A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators

H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra [⋆]

Department of Electrical Engineering and Computer Science,
University of Tennessee, Knoxville
{ltaief, tomov, rnath1, du, dongarra}@eecs.utk.edu

Abstract. We present a Cholesky factorization for multicore with GPU accelerators systems. 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. 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 two dual-core AMD Opteron @1.8GHz processors, we reach 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.

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 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,

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

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× compared to current x86-based homogeneous multicore systems [2]. The area of dense linear algebra (DLA) is no exception as evident from previous work on a single core with a single GPU accelerator [3, 4], as well as BLAS for GPUs (see the CUBLAS library [5]). 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 [9]. 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) [6] 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 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. Indeed, although the computing power of order 1 TFlop/s is concentrated in the GPUs, communications 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 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 3 presents the performance results of those different versions. Section 4 describes the on-going work in this area and finally, Section 5 summarizes this work.

2 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.

2.1 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 [7], to additionally handle computation on GPUs. Instead of assigning tasks to a single CPU, the static scheduler is now able to assign 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, there are four kernels to compute the Cholesky factorization and they need to be redefined (from PLASMA). Three of them – 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 kernel from MAGMA. 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 algorithm.

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 communications 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.

2.2 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. Each GPU device is dedicated to a particular CPU host, and this principle holds for all versions described below.

2.3 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 this following order. First, the core working on that task triggers the computation on its corresponding GPU by offloading the necessary data. Second, the GPU performs the current computation. Third, 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 as for each kernel call (except xPOTRF) , two data transfers are needed (steps one and three).

2.4 Data Persistence Optimizations

In this implementation, the amount of communications 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 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 asynchronous communications to overlap communications by computations (using the `cudaMemcpy2DAsync` function and pinned CPU memory allocation).

2.5 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.

2.6 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: 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 (since the resulting matrix is symmetric). Thus, no redundant computations are performed for blocks off of the diagonal. Only the threads that would compute diagonal blocks are let 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.
xTRSM: Similarly to [3, 8], 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 but this performed best for the tile sizes used in the Cholesky factorization (see Section 3.2) and our particular hardware configuration.

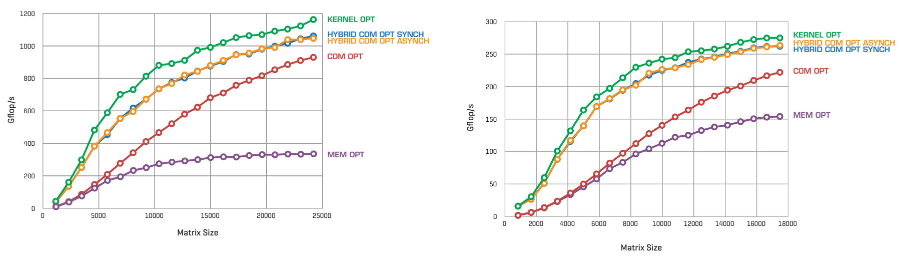
3 Experimental Results

3.1 Environment Setup

The experiments 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 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 2.6.18 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 the maximum amount of cores and GPUs available on the machine, i.e., four cores and four GPUs.

3.2 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 “optimal” values. The selection of the tile size (the outer blocking size) is determined by the performance of xGEMM. The goal is to determine from which tile size the performance of xGEMM on a single GPU starts to asymptotically flatten. Several values in that region were tested to finally select the best performing ones, namely $b_s = 576$ in single and $b_d = 832$ in double precision arithmetic. 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 [10].

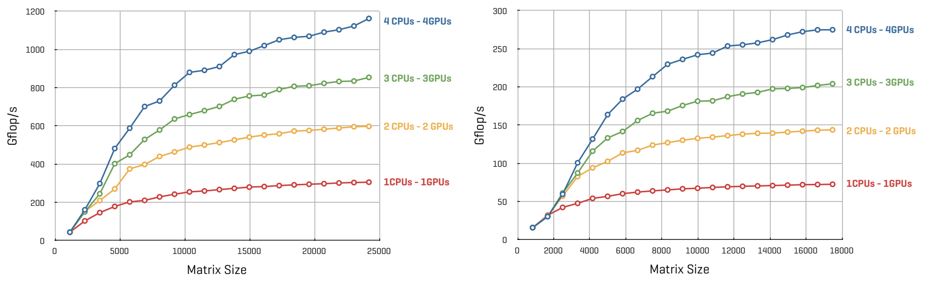


(a) Single Precision. (b) Double Precision.

Fig. 1. Performance comparisons of various implementations.

3.3 Performance Results

Figure 1 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 between the synchronous and the asynchronous 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 reach up to 1.163 Tflop/s for single and 275 Gflop/s for double precision arithmetic. Compared with the performance of the embarrassingly parallel xGEMM over four GPUs, i.e. $400 \times 4 = 1.6$ Tflop/s for single precision (58% of the theoretical peak of the NVIDIA card) and $82 \times 4 = 328$ Gflop/s for double precision arithmetic (95% of the theoretical peak of the NVIDIA card), our algorithm runs correspondingly at 73% and 84%. Figure 2 highlights the scalable speed-up of the tile hybrid Cholesky factorization 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. 2. Speed up of the tile hybrid Cholesky factorization.

4 Related Work

Several authors have presented work on multiGPU algorithms for dense linear algebra. Volkov and Demmel [3] 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 [11]. 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.

5 Summary and Future Work

This paper shows how to redesign the Cholesky factorization to greatly enhance its performance in the context of multicore with GPU accelerators systems. It initially achieves up to 20 GFlop/s in single and up to 10 GFlop/s in double precision arithmetic by using only two dual-core 1.8 GHz AMD Opteron processors. Adding four GPUs and redesigning the algorithm accelerates the computation up to 65× and 27× for single and double precision arithmetic respectively. This acceleration is due to a design that enables efficient cooperation between the four Opteron cores and the four NVIDIA GPUs (30 cores per GPU, @1.44 GHz per core). By reusing concepts developed in the PLASMA and MAGMA libraries along with data persistence techniques, we achieve an astounding performance of $1,163$ TFlop/s in single and 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. Although this paper focused only on the Cholesky factorization, a full high-performance linear solver is possible [12]. This hybrid algorithm will eventually be included in the future release of MAGMA. Future work includes the extension 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. V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In Proc. of *SC '08*, pages 1–11, Piscataway, NJ, USA, 2008.
4. S. Tomov and J. Dongarra. Accelerating the reduction to upper Hessenberg form through hybrid GPU-based computing. LAPACK Working Note 219, 05/2009.
5. CUDA CUBLAS Library. http://developer.download.nvidia.com.
6. 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.
7. 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.
8. M. Baboulin, J. Dongarra, and S. Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. LAPACK Working Note 200, 05/2008.

9. S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 User Guide. http://icl.cs.utk.edu/magma, 11/2009.
10. Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In Proc. of *ICCS '09*, pages 884–892, Baton Rouge, LA, 2009.
11. E. Ayguadé, R. Badia, F. Igual, J. Labarta, R. Mayo, and E. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In Proc. of *Euro-Par '09*, pages 851–862, Delft, The Netherlands, 2009.
12. S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. Proceedings of IPDPS 2010, Atlanta, GA, April 2010.