Toward Accelerating the Matrix Inversion Computation of Symmetric Positive-Definite Matrices on Heterogeneous GPU-Based Systems

Huda Ibeid[*][†], Dinesh Kaushik[‡], David Keyes[†] and Hatem Ltaief[‡]
[*]*Student Author*
[†]*Division of Mathematical and Computer Sciences and Engineering*
[‡]*Supercomputing Laboratory*
King Abdullah University of Science and Technology, Thuwal, KSA
Email: huda.ibeid,dinesh.kaushik,david.keyes,hatem.ltaief@kaust.edu.sa

Abstract*—The goal of this paper is to implement an efficient matrix inversion of symmetric positive-definite matrices on heterogeneous GPU-based systems. The matrix inversion procedure can be split into three stages: computing the Cholesky factorization, inverting the Cholesky factor and calculating the product of the inverted Cholesky factor with its transpose to get the final inverted matrix. Using high performance data layout, which represents the matrix in the system memory with an optimized cache-aware format, the computation of the three stages is decomposed into fine-grained computational tasks. The data flow programming model can then be represented as a directed acyclic graph, where nodes represent tasks and edges the dependencies between them. Standard implementations of matrix inversions as well as other numerical algorithms (e.g., linear and eigenvalue solvers), available in the state-of-the-art numerical libraries (e.g., LAPACK), rely on the expensive fork-join paradigm to achieve parallel performance and are characterized by artifactual synchronization points, which have to be removed to fully exploit the underlying hardware capabilities. Our tile algorithmic approach allows to remove those bottlenecks and to flawlessly execute the tasks, as soon as the data dependencies are satisfied. A hybrid runtime environment system becomes paramount to dynamically schedule the numerical kernels on the available processing units, whether it is a hardware accelerator (i.e, GPU) or a homogeneous multicore (i.e., x86), and this is transparently carried out from the user. Preliminary results are shown on a dual-socket quad-core Intel Xeon 2.67GHz workstation with two nVIDIA Fermi C2070 GPU cards. Our implementation (448 Gflop/s) results in up to 5 and 6-fold improvement compared to the equivalent routines from MAGMA V1.0 and PLASMA V2.4, respectively, and 10-fold improvement compared to LAPACK V3.2 linked with multithreaded Intel MKL BLAS V10.2, with a matrix size of 24960$\times$ 24960.

***Keywords*-Matrix Inversion; Tile Algorithms; Dynamic Load-Balanced Scheduling; Heterogeneous CPU-GPU Computing**

I. Introduction

The exascale roadmap [1] has clearly highlighted two possible swim lanes to achieve 10^{18} floating point operations by 2018. Indeed, the next-generation exascale platforms will most likely be composed of homogeneous x86 multicore (e.g., Blue Gene chips) or heterogeneous accelerator-based systems (e.g., GPU architecture). In both cases, high performance will have to be extracted using a bottom-up approach, starting from a single processing node up to the distributed system using a high-speed network interconnect.

In this paper, we focus on an efficient implementation of the matrix inversion for symmetric positive-definite matrices using an heterogeneous GPU-based workstation, which may be seen as a typical node constituting the future generation of high performance computing systems. Although the explicit matrix inversion computation may be in many cases unnecessary and inadvisable, it still represents a critical component of the variance-covariance matrix computation in statistics [2] (see p.260 §3 for more applications). The overall matrix inversion algorithm can be decomposed into three successive steps: (1) computing the Cholesky factorization, (2) inverting the Cholesky factor and (3) calculating the product of the inverted Cholesky factor with its transpose to get the final inverted matrix. Using tile algorithms [3], the original matrix is split into square tiles, in which the elements are contiguously stored in the system memory. This high performance data layout allows not only to match the algorithmic design with the underlying cache architecture, but also to expose more parallelism by removing artifactual synchronization points thanks to the fine-grained computational tasks. A directed acyclic graph can be used to characterize the resulting data flow programming model, where nodes represent tasks and edges the dependencies between them. The different tasks are then scheduled for execution as soon as their data dependencies are satisfied. A dynamic runtime environment system StarPU [4] is employed to transparently schedule the numerical kernels on the available processing units, whether it is a hardware accelerator (i.e, GPU) or a homogeneous multicore (i.e., x86). The preliminary results of our heterogeneous Cholesky-based matrix inversion implementation show almost half a Tflop/s (448 Gflop/s) on a dual-socket quad-core Intel Xeon 2.67GHz workstation with two nVIDIA Fermi C2070 GPU cards and achieve many fold speed up against the equivalent routines from the state-of-the-art numerical libraries i.e., LAPACK [5], PLASMA [3] and MAGMA [6].

The remainder of this paper is organized as follows: Section II gives a detailed overview of previous projects in this area. Section III presents the bottlenecks seen in the standard block Cholesky-based matrix inversion algorithm. Section IV explains how the tile algorithms associated with fine-grained tasks overcomes those bottlenecks. Section V introduces the new fine-grained and cache-friendly numerical kernels. Section VI describes the dynamic heterogeneous scheduler StarPU. The performance result are shown in Section VII, comparing our implementation with the state-of-the-art, high performance dense linear algebra software libraries. Finally, Section VIII summarizes the preliminary results of this paper and presents the ongoing work.

II. Related Work

A number of numerical software packages implement the matrix inversion of symmetric positive definite matrices.

LAPACK [5] is a library of Fortran 77 subroutines for solving those most commonly occurring problems in dense matrix computations. Using block algorithm, it has been designed to be efficient on a wide range of modern high-performance computers with memory hierarchy design. The name LAPACK is an acronym for Linear Algebra PACKage. LAPACK can solve systems of linear equations, linear least squares problems, eigenvalue problems and singular value problems. LAPACK can also handle many associated computations, such as matrix factorizations or estimating condition numbers. To compute the actual matrix inversion, the symmetric definite-positive matrix is first factorized using the routine DPOTRF (i.e., Cholesky factorization) followed by the routine DPOTRI to calculate the final inverted matrix.

MAGMA [6] is a dense linear algebra library similar to LAPACK but rather targeting heterogeneous architectures. The current MAGMA distribution focuses on multicore systems associated with a single GPU card. The main idea is to off-load the compute-intensive operations found in LAPACK algorithms (e.g. level 3 BLAS type of operations) to the graphical device, while the least parallel and inefficient ones are still handled by the CPU host. This allows to basically accelerate LAPACK out-of-box and effortlessly port existing LAPACK codes to heterogeneous platforms. The Cholesky-based matrix inversion has been implemented in MAGMA along those lines.

PLASMA [3] is also a dense linear algebra library developed to face the challenges brought by the introduction of homogeneous multicore architectures. Based on tile algorithms, the coarse-grained tasks in LAPACK algorithms are broken into smaller granularity to enforce asynchronous, out of order scheduling of operations. In particular, the three stage cholesky-based matrix inversion has been studied by Agullo et al. [7] in the context of PLASMA using homogeneous x86 processing units. The authors show very promising results, where computational tasks from the last stage can potentially overlap with tasks from the previous stages, as long as data dependencies are not violated for numerical correctness purposes. It is noteworthy to mention a very similar project FLAME [8], [9].

In this paper, the authors leverage the previous research works explained above to tackle more complex and challenging heterogeneous systems composed by homogeneous multicores associated with multiple hardware accelerators, thanks to the dynamic load balancing scheduling framework StarPU. In fact, this work proposed here represents a straight continuation of previous works for the one-sided factorizations [10]. Dealing with multiple stage numerical algorithms in the context of heterogeneous platforms along with new kernel implementations are the main contributions of this paper.

The next Section recalls the standard block algorithm to compute the matrix inversion of symmetric definite-positive matrices using the Cholesky factorization.

III. The Standard Block Cholesky-Based Matrix Inversion Algorithm

Block algorithms correspond to the main design of LAPACK [5]. The computation is basically split into successive sequences composed by two phases: (1) the panel computation phase, mainly based on level 2 BLAS, in which the transformations are accumulated within a panel of the matrix and (2) the update of the trailing submatrix, in which the transformations from the panel phase are applied at once to the trailing submatrix in terms of level 3 BLAS operations. One of the bottlenecks with such approach is the creation of unnecessary synchronization points between the phases. Moreover, LAPACK extracts its performance for the most part from the parallel multithreaded BLAS. The parallel paradigm behind it is very similar to the fork-join model, which further exacerbates the issue related to artifactual synchronization points. And the design of the block algorithm for calculating the Cholesky-based matrix inversion is not an exception and falls into this category. The original matrix is first reduced using the Cholesky factorization $A = LL^T$ where L is a lower triangular square matrix with positive diagonal elements (i.e., *DPOTRF* routine call). The inverse of the Cholesky factor is then computed $T = L^{-1}$ (i.e., *DTRTRI* routine call). The last step calculates the actual inverse $A^{-1} = TT^T$ and stores it (in-place) in the lower side of the matrix (i.e., *DLAUUM* routine call). This three-steps block Cholesky-based matrix inversion is therefore very limited in terms of parallelism and cannot fully benefit from the available highly-parallel processing units.

The next Section recalls the concepts of tile algorithms and describes how it can be applied to the numerical algorithm studied in this paper.

IV. Cholesky-Based Matrix Inversion Using Tile Algorithm

The idea behind tile algorithm is to transform the original matrix data with column-major data layout into tile data

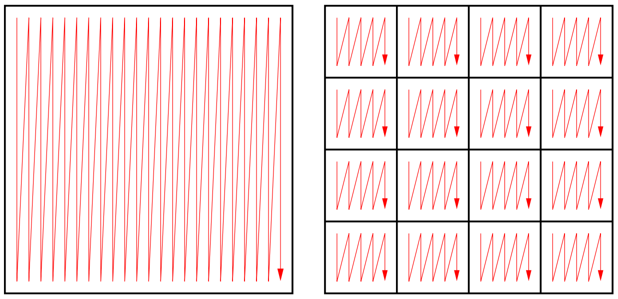
Figure 1. Translation from LAPACK Layout (column-major) to Tile Data Layout

layout, as seen in Figure 1. The parallelism becomes then exposed to the user thanks to the task fine granularity. Indeed, the matrix tiles can be seen as the fundamental unit of computations of the numerical algorithms. The tile Cholesky-based matrix inversion can be then written as in Algorithm 1. The rigid panel-update sequence, previously described in Section III, is now replaced by an out-of-order task execution flow, where numerical kernels from different stages operating on tiles can concurrently run. For example, the tasks involved in the third stage (DLAUUM) can start executing, while the first stage i.e., Cholesky factorization (DPOTRF), is still being processed. Therefore, the strong and artifactual synchronizations points, seen in block algorithms, are completely removed using tile algorithms not only within a single stage but also in-between stages of the Cholesky-based matrix inversion algorithm. This has already been studied in [7] in the context of homogeneous multicore architectures. The authors leverage this study to tackle challenging and complex heterogeneous platforms using hardware accelerators.

The next Section describes the implementations of the new high performance numerical kernels required for GPU computing.

V. High Performance Kernel Descriptions

Most of the functions involved in the matrix inversion using Cholesky inversion are straight calls to level 3 BLAS routines i.e., DSYRK, DTRMM, DTRSM and DGEMM. Therefore, by linking the application with the CUBLAS library (nVIDIA BLAS package), those functions can be substantially accelerated and ported on the GPU architecture. However, there are three other routines i.e., DPOTRF, DTRTRI and DLAUUM corresponding to the standard LAPACK computational drivers, which require GPU support. While the GPU version of the DPOTRF kernel has already been developed and distributed in MAGMA [6], DTRTRI and DLAUUM are not included yet in this numerical library and have been newly implemented to lead this proposed research work. Following the work done for the GPU DPOTRF kernel, the GPU versions of DTRTRI and DLAUUM kernels use the hybridization methodology, where both, CPU and GPU, work hand in hand. The inefficient and not parallel portion of the numerical kernels (DTRTI2 and DLAUU2, respectively) are handled by the CPU, while the high compute-intensive operations are off-loaded to the GPU. Those two routines will be eventually integrated into the next software release of MAGMA.

Once the GPU versions of all numerical kernels are available, a dynamic heterogeneous runtime environment system is necessary to schedule in a load balanced manner

Algorithm 1 Tile sequential in-place Cholesky-based matrix inversion (lower case). A is an $NT \times NT$ tile matrix.

```
1:  {Stage 1: Cholesky factorization}
2:  for $k$ = 0 to NT−1 do
3:    DPOTRF($A_{k,k}$)
4:    for $m$ = k+1 to NT−1 do
5:      DTRSM($A_{k,k}$, $A^T_{m,k}$)
6:    end for
7:    for $m$ = k+1 to NT−1 do
8:      DSYRK($A_{m,k}$, $A_{k,k}$)
9:      for $n$ = k+1 to m−1 do
10:       DGEMM($A_{m,k}$, $A^T_{n,k}$, $A_{m,n}$)
11:     end for
12:   end for
13: end for
14: {Stage 2: Calculate $L^{-1}$}
15: for $n$ = 0 to NT−1 do
16:   for $m$ = n+1 to NT−1 do
17:     DTRSM($A_{n,n}$, $A_{m,n}$)
18:   end for
19:   for $m$ = n+1 to NT−1 do
20:     for $k$ = 0 to n−1 do
21:       DGEMM($A_{m,n}$, $A_{n,k}$, $A_{m,k}$)
22:     end for
23:   end for
24:   for $m$ = 0 to n−1 do
25:     DTRSM($A_{n,n}$, $A_{m,n}$)
26:   end for
27:   DTRTRI($A_{n,n}$)
28: end for
29: {Stage 3: Compute $A^{-1} = L^{-T} \times L^{-1}$}
30: for $m$ = 0 to NT−1 do
31:   for $n$ = 0 to m−1 do
32:     DSYRK($A_{m,n}$, $A_{n,n}$)
33:     for $k$ = n+1 to m−1 do
34:       DGEMM($A^T_{m,k}$, $A_{m,n}$, $A_{k,n}$)
35:     end for
36:   end for
37:   for $n$ = 0 to m−1 do
38:     DTRMM($A_{m,m}$, $A^T_{m,n}$)
39:   end for
40:   DLAUUM($A_{m,m}$)
41: end for
```

the various tasks across the processing units (CPU and/or GPU), which is the topic of the next Section.

Algorithm 2 Tile Hybrid CPU-GPU in-place Cholesky-based matrix inversion (lower case) using dynamic heterogeneous scheduler StarPU. A is an $NT \times NT$ tile matrix.

```
{Stage 1: Cholesky factorization}
for $k$ = 0 to NT−1 do
    Starpu_Insert_Task(DPOTRF($A_{k,k}$))
    for $m$ = k+1 to NT−1 do
        Starpu_Insert_Task(DTRSM($A_{k,k}$, $A^T_{m,k}$))
    end for
    for $m$ = k+1 to NT−1 do
        Starpu_Insert_Task(DSYRK($A_{m,k}$, $A_{k,k}$))
        for $n$ = k+1 to m−1 do
            Starpu_Insert_Task(DGEMM($A_{m,k}$, $A^T_{n,k}$, $A_{m,n}$))
        end for
    end for
end for
{Stage 2: Calculate $L^{-1}$}
for $n$ = 0 to NT−1 do
    for $m$ = n+1 to NT−1 do
        Starpu_Insert_Task(DTRSM($A_{n,n}$, $A_{m,n}$))
    end for
    for $m$ = n+1 to NT−1 do
        for $k$ = 0 to n−1 do
            Starpu_Insert_Task(DGEMM($A_{m,n}$, $A_{n,k}$, $A_{m,k}$))
        end for
    end for
    for $m$ = 0 to n−1 do
        Starpu_Insert_Task(DTRSM($A_{n,n}$, $A_{m,n}$))
    end for
    Starpu_Insert_Task(DTRTRI($A_{n,n}$))
end for
{Stage 3: Compute $A^{-1} = L^{-T} \times L^{-1}$}
for $m$ = 0 to NT−1 do
    for $n$ = 0 to m−1 do
        Starpu_Insert_Task(DSYRK($A_{m,n}$, $A_{n,n}$))
        for $k$ = n+1 to m−1 do
            Starpu_Insert_Task(DGEMM($A^T_{m,k}$, $A_{m,n}$, $A_{k,n}$))
        end for
    end for
    for $n$ = 0 to m−1 do
        Starpu_Insert_Task(DTRMM($A_{m,m}$, $A^T_{m,n}$))
    end for
    Starpu_Insert_Task(DLAUUM($A_{m,m}$))
end for
```

VI. The Dynamic Heterogeneous Runtime StarPU

StarPU [4] is a runtime system that dynamically schedules tasks on accelerator-based platforms, such as multicore architectures enhanced by GPUs accelerators. StarPU ensures data availability and coherency between the memories of different units, similar to the policy of the virtual shared memory subsystem. This allows the programmer to focus on what to do (e.g., choosing a scheduling strategy) while the runtime system takes care of how to do it efficiently (e.g., ensuring data transfers and coherency). The codelet is one of the fundamental data structures of StarPU. It defines a multi-version of the computational kernel that should be implemented for each device e.g., CPU core and/or GPU (see Figure 2). Applying a codelet on a data set means executing a StarPU task, these tasks are asynchronous and the StarPU runtime keeps track and appropriately handles any data dependencies between them. Moreover, StarPU is able to extract parallelism from a sequential nested loop program. By accordingly linking the sequential application with StarPU and calling the *Starpu_Insert_Task* API, its runtime will run the different tasks on the available heterogeneous processing units. This is a huge gain in terms of user productivity, going from sequential nested loops to parallel heterogeneous out-of-order scheduling. Algorithm 2 represents the new parallel version of the sequential Cholesky-based matrix inversion described in Algorithm 1. The *Starpu_Insert_Task* API is actually a wrapper to the user-defined codelet. Depending on the availability of the GPU devices, StarPU may decide to run the task on the CPU whenever possible (and vice versa), which permit at the same time to exploit all available resources of the heterogeneous system and reducing by the same token the obvious load imbalance between CPU and GPU computational power.

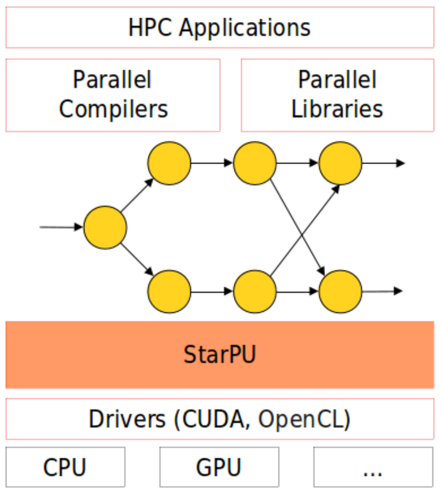

Figure 2. The StarPU framework for dynamic load-balanced scheduling on heterogeneous systems.

The next Section presents the preliminary results of our new implementation on heterogeneous GPU-based system.

VII. Experimental Results

This Section highlights the preliminary results of our new implementation of the Cholesky-based matrix inversion using StarPU. All the experiments have been conducted on a hybrid GPU-based system of 8 Intel Xeon X5550 CPU cores running at 2.67GHz with 24 GB of memory and enhanced with two nVIDIA Fermi C2070 GPUs (Fermi-based, 448 CUDA cores each) with 6 GB of memory. The application has been linked against nVIDIA CUDA 4.0 library to access the CUBLAS routines. Figure 3 shows the performance comparison in Gflop/s of our heterogeneous version against the same routines distributed in MAGMA V1.0 (single CPU-GPU support only), in PLASMA V2.4 using tile algorithms on the 8 available cores (homogeneous x86 multicore architecture support only) and in LAPACK V3.2 linked with multithreaded Intel MKL BLAS V10.2 using also the 8 available cores. Our high performance implementation achieves almost half a Tflop/s (448 Gflop/s), which corresponds to 5 and 6-fold improvement compared to the equivalent routines from MAGMA and PLASMA, respectively, and 10-fold improvement compared to LAPACK. Furthermore, compared to MAGMA, our implementation is not memory-limited and can scale beyond the actual memory available in the GPU devices, thanks to the cooperative computing work done with the available x86 cores.

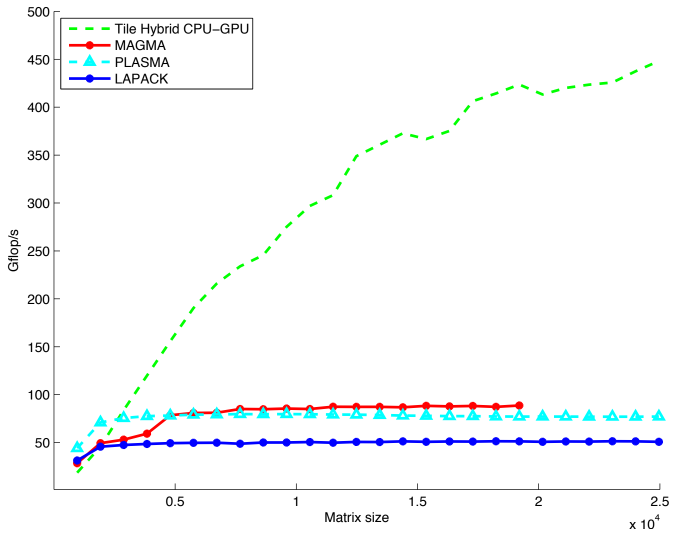

Figure 3. Cholesky-based matrix inversion (DPOTRF + DTRTRI + DLAUUM) on eight Intel Xeon X5550 @ 2.67GHz and two nVIDIA Tesla C2070 GPU cards.

VIII. Conclusion and Future Work

This paper presents preliminary results of our Cholesky-based matrix inversion on x86 architecture enhanced with GPU accelerators using the heterogeneous dynamic runtime system StarPU. Our implementation (448 Gflop/s) results in up to 5 and 6-fold improvement compared to the equivalent routines from MAGMA V1.0 and PLASMA V2.4, respectively, and 10-fold improvement compared to LAPACK V3.2 linked with multithreaded Intel MKL BLAS V10.2, with a matrix size of 24960$\times$ 24960. There are many new directions to deeper study this algorithm in the context of StarPU. The authors plan (1) to expand the analysis of the actual directed acyclic graph of this three-stage algorithm, (2) to generate the execution traces, (3) to evaluate the application scalability on a cutting-edge NUMA system composed of tens of cores with multiple GPU accelerators and (4) to study the impact of GPU Direct communication brought by CUDA 4.0.

IX. Acknowledgement

The authors would like to thank nVIDIA for their generous graphics hardware donations used in these experiments.

References

[1] J. Dongarra, P. Beckman, and et al., "The International Exascale Software Roadmap." *International Journal of High Performance Computer Applications*, vol. 25, no. 1, pp. 1–82, 2011.

[2] N. J. Higham, *Accuracy and Stability of Numerical Algorithms, Second Edition.* Society for Industrial and Applied Mathematics, 2002.

[3] *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.4*, University of Tennessee at Knoxville, May 2011.

[4] *StarPU Users' Guide, A Unified Runtime System for Heterogeneous Multicore Architectures Version 0.9.2*, INRIA Bordeaux, France, September 2011.

[5] E. Anderson, Z. Bai, C. Bischof, S. L. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen, *LAPACK User's Guide*, 3rd ed. Philadelphia: Society for Industrial and Applied Mathematics, 1999.

[6] *MAGMA Users' Guide, Matrix Algebra on GPU and Multicore Architectures V1.0*, Technical report, University of Tennessee at Knoxville, ICL, 2011.

[7] E. Agullo, H. Bouwmeester, J. Dongarra, J. Kurzak, J. Langou, and L. Rosenberg, "Towards an Efficient Tile Matrix Inversion of Symmetric Positive Definite Matrices on Multicore Architectures." in *VECPAR'10*, 2010, pp. 129–138.

[8] *The FLAME Project, Formal Linear Algebra Methods Environment*, University of Texas at Austin, April 2010, http://z.cs.utexas.edu/wiki/flame.wiki/FrontPage.

[9] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 123–132.

[10] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "Faster, Cheaper, Better a Hybridization Methodology to Develop Linear Algebra Software for GPUs," in *GPU Gems, Chapter 34*, 2010, pp. 473–484.