pose a problem on small-scale multicore/SMP systems for small to medium matrix sizes. Many alternatives are possible. Replicated progress tables were used on the IBM CELL processor [11, 12]. This technique allows for pipelined execution of factorizations steps, which provides similar benefits to dynamic scheduling, namely, execution of the inefficient Level 2 BLAS operations in parallel with the efficient Level 3 BLAS operations. The main disadvantage of the technique is potentially suboptimal scheduling, i.e., stalling in situations where work is available. Another obvious weakness of the static schedule is that it cannot accommodate dynamic operations, e.g., divide-and-conquer algorithms.

3.3 PLASMA Cholesky

The tile Cholesky algorithm is identical to the block Cholesky algorithm implemented in LAPACK, except for processing the matrix by tiles. Otherwise, the exact same operations are applied. The algorithm relies on four basic operations implemented by four computational kernels (Figure 1):

xSYRK: The kernel applies updates to a diagonal (lower triangular) tile T of the input matrix, resulting from factorization of the tiles A to the left of it. The operation is a symmetric rank-k update.

xPOTRF: The kernel performs the Cholesky factorization of a diagonal (lower triangular) tile T of the input matrix and overrides it with the final elements of the output matrix.

xGEMM: The operation applies updates to an off-diagonal tile C of the input matrix, resulting from factorization of the tiles to the left of it. The operation is a matrix multiplication.

xTRSM: The operation applies an update to an off-diagonal tile C of the input matrix, resulting from factorization of the diagonal tile above it and overrides it with the final elements of the output matrix. The operation is a triangular solve.

Figure 2 shows the task graph of the tile Cholesky factorization of a 5×5 tiles matrix. Although the code is as simple as four loops with three levels of nesting, the task graph is far from intuitive, even for a tiny size.

Figure 3 shows the factorization steps of the left-looking tile Cholesky using four cores on a 5×5 tile matrix. At a given panel factorization step, matrix rows are processed by cores in one-dimensional cyclic fashion which engenders lots of data reuse.

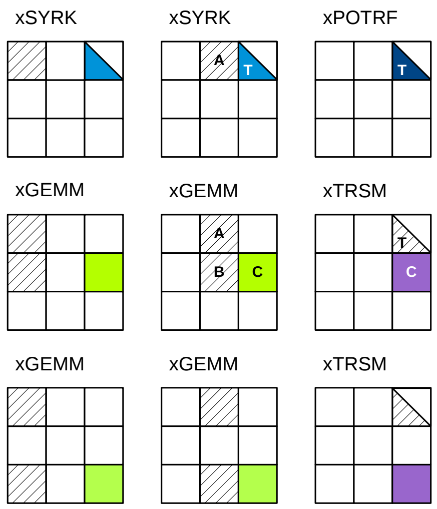


Fig. 1. Tile operations in Cholesky factorization.

The performances of the single and double precision tile Cholesky are shown in Figure 4 on a quad-socket, quad-core machine based on an Intel Xeon EMT64 E7340 processor operating at 2.39 GHz. The theoretical peak is 307.2 Gflop/s in single and 153.6 Gflop/s in double precision. Both single and double precision tile Cholesky outperform by far LAPACK and MKL v10.1 [13] on 16 cores. Performance results as well as comparisons against the state of the art numerical libraries for the other one-sided factorizations (i.e., LU and QR) are presented in [14].

4 The MAGMA Framework

The goal of the MAGMA project is the development of a new generation of linear algebra libraries that achieve the fastest possible time to an accurate solution on hybrid/heterogeneous architectures, starting with current multicore with GPU accelerators systems. To address the complex challenges stemming from these systems' heterogeneity, massive parallelism, and gap in compute power *vs* CPU-GPU communication speeds, MAGMA's research is based on the idea that optimal software solutions will themselves have to hybridize, combining the strengths of different algorithms within a single framework. Building on this idea, we design linear algebra algorithms and frameworks for hybrid multicore and multiGPU systems that can enable applications to fully exploit the power that each of the hybrid components offers.

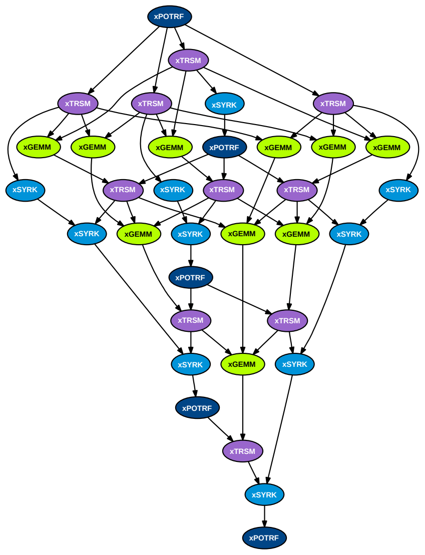


Fig. 2. Task graph of tile Cholesky factorization (5×5 tiles).

4.1 Hybridization of DLA algorithms

We split the computation into sub-tasks and schedule their execution over the system's hybrid components. The splitting itself is simple, as it is based on splitting BLAS operations. The challenges are choosing the granularity (and shape) of the splitting and the subsequent scheduling of the sub-tasks. It is desired that the splitting and scheduling (1) allow for asynchronous execution and load balance among the hybrid components, and (2) harness the strengths of the components of a hybrid architecture by properly matching them to algorithmic/task requirements. We call this process *hybridization* of DLA algorithms. We have developed hybrid algorithms for both one-sided [15, 16] and two-sided factorizations [6]. Those implementations have been released in the current MAGMA library [17]. The task granularity is one of the key for an efficient and balanced execution. It is parametrized and tuned empirically at software installation time [18].

4.2 Scheduling of hybrid DLA algorithms

The scheduling on a parallel machine is crucial for the efficient execution of an algorithm. In general, we aim to schedule the execution of the critical path of an algorithm with a higher priority. 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).

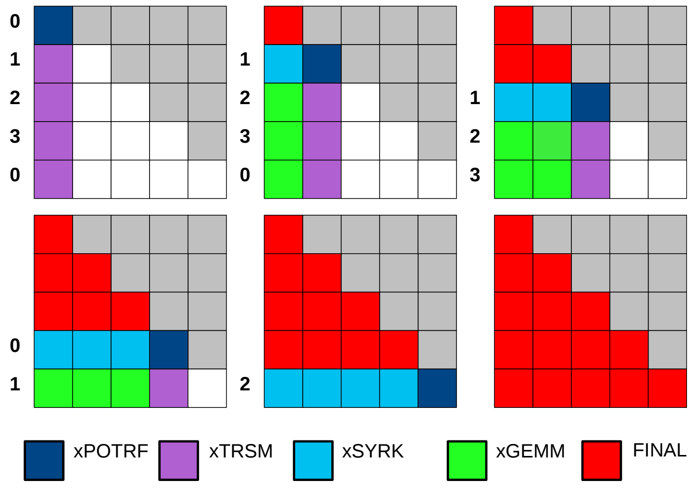
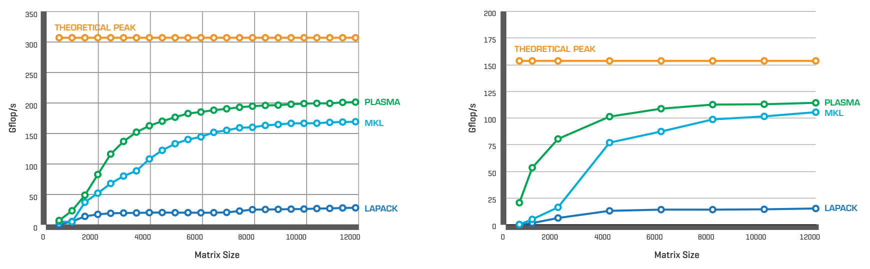


Fig. 3. Scheduling scheme of left-looking tile Cholesky factorization (5×5 tiles).

In the case of one-sided factorizations, the panel factorizations are scheduled on the CPU, and the Level 3 BLAS updates on the trailing sub-matrices are scheduled on the GPU. The task splitting and scheduling are such that we get the effect of the so called look-ahead technique, e.g. used before in the LINPACK benchmark [19], that allows us to overlap the CPU's and GPU's work (and some communication associated with it). In the case of two-sided factorizations (e.g., Hessenberg reduction), the panel factorization involves the computation of large matrix-vector products that are bottlenecks for CPU computing. Therefore these tasks are scheduled on the GPU to exploit its larger bandwidth (currently 10×). Further details can be found in [6, 15].

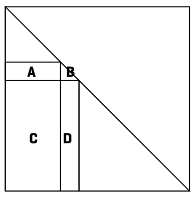
4.3 MAGMA Cholesky

MAGMA uses the left-looking version of the Cholesky factorization. Figure 5 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 back and forth data 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 6 using the NVIDIAs GeForce GTX 280 GPU and its multicore host, a dual socket quad-core Intel Xeon running at 2.33 GHz. The hybrid MAGMA Cholesky factorization runs asymptotically at 300 Gflop/s in single and almost 70 Gflop/s in double precision arithmetic.



(a) Single Precision. (b) Double Precision.

Fig. 4. Parallel performance of tile Cholesky with MKL BLAS V10.1 on a quad-socket, quad-core machine based on an Intel Xeon EMT64 E7340 processor operating at 2.39 GHz.



MATLAB code	LAPACK code	Hybrid code
[1] B = B – A*A'	ssyrk_("L", "N", &nb, &j, &mone, hA[j,0], ...]	cublasSsyrk('L', 'N', nb, j. mone, dA[j,0], ...]
		cublasGetMatrix[nb, nb, 4, dA[j, j], *lda, hwork, nb]
[2] B = chol[B, 'lower']	spotrf_("L", &nb, hA[j, j], lda, info]	cublasSgemm('N', 'T', j, ...]
[3] D = D – C*A'	sgemm_("N", "T", &j, ...]	spotrf_("L", &nb, hwork, &nb, info]
		cublasSetMatrix[nb, nb, 4, hwork, nb, dA[j, j], *lda]
[4] D = D\B	strsm_("R", "L", "T", "N", &j, ...]	cublasStrsm('R', 'L', 'T', 'N', j, ...]

Fig. 5. 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).

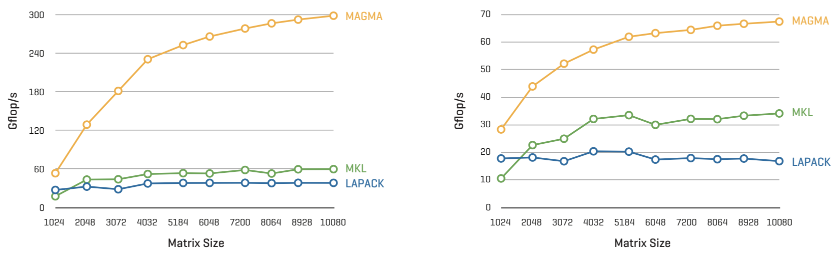
5 Cholesky Factorization on Multicore+MultiGPUs

In this section, we describe our new technique to efficiently perform the Cholesky factorization on a multicore system enhanced with multiple GPUs.

5.1 Principles and Methodology

This section represents our main twofold contribution.

First, the idea is to extend the runtime environment (RTE) of PLASMA, namely the static scheduler, to additionally handle computation on GPUs. Instead of assigning tasks to a single CPU, the static scheduler is now able to assign



(a) Single Precision. (b) Double Precision.

Fig. 6. Parallel performance of MAGMA's hybrid Cholesky on GTX 280 *vs* MKL 10.1 and LAPACK (with multi-threaded BLAS) on Intel Xeon dual socket quad-core 2.33GHz

tasks to a CPU+GPU couple. Each CPU host is dedicated to a particular GPU device to offload back and forth data. PLASMA's RTE ensures dependencies are satisfied before a host can actually trigger the computation on its corresponding device. Moreover, the four kernels to compute the Cholesky factorization, as described in Section 3, need to be redefined. Three out of the four kernels, i.e., xTRSM, xSYRK and xGEMM, can be efficiently executed on the GPU using CUBLAS or the MAGMA BLAS libraries. In particular, we developed and used optimized xTRSM and xSYRK (currently included in MAGMA BLAS). But most importantly, the novelty here is to replace the xPOTRF LAPACK kernel by the corresponding hybrid MAGMA kernel, following the guidelines described in Section 4. High performance on this kernel is achieved by allowing both host and device to factorize the diagonal tile together in a hybrid manner. This is paramount to improve the kernel because the diagonal tiles are located in the critical path of the DAG.

Second, we developed a data persistence strategy that optimizes the number of transfers between the CPU hosts and GPU devices, and vice versa. Indeed, the host is still the only gateway to any transfers occurring between devices which appears to be a definite bottleneck if communication are not handled cautiously. To bypass this issue, the static scheduler gives us the opportunity to precisely keep track of the location of any particular data tile during runtime. One of the major benefits of such a scheduler is that each processing CPU+GPU couple knows ahead of time its workload and can determine where a data tile resides. Therefore, many assumptions can be taken before the actual computation in order to limit the amount of data transfers to be performed.

The next sections present incremental implementations of the new tile Cholesky factorization on multicore with GPU accelerators systems. The last implementation is the most optimized version containing both contributions explained above.

5.2 Implementations Details

We describe four different implementations of the tile Cholesky factorization designed for hybrid systems. Each version introduces a new level of optimizations and simultaneously includes the previous ones. Again, each GPU device is dedicated to a particular CPU host, and this principle holds for all versions described below.

5.3 Memory optimal

This version of the tile Cholesky factorization is very basic in the sense that the static scheduler from PLASMA is reused out of the box. The scheduler gives the green light to execute a particular task after all required dependencies have been satisfied. Then, three steps occur in the following order:

1. The core working on that task triggers the computation on its corresponding GPU by offloading the necessary data.
2. The GPU then performs the current computation.
3. The specific core requests the freshly computed data back from the GPU.

Those three steps are repeated for all kernels except for the diagonal factorization kernel, i.e., xPOTRF, where no data transfers are needed since the computation is only done by the host. This version only requires, at most, the size of three data tiles to be allocated on the GPU (due to the xGEMM kernel). However, the amount of communication involved is tremendous. To be more specific, for each kernel call (except xPOTRF) at any given time, two data transfers are needed (step 1 and 3).

5.4 Data Persistence Optimizations

In this implementation, the amount of communication is significantly decreased by trading off the amount of memory allocated on GPUs. To understand how this works, it is important to mention that each data tile located on the left side of the current panel being factorized corresponds to the final output, i.e., they are not transient data tiles. And this is obviously due to the nature of the left-looking Cholesky factorization.

Therefore, the idea is to keep in GPU's memory any data tile loaded for a specific kernel while processing the panel, in order to be eventually reused by the same GPU for other subsequent kernels. After applying all operations on a specific data tile located on the panel, each GPU device uploads back to its CPU host the final data tile to ensure data consistency between hosts/devices for the next operations. As a matter of fact, another progress table has been implemented to determine whether a particular data tile is already present in the device's memory or actually needs to be uploaded from host's memory. This technique requires, at most, the amount of half the matrix to be stored in GPU's memory.

Besides optimizing the number of data transfers between hosts and devices, we also try to introduce in this version asynchronous communication to overlap communication by computations. This is more of a hardware optimization, in the sense that it is left up to the GPU hardware to hide the overhead of communication.

5.5 Hybrid xPOTRF Kernel

The implementation of this version is straightforward. The xPOTRF kernel has been replaced by the hybrid xPOTRF MAGMA kernel, where both host and device compute the factorization of the diagonal tile (see Section 4.3).

5.6 xSYRK and xTRSM Kernel Optimizations

This version integrates new implementations of the BLAS xSYRK and xTRSM routines, which are highly optimized for GPU computing as explained below.

xSYRK The symmetric rank k-update operation has been improved both in single and double precision arithmetic compared to CUDA-2.3. A block index reordering technique is used to initiate and limit the computation only to blocks that are on the diagonal or in the lower (correspondingly upper) triangular part of the matrix. In addition, all the threads in a diagonal block are responsible to compute redundantly half of the block in a data parallel fashion in order to avoid expensive conditional statements that would have been necessary otherwise. Some threads also load unnecessary data so that data is fetched from global memory in a coalesced manner. At the time the computation is over, the results from the redundant computations (in the diagonal blocks) are simply discarded and the data tile is correctly updated.

xTRSM Algorithms that trade off parallelism and numerical stability, especially in algorithms related to triangular solvers, have been known and studied before [20, 21]. Some of them are getting extremely relevant with the emerging highly parallel architectures, e.g., GPUs, and are now implemented in the MAGMA library [17]. Here in particular, similarly to the ideas developed in [4, 16], we explicitly invert blocks of size 32×32 on the diagonal of the matrix and use them in blocked xTRSM algorithms. The inverses are computed simultaneously, using one GPU kernel, so that the critical path of the blocked xTRSM can be greatly reduced by doing it in parallel (as a matrix-matrix multiplication). We have implemented multiple kernels, including kernels where the inverses are computed on the CPU, with various block sizes (e.g., recursively increasing it from 32), etc, but this kernel performed best for the tile sizes used in the Cholesky factorization (see Section 6.2) and the particular hardware configuration (highly imbalanced CPU *vs* GPU speeds). Finally, similarly to xSYRK, extra flops are performed to reach better performance – the empty halves of the diagonal triangular matrices are set to zeros and the multiplications with them are done with xGEMMs instead of with xTRMMs. This avoids threads from diverting in a warp and ensures efficient parallel execution.

6 Experimental Results

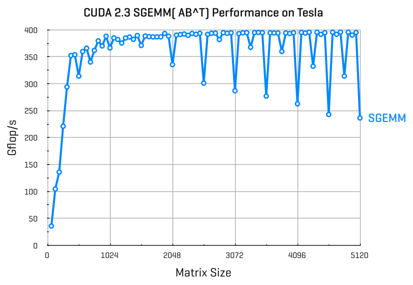
6.1 Environment Setup

The experiments have been performed on a quad-socket quad-core host machine based on an AMD Opteron(tm) Processor 8358 SE, operating at 2.4 GHz. The cache size per core is 512 KB and the size of the main memory is 32 GB. The NVIDIA S1070 graphical card is connected to the host via PCI Express 16x adapter cards. 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. Each processing core has eight SIMD functional units and each functional unit can issue three floating point operations per cycle (1 mul-add + 1 mul = 3 flops). The single precision theoretical peak performance of the S1070 card is then $30 \times 8 \times 3 \times 1.44 \times 4 = 4.14$ TFlop/s. However, only two flops per cycle can be used for general purpose computations in our dense linear algebra algorithm (1 mul-add per cycle). So, in our case, the single precision peak performance drops to $2/3 \times 4.14 = 2.76$ TFlop/s. The double precision peak is computed similarly with the only difference being that there is only one SIMD functional unit per core, i.e., the peak will be $30 \times 1 \times 2 \times 1.44 \times 4 = 345$ Gflop/s. The host machine is running Linux 2.6.33 and provides GCC Compilers 4.1.2 together with the CUDA 2.3 library. All the experiments presented below focus on asymptotic performance and have been conducted on four cores and four GPUs.

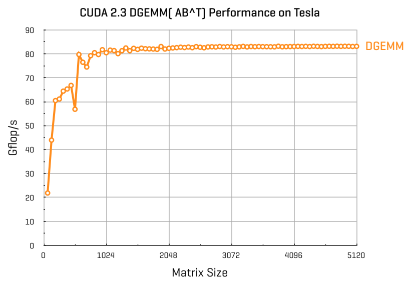
6.2 Tuning

The performance of the new factorization strongly depends on tunable execution parameters, most notably various block sizes for the two levels of nested parallelism in the algorithm, i.e., the outer and inner block sizes. These parameters are usually computed from an auto-tuning procedure (e.g., established at installation time) but for now, manual tuning based on empirical data is used to determine their close to optimal values.

The selection of the tile size (the outer blocking size) is determined by the performance of the most compute intensive tile operation/kernel, i.e., xGEMM in our case. The goal is to determine from which tile size the performance of xGEMM on a single GPU starts to asymptotically flatten. From Figures 7(a) and 7(b), the flattening starts at matrix dimension around 500 in single and 800 in double. Several sizes around those were tested to finally select the best performing ones, namely $b_s = 576$ in single and $b_d = 832$ in double precision arithmetic, respectively. Noteworthy to mention is that the SGEMM (AB^T case) CUBLAS 2.3 kernel shows performance deteriorations in certain dimensions. Those specific sizes were obviously discarded from the selection mechanism.



(a) Single Precision.



(b) Double Precision.

Fig. 7. Performance of the xGEMM kernel on a single GPU.

The selection of the inner blocking sizes for the splitting occurring within the hybrid kernels (i.e., MAGMA's xPOTRF) and the GPU kernels (i.e., MAGMA BLAS's xSYRK, xTRSM, xGEMM) is done similarly, based on empirical data for problem sizes around 500 and 800 for single and double precision arithmetic, respectively [18]. Some of the MAGMA BLAS kernels (e.g. xTRSM and xGEMM) have different implementations and, again, the best performing ones were selected based on empirical data.

6.3 Parallel GEMM with PCI-aware Communication Techniques

A parallel xGEMM was implemented in order to make a reasonable guess about the achievable peak performance of the reference system configuration. Although the system's peak performance is 2.76 TFlop/s in single and 345 GFlop/s in double precision arithmetic, the embarrassingly parallel xGEMM, i.e., small independent xGEMMs running on the four GPUs without any communication, achieves correspondingly 58% and 96% of those peaks (see Figure 7). As there is huge gap between the GPU's compute power *vs* the CPU-GPU communication speed, it is relevant to know the performance peak of a multiGPU xGEMM (i.e., the performance when communication is included).

Several versions were developed in order to investigate various optimization techniques. All versions feature a static scheduler that distributes computational tasks into the task queues of four threads, running on different cores, each thread/core associated with a GPU device. The initial xGEMM task

$$C := \alpha AB + \beta C,$$

with A, B, and C given on the CPU memory (with result C expected on the CPU memory), is split into four xGEMM tasks by two-dimensional block-cyclic data distribution. Each thread, after getting its corresponding xGEMM task form its own queue, transfers the entire necessary for its computation part of A to the device. Next, the computation of the local xGEMM task is additionally blocked following a one-dimensional block-cyclic column distribution for the locally needed sub-matrices of B and C. In particular, a thread would initiate a CPU-to-GPU data transfer for a block of columns for the locally needed sub-matrices of B and C, followed by the xGEMM involving the just transferred data.

Note that the above framework is designed to **reduce communication** by the two-dimensional block partitioning of the original xGEMM task, to **overlap communication and computation** by using the internal blocking, and to **hide communication latencies** by blocking the data transfers. Additionally, variants of the algorithm were tested and empirically tuned. Most notably, we developed **PCI-aware communication techniques** to efficiently use the available PCI bandwidth. This is very important because communications are a bottleneck in multi-GPU computing, as previously stressed. The motivation and insight for developing these techniques can be explained by the following experiment.

We measure the throughput for transferring square matrices of different sizes between host and devices, using combinations of the four GPUs available in the system. There are two PCI buses for the particular set up of our system. GPUs 0 and 1 share one PCI bus whereas GPUs 2 and 3 share the other. We measured the throughput per GPU while using the following combinations of GPUs: {0}, {0, 1}, {0, 2}, {0, 1, 2} and {0, 1, 2, 3}. The throughput achieved (per GPU) for the different combinations of GPUs are shown in Figure 8. The low throughput

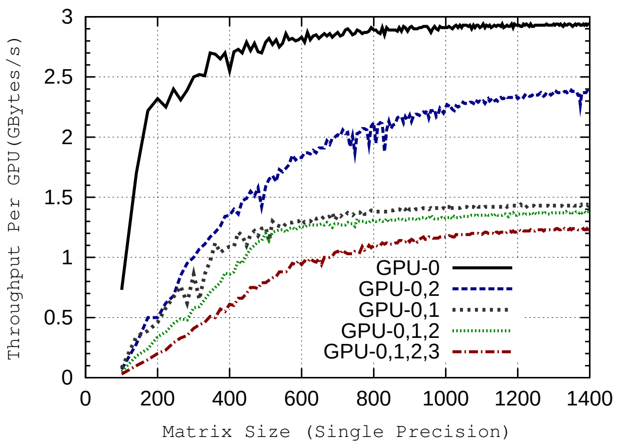
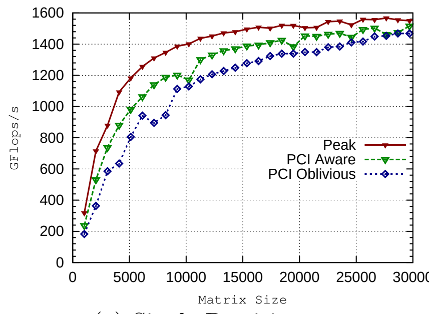


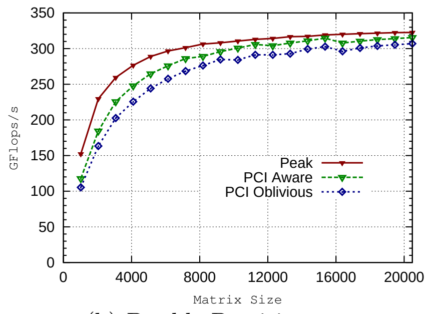
Fig. 8. CPU-to-GPU throughput achieved per GPU on transferring square matrices using combinations of four GPUs sharing two PCI buses.

for small matrices is due to latencies, which in our case are about 17 μs for CPU-to-GPU transfers (and about 20 μs for GPU-to-CPU transfers). Note that the result clearly shows that if two GPUs share the same bus the throughput per GPU is significantly lower than the case if they do not. Moreover, note that in case of all four GPUs transferring data, if we enforce first GPUs {0, 2} to transfer, followed by GPUs {1, 3}, not only the overall transfer would be faster, but also a group of two GPUs would always be free to do computation while the other group is transferring data.

These observations motivated us to develop PCI-aware communication techniques. One example technique is, as mentioned in the motivation above, to enforce that only one GPU at a time is using a PCI bus. The effect of this technique on the performance of xGEMM is shown in Figure 9. The parallel SGEMM



(a) Single Precision.



(b) Double Precision.

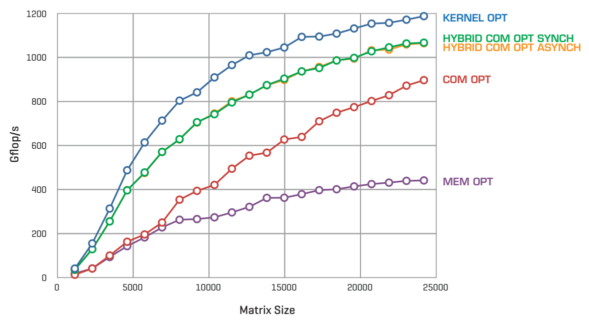
Fig. 9. Performance of the xGEMM with 4 Tesla C1060 GPUs.

and DGEMM reach up to 1.52 TFlop/s and 316 Gflop/s respectively. These results are 96% of the embarrassingly parallel SGEMM and DGEMM peaks using four GPUs. The PCI-aware technique yields performance improvements of up 246 GFlop/s in single, and up to 23 GFlop/s in double precision arithmetic. In terms of percentages, the improvements compared to the PCI-oblivious implementation are up to 46% and 13% respectively. The "Peak" curve in Figure 9 marks the performance achieved when three of the four threads are disabled, i.e., the running thread would not encounter any competition for the shared resources, and would achieve its peak throughput per GPU (see Figure 8).

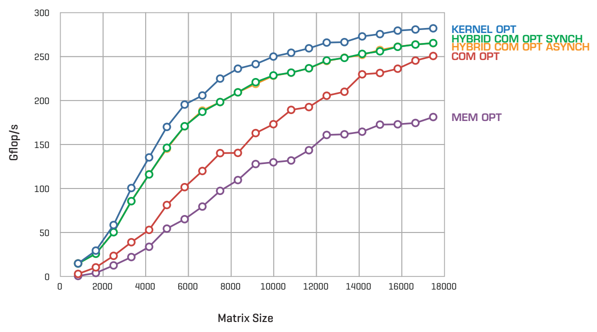
The performance of this parallel Matrix Multiplication over multiple GPUs is used as an upper bound for the Cholesky factorization on multiple GPUs presented in the next section.

6.4 Performance Results

Figure 10 shows the incremental performance in single and double precision arithmetic of the tile hybrid Cholesky factorization using the entire system resources, i.e. four CPUs and four GPUs. Each curve represents one version of the Cholesky factorization. The memory optimal version is very expensive due to the high number of data transfers occurring between hosts and devices. The communication optimal or data persistence techniques trigger a considerable boost in the overall performance, especially for single precision arithmetic. The integration of the hybrid kernel (i.e., MAGMA's xPOTRF) to accelerate the execution of tasks located on the critical path improves further the performance. To our surprise, we did not see any improvements compared to the synchronous version. Most probably this feature is not yet handled efficiently at the level of the driver. Finally, the additional optimizations performed on the other MAGMA BLAS kernels (i.e., MAGMA BLAS's xSYRK, xTRSM, xGEMM) make the Cholesky factorization reaches up to 1.189 TFlop/s for single and 282 Gflop/s for double precision arithmetic. Compared with the performance of the parallel xGEMM over four GPUs, our algorithm runs correspondingly at 78% and 89%.



(a) Single Precision.

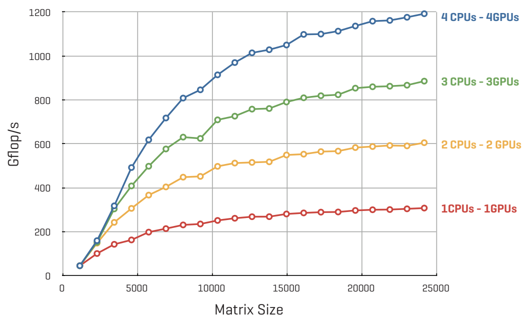


(b) Double Precision.

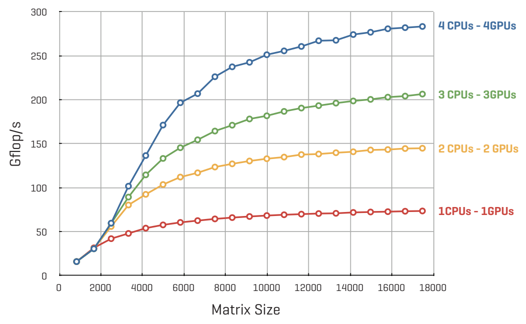
Fig. 10. Performance comparisons of various implementations.

6.5 Strong Scalability

Figure 11 highlights the scalable speed-up of the tile hybrid Cholesky factorization (i.e., the kernel optimization version) using four CPUs - four GPUs in single and double precision arithmetics. The performance doubles as the number of CPU-GPU couples doubles.



(a) Single Precision.



(b) Double Precision.

Fig. 11. Speed up of the tile hybrid Cholesky factorization.

6.6 Tracing

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 12 with four GPUs. Each task row corresponds to a particular GPU trace execution. The different kernels are clearly identified by their colors described initially in Figure 3. 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.

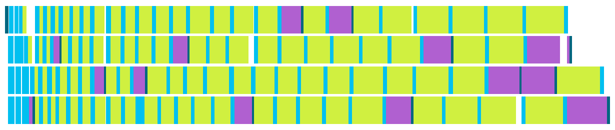


Fig. 12. Execution trace of the hybrid tile Cholesky on four GPUs.

7 Related Work

Several authors have presented work on multiGPU algorithms for dense linear algebra. Volkov and Demmel [4] presented an LU factorization for two GPUs (NVIDIA GTX 280) running at up to 538 GFlop/s in single precision. The algorithm uses 1-D block cyclic partitioning of the matrix between the GPUs and achieves 74% improvement *vs* using just one GPU. Although extremely impressive, it is not clear if this approach will scale for more GPUs, especially by taking into account that the CPU work and the CPU-GPU bandwidth will not scale (and actually will remain the same with more GPUs added).

Closer in spirit to our work is [22]. The authors present a Cholesky factorization and its performance on a Tesla S1070 (as we do) and a host that is much more powerful than ours (two Intel Xeon Quad-Core E5440 @2.83 GHz). It is interesting to compare with this work because the authors, similarly to us, split the matrix into tiles and schedule the corresponding tasks using a dynamic scheduling. Certain optimizations techniques are applied but the best performance obtained is only close to our memory optimal version, which is running three times slower compared to our best version. The algorithm presented in here performs better for a set of reasons, namely the data persistence optimization techniques along with the efficiency of our static scheduler, the integration of the hybrid kernel, and the overall optimizations of the other GPU kernels.

8 Summary and Future Work

This paper shows how to redesign the Cholesky factorization to greatly enhance its performance in the context of multicore systems with GPU accelerators. The results obtained demonstrate that scalable high performance is achievable on these systems despite the challenges related to their heterogeneity, massive parallelism, and huge gap between the GPUs' compute power vs the CPU-to-GPU communication speed. The redesign enables the efficient cooperation between the cores of a multicore and four NVIDIA GPUs (30 cores per GPU, @1.44 GHz per core). By reusing concepts developed in the PLASMA (i.e., static scheduling) and MAGMA (i.e., hybrid computations) libraries along with data persistence techniques, we achieve an astounding performance of $1,189$ TFlop/s in single and 282 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 78% and 89% for single and double precision arithmetic respectively. This hybrid algorithm will eventually be included in the future release of MAGMA. Future work includes the integration of the whole multicore system resources to be part of the computation. It would relieve the actual constraint of having the number of GPUs matching the number of cores. Finally, it is noteworthy to also mention that this work could be extended to LU and QR factorizations.

References

1. NVIDIA CUDA Compute Unified Device Architecture - Programming Guide. http://developer.download.nvidia.com, 2007.
2. NVIDIA CUDA ZONE. http://www.nvidia.com/object/cuda_home.html.
3. General-purpose computation using graphics hardware. http://www.gpgpu.org.
4. 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.
5. S. Tomov, J. Dongarra, V. Volkov, and J. Demmel. MAGMA Library, version 0.1. http://icl.cs.utk.edu/magma, 08/2009.
6. S. Tomov and J. Dongarra. Accelerating the reduction to upper Hessenberg form through hybrid GPU-based computing. Technical Report 219, LAPACK Working Note, May 2009.
7. CUDA CUBLAS Library. http://developer.download.nvidia.com.
8. NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/object/fermi_architecture.html, 2009.
9. 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.
10. BLAS: Basic linear algebra subprograms. http://www.netlib.org/blas/.
11. J. Kurzak, A. Buttari, and J. J. Dongarra. Solving systems of linear equations on the CELL processor using Cholesky factorization. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1–11, September 2008.
12. Jakub Kurzak and Jack Dongarra. QR factorization for the cell broadband engine. *Scientific Programming*, 17(1-2):31–42, 2009.
13. http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm.
14. Emmanuel Agullo, Bilel Hadri, Hatem Ltaief, and Jack J. Dongarra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. *University of Tennessee Computer Science Technical Report (also LAPACK Working Note 217), Accepted for publication at SuperComputing 2009.*
15. S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. Technical report, LAPACK Working Note 210, October 2008.
16. M. Baboulin, J. Dongarra, and S. Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. Technical report, LAPACK Working Note 200, May 2008.
17. S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 User Guide. http://icl.cs.utk.edu/magma, 11/2009.
18. Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

19. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:820, 2003.
20. James W. Demmel. Trading off parallelism and numerical stability. Technical Report UCB/CSD-92-702, EECS Department, University of California, Berkeley, Sep 1992.
21. Nicholas J. Higham. Stability of parallel triangular system solvers. *SIAM J. Sci. Comput.*, 16(2):400–413, 1995.
22. Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.