High-performance Cholesky factorization for GPU-only execution

Azzam Haidar   Ahmad Abdelfatah
Stanimire Tomov
University of Tennessee, U.S.A.
haidar,ahmad,tomov@icl.utk.edu

Jack Dongarra
University of Tennessee, U.S.A.
Oak Ridge National Laboratory, U.S.A.
University of Manchester, UK
dongarra@icl.utk.edu

Abstract

We present our performance analysis, algorithm designs, and the optimizations needed for the development of high-performance GPU-only algorithms, and in particular, for the dense Cholesky factorization. In contrast to currently promoted designs that solve parallelism challenges on multicore architectures by representing algorithms as Directed Acyclic Graphs (DAGs), where nodes are tasks of fine granularity and edges are the dependencies between the tasks, our designs explicitly target manycore architectures like GPUs and feature coarse granularity tasks (that can be hierarchically split into fine grain data-parallel subtasks). Furthermore, in contrast to hybrid algorithms that schedule difficult to parallelize tasks on CPUs, we develop highly-efficient code for entirely GPU execution. GPU-only codes remove the expensive CPU-to-GPU communications and the tuning challenges related to slow CPU and/or low CPU-to-GPU bandwidth. We show that on latest GPUs, like the P100, this becomes so important that the GPU-only code even outperforms the hybrid MAGMA algorithms when the CPU tasks and communications can not be entirely overlapped with GPU computations. We achieve up to $4,300$ GFlop/s in double precision on a P100 GPU, which is about 7-8× faster than high-end multicore CPUs, e.g., two 10-cores Intel Xeon E5-2650 v3 Haswell CPUs, where MKL runs up to about 500-600 Gflop/s. The new algorithm also outperforms significantly the GPU-only implementation currently available in the NVIDIA cuSOLVER library.

Categories and Subject Descriptors G.1.3 [*Numerical Linear Algebra*]: Linear systems (direct and iterative methods)

General Terms Algorithms, Experimentation, Measurement, Performance

Keywords factorization; numerical linear algebra; hardware accelerators; numerical software libraries; one-sided factorization algorithms

1. Introduction

The scientific high performance computing community has faced dramatic hardware changes since the emergence of multicore architectures. Multicore architectures are now ubiquitous – not only in the fastest high performance computers in the world, as ranked in the Top500 list (Strohmaier et al. 1993-2016), but also even in small portable devices like smart phones and watches. Moreover, the number of cores on the chip continues to grow, with architectures containing 10s of independent cores or more (referred to as manycore) becoming common. Latest examples include the new Intel Knights Landing (KNL) Xeon Phi processor with up to 72 cores, and the new manycore P100 GPU accelerator from NVIDIA, featuring 56 multi-processors (MP) with 64 CUDA cores each. This presents the scientific software community with both a daunting challenge and a unique opportunity. The challenge arises from the disturbing mismatch between the design of systems based on this new chip architecture – many cores with reduced bandwidth and memory available per core – and the components of the traditional software stack, such as numerical libraries, on which scientific applications have relied for their accuracy and performance. The state of the art, high performance dense linear algebra software libraries, (i.e., LAPACK (Anderson et al. 1992)) have shown limitations on multicore architectures (Agullo et al. 2009). The performance of LAPACK relies on the use of a standard set of Level-3 Basic Linear Algebra Subprograms (BLAS) (Dongarra et al. 1990) within which nearly all of the parallelism occurs following the expensive fork-join paradigm, making it prudent to revisit and/or redesign existing numerical linear algebra algorithms to be better suited for such hardware.

The PLASMA library (Parallel Linear Algebra for Scalable Multi-core Architectures) (Inn 2010) tackles this challenge for multicore architectures by designing and using tile algorithms to achieve high performance. These tile algorithms can then be represented by Directed Acyclic Graphs (DAGs), where nodes are tasks of fine granularity and edges are the data dependencies between the tasks. Then, a runtime environment can be used to efficiently schedule the DAG across the multicore platform. Using this methodology, PLASMA provides very efficient algorithms for multicore architectures because the scheduling mechanism provides asynchronous execution of the fine granularity tasks that can remove the expensive synchronizations associated with fork-join between large tasks (BLAS done in parallel). There are, however, overheads of scheduling many fine granularity tasks, and on manycore architectures like current GPUs and Xeon Phi, hybrid algorithms as in MAGMA (Tomov et al. 2010; Haidar et al. 2015b; Cao et al. 2013) have been more advantageous by keeping top level tasks of coarse granularity, that are, however, split hierarchically into fine grain data-parallel subtasks (through parallel BLAS implementations). However, MAGMA schedules the difficult-to-parallelize tasks on CPUs, and thus is not directly applicable for GPU-only execution.

The objective of this paper is to revisit the current state-of-the-art algorithms designed originally for multicore and heterogeneous architectures (as in the PLASMA and MAGMA libraries (Agullo et al. 2009) and redesign them for GPU-only execution. We present our performance analysis and algorithm designs, and the optimizations needed to achieve this goal of providing high-performance GPU-only algorithms, and in particular, the dense Cholesky factorization. GPU-only codes are of high interest because they remove the expensive CPU-to-GPU communications and the tuning challenges related to slow CPU and/or low CPU-to-GPU bandwidth. Indeed, we show that on the latest GPUs, like the P100, this becomes so important that the GPU-only code even outperforms the hybrid MAGMA algorithms when the CPU tasks and communications can not be entirely overlapped with GPU computations.

2. Related Work

The development of GPU-only dense linear algebra algorithms was avoided in the past because:

- The implementation and optimization of difficult-to-parallelize parts of the computation could be evaded through the use of hybrid algorithms, and
- Hybrid algorithms were faster.

However, recent need in many applications for many independent linear algebra problems of small sizes motivated the development of the so-called batched linear algebra algorithms (Haidar et al.; Dongarra et al. 2016). Batched LU, QR, and Cholesky were developed for both fixed matrix sizes (Dong et al. 2014a; Haidar et al. 2015a; Dong et al. 2014b) and variable sizes (Abdelfattah et al. 2016b,a) that are GPU-only. The reason for developing them for GPUs only is that the sizes were so small that there was not enough computation for the GPU work to overlap the expensive CPU-to-GPU communications. Regardless of the motivation, since they were developed, it was possible to easily extend them to compute single large factorizations for GPU-only execution (Abdelfattah et al. 2016a; Haidar et al. 2015c). Rather than these early implementations that resulted from highly-optimized batched factorizations for small problems, in this paper we concentrate on and study in detail specifically GPU-only algorithms. In turn, the algorithm designs and optimizations developed here, outperform significantly the early results, including the implementations that were subsequently made available through the cuSOLVER library from NVIDIA (NVIDIA Corporation).

Besides extending ideas from the batched linear algebra routines, manycore algorithms can also be built on ideas from the hybrid linear algebra algorithms. This was demonstrated for the case of KNL processors in (Haidar et al. 2016). The difficult-to-parallelize tasks are the panel factorizations (see Section 3), and these are the tasks offloaded for execution to the CPUs in the hybrid algorithms. As the KNL is self-hosted (i.e., there is no additional CPU host), a virtual CPU abstraction was created from a subset of the KNL cores that enabled hybrid algorithms to run efficiently on homogeneous manycore processors (Haidar et al. 2016). The panel factorizations can be done in parallel with the trailing matrix updates in factorizations like QR, LU, and Cholesky (see Section 3), which is used in the hybrid algorithms to overlap CPU work and CPU-to-GPU communications with GPU work on the trailing matrix updates. We will see that similar techniques can be developed for GPU-only execution, where some of GPU's MPs will perform the compute-intensive matrix update, while others (possibly the same) will do the panel factorization through different GPU streams.

3. Background

In this section, we review the paradigm behind the state-of-the-art numerical software, namely the LAPACK library for shared-memory. In particular, we focus on the Cholesky factorization which is one of the three widely used one-sided factorizations (QR, LU and Cholesky) in the scientific community. These factorizations are the main components of solving numerical linear systems of equations.

The Cholesky factorization (or Cholesky decomposition) is mainly used as a first step for the numerical solution of the linear system of equations $Ax = b$, where A is a symmetric and positive definite matrix. Such systems often arise in physics applications, where A is positive definite due to the nature of the modeled physical phenomenon. 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. Due to the symmetry, the matrix can be factorized either as an upper triangular matrix or as a lower triangular matrix. In LAPACK, the double precision algorithm is implemented by the DPOTRF routine. We note that the reference number of operations for the Cholesky factorization is known to be $\mathcal{O}(\frac{n^3}{3})$, for that we used this formula to produce all the Gflop/s mentioned on the figures which reflect the total elapsed time and thus the higher the flops the faster the routine is.

3.1 Description and Concept

The LAPACK library provides a broad set of linear algebra operations aimed at achieving high performance on systems equipped with memory hierarchies. The factorisation algorithms implemented in LAPACK leverage the idea of blocking to limit the amount of bus traffic in favor of a high data reuse. LAPACK consists of a sequential algorithm that relies on parallel building blocks (i.e., the BLAS with its Level-1, 2, and 3 types of operations) in order to exploit parallelism. Most of these algorithms can be described as the repetition of two fundamental phases as shown in Figure 1:

- *Panel factorization*: Depending on the linear algebra operation that must be performed, a number of transformations are computed for a small portion of the matrix – *the panel* – marked by the light and dark blue portion of Figure 1;
- *Trailing submatrix update*: In this step, all the transformations that have been computed during the panel factorization step must be applied to the rest of the matrix – *the trailing submatrix* – marked by the green and magenta portion of Figure 1. This is done by means of Level-3 BLAS operations.

	Cholesky	QR	LU
PanelFactorize	xPOTF2 xTRSM	xGEQF2	xGETF2
TrailingMatrixUpdate	xSYRK xGEMM	xLARFB	xLASWP xTRSM xGEMM

Table 1: Routines for panel factorization and the trailing matrix update.

This design as a two-phase process is typical for the blocked algorithms in LAPACK. It consists of organizing the linear algebra algorithm in such a way that only a small part of the computation is done in the panel phase, while most is done in the update phase. The panel factorization can be identified as a sequential execution task that represents a small fraction of the total number of FLOPS

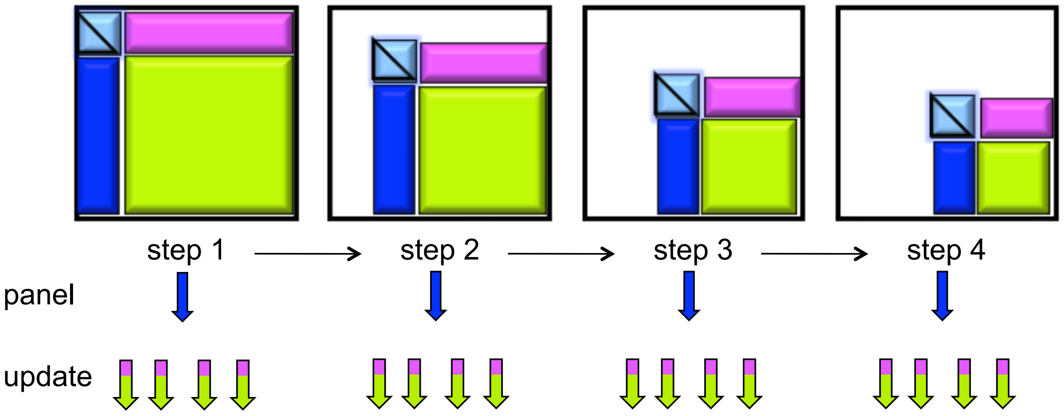
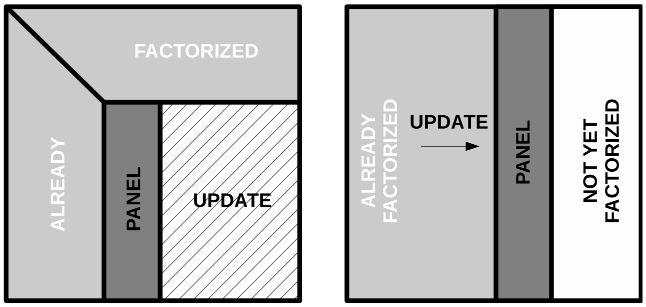


Figure 1: Description and concepts of the LAPACK algorithms.

– only $\theta(n^2)$ are in the panel vs. a total of $\theta(n^3)$ FLOPS. It is referenced as a sequence of memory-bound operations and cannot be parallelized easily. For the Cholesky factorization, this applies to the potf2 routine, while the trsm is a Level 3 BLAS routine that can exhibit parallelism. Parallelism in the Cholesky factorization is exploited at the Level 3 BLAS routines, which are mainly used in the trailing matrix update phase. Most of the flops are computed in this phase, and for an optimal, well-designed implementation, the performance of the factorization should behave similar to the performance of the Level 3 BLAS routine that the trailing matrix update uses. This methodology implies a *"fork-join"* parallel model (as shown in Figure 1) since the execution flow of the matrix factorization represents a sequence of sequential operations (the panel factorizations) interleaved with parallel ones; namely, the updates of the trailing submatrices. For the sake of completeness, we present in Table 1 the BLAS routines that should be substituted for each of the phases for the three LAPACK factorizations.

3.2 Implementation Design Variants

Several algorithmic variants exist for the one-sided factorizations described above. The two main ones are called *Left Looking* (LL) and *Right Looking* (RL). They only differ on the location of the update applications with regards to the panel. At each step, the RL variant computes the transformations on the current panel, then it applies these transformations to the trailing submatrix to the right of the panel (called *updates*). For example, in Fig. 2a, the light-gray area represents the portion of the matrix that has already been factorized. The dark gray area corresponds to the panel that is currently being factorized. On the right side of the current panel, the dashed area specifies the location of the update portion, after the current panel has been factorized. For the RL variant, the data located in this area is actually transient and is constantly updated until the end of the whole factorization. Algorithm 2 shows the implementation of the RL variant of the Cholesky factorization. In contrast, Fig. 2b shows the LL variant (also called the ”lazy” variant), where the current panel is first updated by applying all the previous transformations coming from the previous panels (from the left), and then is factorized. Thus, the updates are not applied to the entire trailing matrix as in the RL variant but are limited only to the current panel. The matrix is thus completely factorized one panel at a time. Therefore, the LL variant limits the number of memory accesses (e.g., panel writes, if the panel is kept in cache until fully updated) while increasing the reuse of the data located on the panel. The LL variant is known to be cache friendly, but decreases the parallelism, as the subsequent updates of the remaining matrix columns are delayed and will be eventually applied as the panel computations move forward. Algorithm 1 presents the implementation of the LL variant of the Cholesky factorization. Note that the QR and LU factorization will follow the same sequence but with calls to other BLAS routines. We refer the reader to Table 1 for the name of the BLAS routines for the QR and the LU factorization.



(a) RL variant.

(b) LL variant.

Figure 2: One-sided factorization-looking variants.

Algorithm 1 LL Cholesky

```
for $i = 0, nb$ to N do
    if (i > 0) then
        {Update current panel $\mathbf{A_{i:m,i:i+nb}}$}
        DSYRK:
        $A_{i:i+nb,i:i+nb} = A_{i:i+nb,i:i+nb} - A_{i:i+nb,0:i} \times A^T_{i:i+nb,0:i}$
        DGEMM:
        $A_{i+nb:m,i:i+nb} = A_{i+nb:m,i:i+nb} - A_{i+nb:m,0:i} \times A^T_{i:i+nb,0:i}$
    end if
    {Panel factorize $\mathbf{A_{i:m,i:i+nb}}$}
    DPOTF2 $A_{i:i+nb,i:i+nb}$
    DTRSM $A_{i+nb:m,i:i+nb} = A_{i+nb:m,i:i+nb} \times A^{-1}_{i:i+nb,i:i+nb}$
end for
```

4. Methodology and Algorithmic Advancements

The state-of-the-art methodology for server-class accelerated systems is based on hybrid algorithms that use both the CPU and GPU hardware components (Tomov et al. 2010; Agullo et al. 2010;

Algorithm 2 Right looking Cholesky

```
for $i = 0, nb$ to N do
    {Panel factorize $\mathbf{A_{i:m,i:i+nb}}$}
    DPOTF2 $A_{i:i+nb,i:i+nb}$
    DTRSM $A_{i+nb:m,i:i+nb} = A_{i+nb:m,i:i+nb} \times A^{-1}_{i:i+nb,i:i+nb}$
    {Update trailing matrix $\mathbf{A_{i+nb:m,i+nb:m}}$}
    DSYRK: $A_{i+nb:m,i+nb:m} = A_{i+nb:m,i+nb:m} - A_{i+nb:m,i:i+nb} \times A^{T}_{i+nb:m,i:i+nb}$
end for
```

Yamazaki et al. 2012; Dongarra et al. 2014; Haidar et al. 2014). Benchmark software also uses hybridized methods (Fatica 2009). Typically, small or memory bound tasks on the critical path of the algorithm are assigned to the CPUs (e.g., panel phase), and large data-parallel tasks to the GPUs (update phase). This is what we denote as the hybrid approach. While this methodology works very well, it can have significant drawbacks when the balance between the processor and the accelerator is skewed. A slow CPU for example, even after tuning, can make a fast GPU idle. Moreover, this can be further aggravated by the slow CPU-to-GPU communication. Also, from an energy point of view, a hybrid approach consumes power on both the CPU and the GPU since both hardware are computing. Thus, since the power efficiency rate of flops/Watt for the CPU is typically too low compared to the one for the GPU, one can expect a degradation in the energy efficiency of the hybrid algorithms. These reasons further motivate the need for an additional schema that uses only the GPU to perform both the memory bound and the compute intensive tasks; that is, a variant that uses only the GPU to perform the whole computation. When the GPU only is used the CPU is idle and thus its power is too low compared to the hybrid mode when it is fully loaded. We use a careful study and analysis of the algorithm to guide our optimizations for the memory bound operations and to provide a GPU-only implementation that is very competitive with the hybrid one in term of performance, and definitely way ahead in terms of energy efficiency.

We have described above that high-performance linear algebra algorithms can be designed so that their computations use building block, e.g., BLAS. This is important since the use of BLAS has been crucial for the high-performance sustainability of major numerical libraries for decades, and therefore we can also leverage the lessons learned from that success. However, to enable the effective use of a building block-based approach, there is a need to develop highly efficient and optimized kernels.

Below we describe our studies and the methodology toward achieving high-performance GPU algorithms. We recognize three possible paths that can help boosting the performance of any algorithm or application. First is the algorithmic path, then the kernel optimization path, and finally the implementation design path.

4.1 Performance Analysis based on Algorithmic Design

Here we study the two designs of the Cholesky algorithm. As seen above in Section 3.2, the panel factorization of both the left and right looking algorithms is the same. It consists of calls to potf2 and trsm. The main difference between the two designs (left or right looking) is the update phase. Note that, as mentioned above, the performance of the one-sided factorization (Cholesky, LU, or QR) is driven by the performance of its update phase. The update phase of the left looking design enforces locality, and most importantly, it uses the gemm routine for its operations, while the right looking variant exhibits more parallelism and uses the syrk routine for its operations. Historically, it is well known that the left looking update phase does not exhibit as much parallelism as the right looking one. This raises the question of whether the right looking should be considered as the only suitable approach for multicore CPUs or GPUs.

To answer this, we first point out that this description is not perfectly accurate, as it requires an explanation of the meaning that one design may be more suitable or may exhibit more parallelism than another. To do this, we describe first some of our analysis and then show performance results for both designs. For a multicore CPU, the parallelism is at the core/thread level, and thus parameters such as L2 cache sizes, use of SIMD/AVX instructions, NUMA node effect, etc., will have to be considered. For a GPU, the parallelism is at the thread-block and at the SMX level, and thus a large number of working thread-blocks is always preferable.

Since the performance should be driven by the update phase, we studied the operations involved in this phase to find out how parallelism can be extracted. First of all, the parallelism depends on the shape of the matrices involved, as well as the size and type of the operation involved in the update phase. Figure 3 shows the matrix shapes of both the gemm and the syrk operations, which represent the left and right looking variants, respectively. It is true that the syrk can exhibit more parallelism, but, we can also extract parallelism from the gemm shape when *nb* is "acceptable" (where the acceptable *nb* is hardware and software dependent – it depends on the implementation of the gemm and the syrk routines, the caches size, and other GPU/CPU hardware features; we will show that *nb*=128, 256, 512 are good choices). Moreover, the performance also depends on the implementation of these two routines. Due to the higher need for gemm and since other Level-3 BLAS can be derived from gemm, most BLAS libraries – such as cuBLAS, MAGMA, MKL, GOTO, and ATLAS – are optimized for their gemm routine for all precisions and shapes, before the other Level-3 BLAS routines, (e.g., their syrk routine). Figures 4 and 5 show the performance of the gemm and the syrk routines in both single and double precision, for the shapes required by the Cholesky update phase for a large value of $n = 10,000$ and for different values of *nb* ranging from 32 to $1,024$ on two different GPU architectures (the Nvidia K40c and the P100). The performance shown in these figures establishes the performance upper bound for the Cholesky factorization.

From these figures one can conclude that:

- First, in order to reach the upper bound of the update routine on GPUs – either gemm or syrk– the *nb* should be large. The minimal *nb* where the gemm or syrk reaches good performance is about $nb = 256$ or $nb = 512$ in double precision, and $nb = 512$ or $nb = 1,024$ in single precision for both architectures;
- Second, we note that when designing a GPU library, before struggling on optimizing kernels or overall implementation, one should study and understand the performance roofline bound of the algorithm;
- Third, one can also notice that these Level 3 BLAS routines are optimized by the vendor, and sometimes the vendor focuses their optimization for one specific shape, or one precision, or just one of the routines.

Since the single precision showed larger difference between the two update routines, we picked up a $nb = 512$ and showed in Figure 6 and 7 the performance of the update phase of the left and the right looking Cholesky factorization step by step during the process for a matrix of size $n = 20480$. In other words, timing the performance of the update phase from inside the Cholesky code for a $nb = 512$ in single precision on both the Nvidia K40c and P100 GPU. This experiments is representative since it illustrates the performance through the factorization steps where also we can easily figure out the expected performance for any matrix size. For

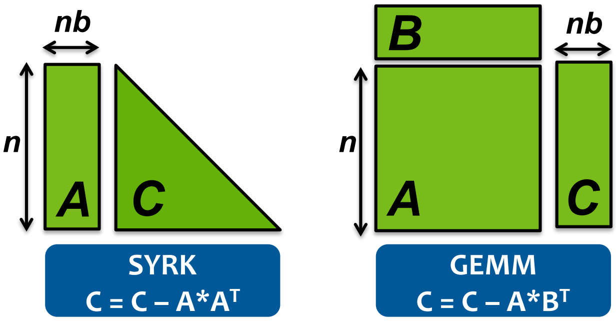


Figure 3: The shape of the update operation for both Left and Right looking design.

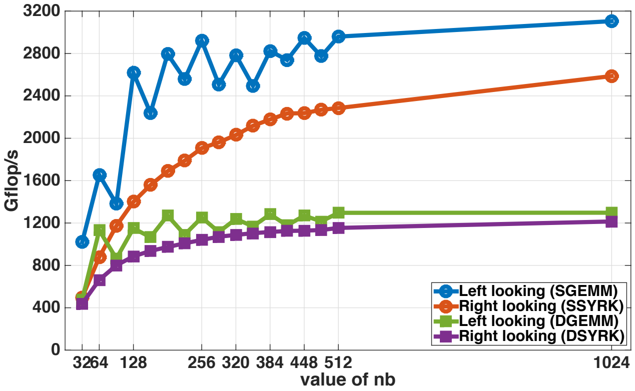


Figure 4: The performance of the gemm and syrk routine for the shape required by the update of either the left or right looking design when varying the size of *nb* in both single and double precision on a Nvidia K40c GPU.

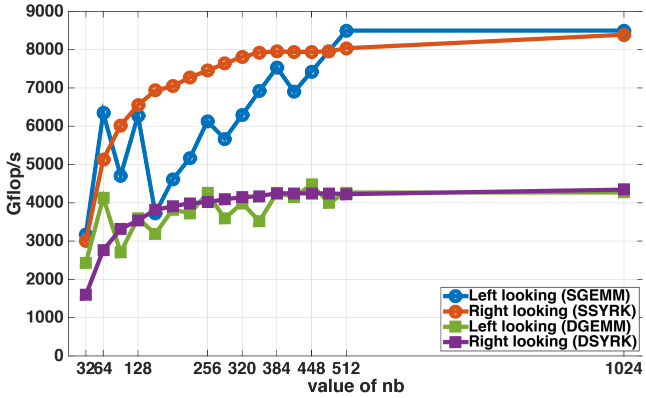


Figure 5: The performance of the gemm and syrk routine for the shape required by the update of either the left or right looking design when varying the size of *nb* in both single and double precision on a Nvidia P100 GPU.

example we can expect that the left looking variant is always faster then the right looking one on single precision on K40c GPU.

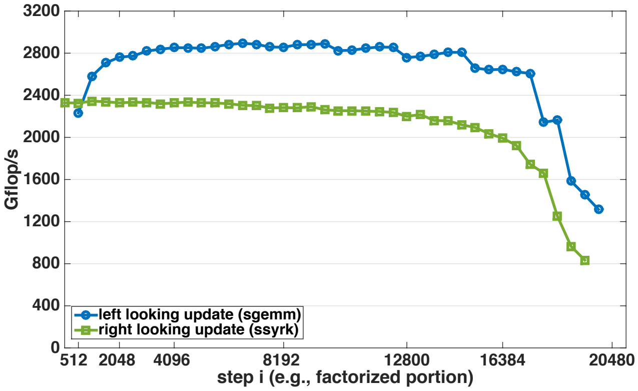


Figure 6: The performance of the gemm and syrk routine for the shape required by the update of either the left or right looking design when varying the size of *nb* in both single and double precision on a Nvidia K40c GPU.

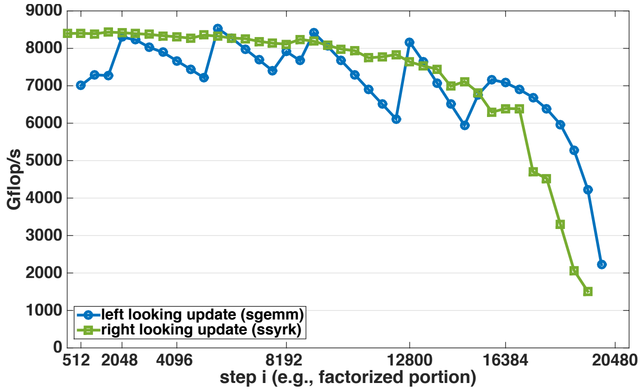


Figure 7: The performance of the gemm and syrk routine for the shape required by the update of either the left or right looking design when varying the size of *nb* in both single and double precision on a Nvidia P100 GPU.

4.2 Performance Analysis based on Kernel Optimization

After we studied the performance of the update phase and converged on the possible choices of *nb* that provide good performance, we could concentrate on optimizing the potf2 kernel for the possible *nb* sizes. The potf2 performance requirement is to have the cost of the panel phase small; otherwise, the performance model will not be driven by the update phase. Indeed, slow potf2 can result in a dramatic slowdown since the potf2 is a memory bound operation, which even when highly optimized is still about 100 times less than a Level 3 BLAS operation (on current GPUs). For hybrid algorithms which are not the focus of this paper, but are mentioned for completeness and comparison, the potf2 routine is performed on the CPU, which is overlapped with GPU work on the update phase (usually relying on optimized potf2 from vendors; e.g., the MKL from Intel). Thus, the panel cost is hidden, which means that the hybrid approach can reach the peak of the level 3 BLAS update routine (syrk or gemm), provided that the updates fully overlap the CPU work and the CPU-to-GPU communications. We note that in the case where the CPU and/or the CPU-to-GPU connection is very slow as compared with the GPU, the potf2 might not always be overlapped. For the GPU-only approach, the performance of the potf2 routine can dramatically affect the performance.

The potf2 routine operates on a square submatrix of size *nb*, which is the algorithm step size, and consists of three types of operations – dot products, matrix-vector products, and scaling of vectors. If the submatrix cannot fit in fast memory, potf2 is a memory bound kernel since it is an unblocked routine. This means that a square matrix of size *nb* is factorized column by column in an unblocked fashion. Since the kernel consists of a sequential process (e.g., the column $i+1$ cannot be factorized before finishing the factorization of column i), it involves a lot of synchronizations. Since *nb* is small (less or equal to $1,024$), the potf2 factorization usually needs only one thread-block on the GPU, and thus the other GPU resources might be unused. If this challenge is not addressed, the inherently slow potf2 (even when it is optimal), can become orders of magnitude slower, e.g., considering that there are 56 MP on a P100 and only one MP may end up being used.

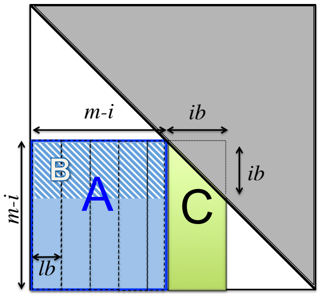


Figure 8: left-looking Cholesky factorization

Algorithm 3 The fused potf2 kernel.

```
1:  for $i = 0$, $ib$ to $m = nb$ do
2:      rAk ← $A_{(i:\mathrm{m},0:\mathrm{lb})}$; rC ← 0
3:      for $k = 0$, $lb$ to $m - i$ do
4:          rAkk ← rAk
5:          sB ← $\mathrm{rAk}_{(i:\mathrm{lb},k:k+\mathrm{lb})}$ inplace transpose
6:          barrier()
7:          rA1 ← $A_{(i:\mathrm{m},k+\mathrm{lb}:k+2\mathrm{lb})}$ prefetching
8:          rC ← rC + rAkk×sB multiplying
9:          barrier()
10:     end for
11:     sC ← rA1 - rC
12:     factorize sC
13: end for
```

We first implemented this routine on GPUs and found that its performance is about 3 Gflop/s for double precision on both architectures. We then performed a detailed performance study based on the collection and analysis of machine counters. Counter readings were taken using performance tools (NVIDIA's CUPTI and PAPI CUDA component (Malony et al. 2011).

While previously the unblocked potf2 algorithm was implemented with outer loops going from 1 to *nb* running on the CPU, calling the computational kernels on the GPU, we discovered that fusing the operations of potf2 into one kernel call is needed for performance. Moreover, to increase data reuse, and hence minimize communications, we concluded that adding an internal layer of blocking for the potf2 is necessary. The purpose of the fusion optimization is to minimize the load/store to the main memory and increase the data reuse at the thread-block level. The inner blocking is needed because it allows the operation to be performed on an inner block of size $nb \times ib$, which in turn decreases the number of register/shared memory required relative to when the whole $nb \times nb$ is in register/shared memory, and thus it allows the factorization of matrix with large *nb* size (for example $nb > 64$ for double precision). The inner blocking can also provide a very good performance similar to if the whole data is in shared memory by implementing techniques such as double buffering and prefetching. The inner blocking layer can be viewed as the Cholesky algorithm described in Algorithm 2 or 1 but at a kernel level, where all the calls are within one kernel. We discovered that blocking at the kernel level should follow a left-looking Cholesky factorization, with a blocking size *ib*, which is known to minimize data writes (in this case from GPU shared memory to GPU main memory).

When the kernel's working data is small, the computation associated with it becomes memory bound. Thus, fusing the four kernels of one iteration of Algorithm 1 (into one GPU kernel), will minimize the memory traffic, increase the data reuse from shared memory, and reduce the overhead of launching multiple kernels. Using a left-looking Cholesky algorithm, the update writes the panel of step k of size m-i×ib in the fast shared memory instead in the main memory, and so the merged potf2 routine can reuse the panel from the shared memory. Note that *nb* and *ib* control the amount of the required shared memory; they are critical for the overall performance, and thus can be used to (auto)tune the implementation.

We developed an optimized and customized *fused kernel* that first performs the update (syrk and gemm operations), and keeps the updated panel in shared memory to be used by the factorization step. The cost of the left looking algorithm is dominated by the update step (syrk and gemm). The panel C, illustrated in Figure 8, is updated as $C = C - A \times B^T$. In order to decrease its cost, we implemented a double buffering scheme that performs the update in steps of lb, as described in Algorithm 3. We mention that we prefix the data array by "r" and "s" to specify register and shared memory, respectively. We prefetch data from A into register array rAk while a multiplication is being performed between register array rAkk and the array sB stored in shared memory. Since the matrix B is the shaded portion of A, our kernel avoids reading it from the global memory and transposes it in place to the shared memory sB. Once the update is finished, the factorization (potf2 and trsm) is performed as one operation on the panel C, held in shared memory.

In order to develop Algorithm 3, a first step is to decide whether the main loop (e.g., the loop over i at line 1 of Algorithm 3) is on the CPU or on the GPU (inside the kernel). In this context, we developed *loop-inclusive* and *loop-exclusive* kernels. The *loop-inclusive* kernel is launched once from the CPU side, meaning that the loop iteration over *ib* of Algorithm 3 are unrolled inside the kernel. The motivation behind the *loop-inclusive* approach is to maximize the reuse of data, not only in the computation of a single iteration but also among iterations. More important is that when it is as one kernel, one can think of overlapping it with computation that might happen on another stream. If the factorization consists of many kernels, we might not see the overlap with other computation, since once the first kernel launch finishes, the GPU scheduler might schedule another queued thread-block than the one of the factorization, resulting in a non overlapped execution. Since the panel factorization requires only one thread-block, this means that resources might be lost. The *loop-exclusive* kernel executes one iteration of Algorithm 3 at each launch. The amount of shared memory decreases per launch (shared memory configurations are based on $m - i$) starting from the same amount of shared memory as the *loop-inclusive* at the first launch. In the *loop-exclusive* kernel, we will have to re-load the previous panel from main memory.

4.3 Performance Optimization based on Kernel Tuning

The autotuning process of the developed kernels has one tuning parameter to consider (ib). Since the range of values for ib is intended to be small, we conducted a sweep of all possible values of ib up to 32. In general, we can define different best-performing values of ib with different GPUs. The autotuning experiment is offline and needs to be conducted once per GPU model/architecture. Figure 9 shows the tuning results for both the *loop-inclusive* and the *loop-exclusive* kernels for different values of ib and for different values of *m*.

As expected, we observe a relatively low performance for the *loop-exclusive* kernel. Since the tile factorization consists of launching one thread-block, minimizing the amount of required shared memory do not have any important effect here. Figure 9 shows that for the same ib the *loop-inclusive* technique is always better than the *loop-exclusive*. The best configuration was obtained with the *loop-inclusive* approach for ib = 16.

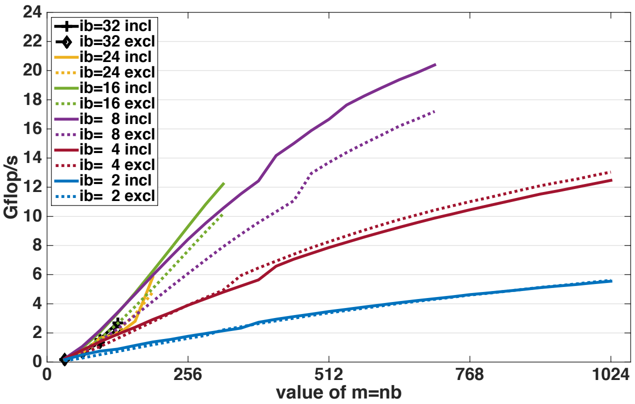


Figure 9: Performance tuning of loop-inclusive(incl) and loop-exclusive(excl) dpotf2 kernel on a Nvidia P100 GPU.

4.4 Performance Optimization based on Algorithmic Recursion

We have discussed above that *nb* is preferred to be large (for example $nb = 512$) in order to extract high performance from the update routine – either gemm or syrk. We have also proposed, implemented, optimized, and tuned a customized kernel to perform the potf2 on GPU. The best tuned *ib* for the potf2 kernel is ib = 16, as shown in Figure 9. This will limit the size of the tile that can be factorized using this parameter to $nb <= 256$ due to the register/shared memory constraints needed for holding the panel and for prefetching. A small *nb* might affect the performance of the update phase and thus might result in lower performance than expected. Going with larger *nb* means that we need to use small *ib*, which in turn means using a kernel below its possible peak performance, and this also might result in lower performance than expected.

Thus, one attractive algorithmic design that can overcome this issue is the implementation of a recursive algorithm. The idea here to to split recursively the tile of size $nb \times nb$ into smaller pieces till when the potf2 performs very well. This way we can factorize tile with large *nb* by recursion over the factorization of tiles of smaller *recnb* while continuing to gain high performance result from the potf2 kernel. Figure 10 and 11 show the performance of the potf2 kernel described in Section 4.3, and the recursive potf2 kernel described here, where the most inner recursion will call the optimized kernel of Section 4.3. The results are shown for both GPU architecture for double precision. We can easily see that the recursive implementation is needed for large *nb*

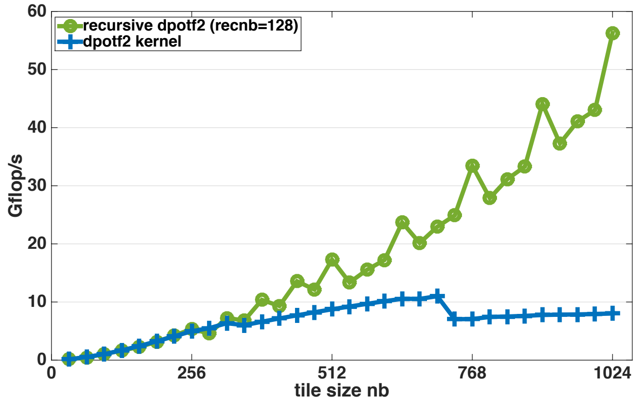


Figure 10: Effect of the recursive design of the dpotf2 routine on a Nvidia K40c GPU.

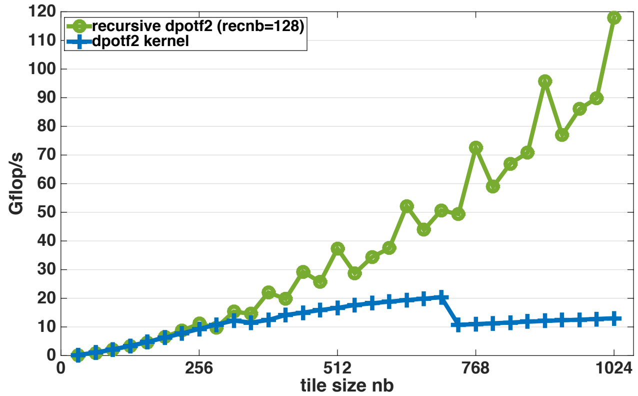


Figure 11: Effect of the recursive design of the dpotf2 routine on a Nvidia P100 GPU.

4.5 Performance Optimization based on Implementation Design

In the previous section we described how to analyze and find the performance roofline for an algorithm, as well as how to design a GPU-only version of it by optimizing the critical kernel (potf2) in such a way as to improve it and make it reach its limit. As shown above the tile factorization (potf2 routine) needs only one thread-block. However, even if this kernel requires a small amount of computational time, having the GPU running only one thread-block for this amount of time is considered from our point of view as an inefficiency and waste of resources. Also losing resources for a short time is not as dramatic as having a non optimized kernel, but we think we could take advantage of the idle resources. Moreover, in this paper we are proposing a methodology and if the potf2 routine requires small percentage of time, the other panel routines for the other factorization (such as getf2 and geqr2) might not, and thus we might see a dramatical performance degradation.

For that, we propose to further modify the design in order to achieve closer to optimal performance. In fact we propose to overlap the memory bound operation with the compute intensive operations by implementing a *lookahead* (Strazdins 1998) technique (e.g., overlapping the tile factorization (potf2 routine) with the update phase). The idea here is similar to the hybrid model but where the potf2 runs on GPUs instead of CPUs. For that, we split the update into two parts. The update of the next tile (where potf2 has to operate) and the update of the remaining. Once the update of the next tile is finished (for example, lets say at step i, once the update of the tile of step $i+1$ where the potf2 operate is done, the potf2 kernel operating on step $i+1$ can start on a separate CUDA stream as the update of the remaining portion of step i. In Figure 12 and 13, we show a snapshot of the Nvidia profiler for the right looking implementation, where we show the potf2 panel is overlapped with the update phase. Since the font and the color of traces are enforced by the Nvidia profiler, we clarified the figures by highlighting and noting the kernels and their sequence to show the overlap. We provided a high quality figures such a way that a zoom in the pdf will show the detailed font of the profiler. The trsm do not need to be overlapped since it is by itself a Level 3 BLAS routine. It run on the same stream as the potf2 routine to guarantee dependency.

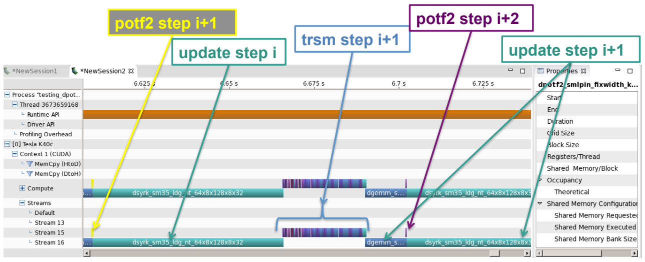


Figure 12: Nvidia profiler snapshot showing the lookahead potf2 computation executed on stream 15 overlapped with the update phase running on stream 16 for the right looking GPU-only variant.

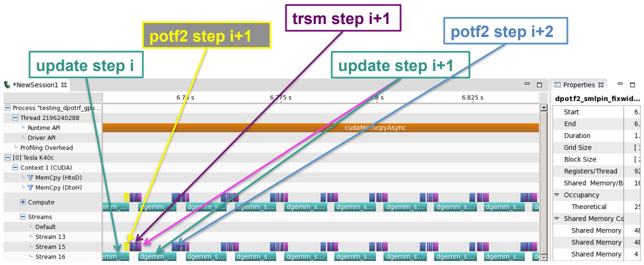


Figure 13: Nvidia profiler snapshot showing the lookahead potf2 computation executed on stream 15 overlapped with the update phase running on stream 16 for the left looking GPU-only variant.

5. Performance Discussions

In this section, we evaluate our proposed design in both of its flavors (left and right) and compare it with the best hybrid (CPU-GPU) and CPU only implementations. Performance experiments are conducted on a two-socket 10-core Intel Xeon E5-2650 v3 (Haswell), running at 2.3 GHz, and two Nvidia GPUs – the Kepler K40c (15 MP x 192 @ 0.88 GHz), and the Pascal P100 (56 MP x 64 @ 1.19 GHz). We show comparison for both single and double precision arithmetic. The hybrid performance numbers are the best obtained among the right or the left looking variant for each data point. The CPUs results were the best obtained over several runs as well and with/without the numactl interleave option. Our aims is not to compare the GPU with the CPU but we present the results obtained by a recent multicore CPUs system to make the paper self contained and to show performance on two types of hardware that have roughly the same cost.

Let's first comment on the single precision spotrf routine illustrated in Figures 14 and 16. As expected from the performance of the update routines (gemm and syrk) depicted in Figure 4 for the K40c and in Figure 5 for the P100, the left and right looking variants provide slightly different performance numbers. The left looking variant is advantageous for large matrices. This is because the sgemm routine outperforms the ssyrk routine for $nb >= 512$ and large n, while the right looking variant is advantageous for small sizes where parallelism is needed. The effect of the left looking variant on large size is better seen for the K40c GPU (Figure 14) since the difference between the sgemm and ssyrk is more pronounced for this GPU. When comparing the achieved performance by the GPU-only routine with the roofline bound illustrated in Figures 4 and 5, we can conclude that our GPU-only design is optimal and reaches close to the roofline peak. When comparing it with the hybrid routine that uses the CPU for the potf2 factorization and hides its cost completely, we can also settle that the GPU-only implementation is one of the best. We also compare it with the cuSOLVER Cholesky factorization, which is a GPU-only implementation provided by Nvidia. It can be seen that we easily outperform the vendor routine by a factor of more than 10% in single precision. Comparing it with the CPU-only implementation on such recent CPU, we can easily deduct a factor of 8 and 3 on the P100 and the K40c, respectively.

We performed the same experiments for double precision arithmetic and illustrated the results in Figures 15 and 17. In double precision the difference between the right and left looking is smoother. This is due to the fact that the gemm and the syrk routines were very well optimized and tuned for double precision. Attractively, the GPU-only routine is able to achieve performance similar to the hybrid one, which means that our proposed potf2 kernel can be considered optimal because the hybrid routine overlaps the cost of this kernel. Note that, the cost of the potf2 kernel, when optimized and tuned, is minimal and less than 5% of the total time. Our proposed design remains better than the cuSOLVER optimized routine on both the K40c and the P100. The difference on the K40c is more noticeable and is around 10%, while on the P100 is about 5%-10%.

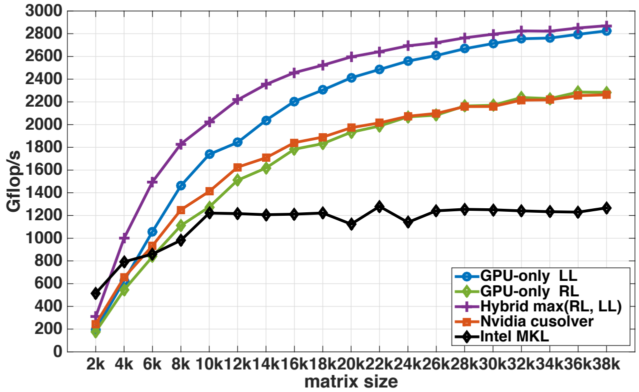


Figure 14: Performance comparison of the GPU-only, hybrid CPU-GPU and CPU-only Cholesky factorization on the Nvidia K40c GPU using single precision spotrf routine.

6. Conclusions and Future Work

In this paper, we presented the methodology of implementing numerical linear algebra routines on the contemporary hardware platforms that feature accelerators. We have provided sufficient evidence that memory bound routines can be designed, optimized,

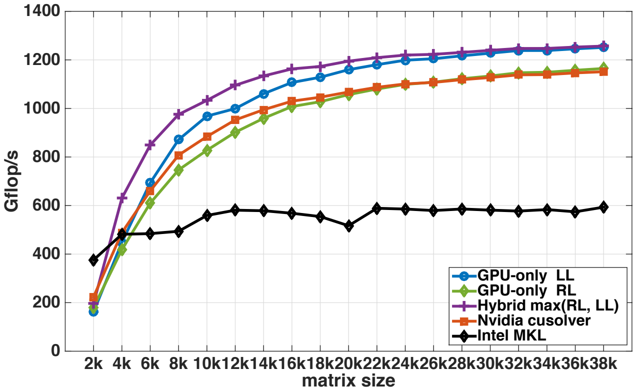


Figure 15: Performance comparison of the GPU-only, hybrid CPU-GPU and CPU-only Cholesky factorization on the Nvidia K40c GPU using double precision dpotrf routine.

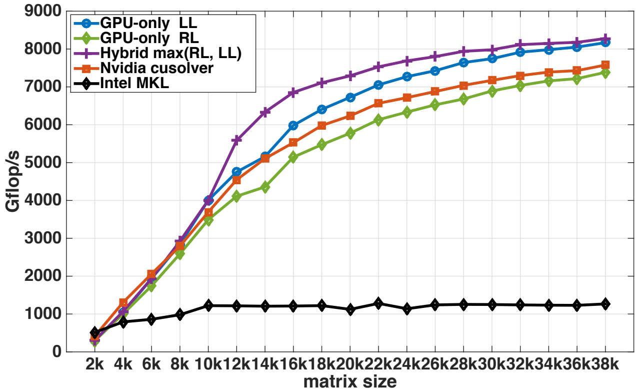


Figure 16: Performance comparison of the GPU-only, hybrid CPU-GPU and CPU-only Cholesky factorization on the Nvidia P100 GPU using single precision spotrf routine.

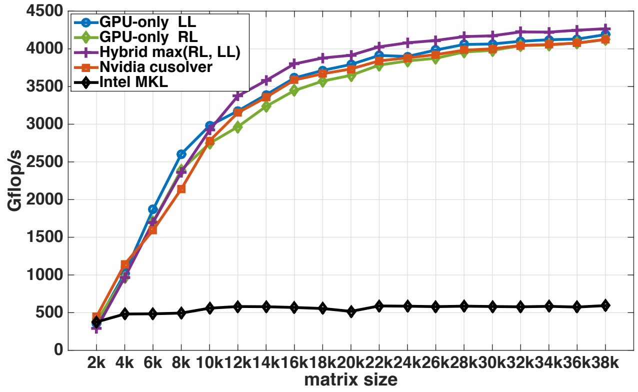


Figure 17: Performance comparison of the GPU-only, hybrid CPU-GPU and CPU-only Cholesky factorization on the Nvidia P100 GPU using double precision dpotrf routine.

and tuned for GPU architecture in a way to be competitive with CPUs and reach their theoretical limits. We show our methodology successfully applied on the development of high performance Cholesky factorization that is designed to run only on GPUs. The proposed work can deliver high performance against state-of-the-art solutions using multicore CPUs, or hybrid (CPU-GPU), or even vendor optimized GPU-only routines. Future directions consist of following the same methodology to develop other highly needed routines, such as the QR and the LU factorizations.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. ACI-1339822, the Department of Energy, and NVIDIA.

References

A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. Performance tuning and optimization techniques of fixed and variable size batched cholesky factorization on gpus. In *International Conference on Computational Science (ICCS'16)*, San Diego, CA, 06-2016 2016a.

A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. On the development of variable size batched computation for heterogeneous parallel architectures. In *The 17th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2016), IPDPS 2016*, Chicago, IL, 05-2016 2016b. IEEE, IEEE.

E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys.: Conf. Ser.*, 180(1), 2009.

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 W. mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, Sept. 2010.

E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1992. `http://www.netlib.org/lapack/lug/`.

C. Cao, J. Dongarra, P. Du, M. Gates, P. Luszczek, and S. Tomov. clmagma: High performance dense linear algebra with opencl. In *The ACM International Conference Series*, Atlanta, GA, may 13-14 2013. (submitted).

T. Dong, A. Haidar, P. Luszczek, A. Harris, S. Tomov, and J. Dongarra. LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU. In *Proceedings of 16th IEEE International Conference on High Performance and Communications (HPCC 2014)*, August 2014a.

T. Dong, A. Haidar, S. Tomov, and J. Dongarra. A fast batched Cholesky factorization on a GPU. In *Proc. of 2014 International Conference on Parallel Processing (ICPP-2014)*, September 2014b.

J. Dongarra, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and A. YarKhan. Model-driven one-sided factorizations on multicore accelerated systems. *International Journal on Supercomputing Frontiers and Innovations*, 1 (1), June 2014.

J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Hogg, P. Valero-Lara, S. D. Relton, S. Tomov, and M. Zounon. A proposed API for Batched Basic Linear Algebra Subprograms. MIMS EPrint 2016.25, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Apr. 2016. URL `http://eprints.ma.man.ac.uk/2464/`.

J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1): 1–17, Mar. 1990. ISSN 0098-3500. doi: 10.1145/77626.79170. URL `http://doi.acm.org/10.1145/77626.79170`.

M. Fatica. Accelerating linpack with cuda on heterogenous clusters. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 46–51, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-517-8. doi: 10.1145/1513895.1513901. URL `http://doi.acm.org/10.1145/1513895.1513901`.

A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra. Batched matrix computations on hardware accelerators based on GPUs. *International Journal of High Performance Computing Applications*, doi:10.1177/1094342014567546, 02/2015 .

A. Haidar, C. Cao, A. Yarkhan, P. Luszczek, S. Tomov, K. Kabir, and J. Dongarra. Unified development for mixed multi-gpu and multi-coprocessor environments using a lightweight runtime environment. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 491–500, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-3800-1. doi: 10.1109/IPDPS.2014.58. URL `http://dx.doi.org/10.1109/IPDPS.2014.58`.

A. Haidar, T. Dong, S. Tomov, P. Luszczek, and J. Dongarra. Framework for batched and gpu-resident factorization algorithms to block householder transformations. In *ISC High Performance*, Frankfurt, Germany, 07-2015 2015a. Springer, Springer.

A. Haidar, J. Dongarra, K. Kabir, M. Gates, P. Luszczek, S. Tomov, and Y. Jia. Hpc programming on intel many-integrated-core hardware with magma port to xeon phi. *Scientific Programming*, 23, 01-2015 2015b. ISSN 1058-9244. doi: 10.3233/SPR-140404.

A. Haidar, S. Tomov, P. Luszczek, and J. Dongarra. Magma embedded: Towards a dense linear algebra library for energy efficient extreme computing. In *2015 IEEE High Performance Extreme Computing Conference (HPEC 15), (Best Paper Award)*, Waltham, MA, 09-2015 2015c. IEEE, IEEE.

A. Haidar, S. Tomov, K. Arturov, M. Guney, S. Story, and J. Dongarra. Lu, qr, and cholesky factorizations: Programming model, performance analysis and optimization techniques for the intel knights landing xeon phi. In *IEEE High Performance Extreme Computing Conference (HPEC'16)*, Waltham, MA, 09-2016 2016. IEEE, IEEE.

PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.0. Innovative Computing Laboratory, University of Tennessee, 2010. `http://icl.cs.utk.edu/projectsfiles/plasma/pdf/users_guide.pdf`.

A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. Parallel performance measurement of heterogeneous parallel systems with gpus. In *Proc. of ICPP'11*, pages 176–185, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4510-3. doi: 10.1109/ICPP.2011.71. URL `http://dx.doi.org/10.1109/ICPP.2011.71`.

NVIDIA Corporation. cuSOLVER 8.0, 2016. Available at `http://docs.nvidia.com/cuda/cusolver/`.

P. E. Strazdins. Lookahead and algorithmic blocking techniques compared for parallel matrix factorization. In *10th International Conference on Parallel and Distributed Computing and Systems, IASTED*, Las Vegas, USA, 1998.

E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. TOP500 supercomputer sites, 1993-2016. Available from: `http://www.top500.org/`.

S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parellel Comput. Syst. Appl.*, 36(5-6):232–240, 2010. DOI: 10.1016/j.parco.2009.12.005.

I. Yamazaki, S. Tomov, and J. Dongarra. One-sided dense matrix factorizations on a multicore with multiple {GPU} accelerators. *Procedia Computer Science*, 9(0):37 – 46, 2012. ISSN 1877-0509. doi: http://dx.doi.org/10.1016/j.procs.2012.04.005. URL `http://www.sciencedirect.com/science/article/pii/S1877050912001263`. Proceedings of the International Conference on Computational Science, {ICCS} 2012.