Analysis and Design Techniques towards High-Performance and Energy-Efficient Dense Linear Solvers on GPUs

Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra
Innovative Computing Laboratory,
University of Tennessee, USA
{ahmad,haidar,tomov,dongarra}@icl.utk.edu

Abstract—Graphics Processing Units (GPUs) are widely used in accelerating dense linear solvers. The matrix factorizations, which dominate the runtime for these solvers, are often designed using a hybrid scheme, where GPUs perform trailing matrix updates, while the CPUs perform the panel factorizations. Consequently, hybrid solutions require high-end CPUs and optimized CPU software in order to deliver high performance. Furthermore, they lack the energy efficiency inherent for GPUs due to the use of less energy-efficient CPUs, as well as CPU-GPU communications.

This paper presents analysis and design techniques that overcome the shortcomings of the hybrid algorithms, and allow the design of high-performance and energy-efficient dense LU and Cholesky factorizations that use GPUs only. The full GPU solution eliminates the need for a high-end CPU and optimized CPU software, which leads to a better energy efficiency. We discuss different design choices, and introduce optimized GPU kernels for panel factorizations. The developed solutions achieve $90+$% of the performance of optimized hybrid solutions, while improving the energy efficiency by 50%. They outperform the vendor library by 30-50% in single precision, and 15-50% in double precision. We also show that hybrid designs trail the proposed solutions in performance when optimized CPU software is not available.

Index Terms—Dense Linear Solvers, GPU Computing, Energy Efficiency.

1 Introduction

Dense matrix factorization is the dominant step in solving large linear systems of equations. Many factorization algorithms, such as Cholesky, LU, and QR factorizations, have standard implementations in widely-distributed packages, such as the open source `LAPACK` library [1], the Math Kernel Library (MKL) [2] (provided by Intel), and the ESSL library [3] (provided by IBM). Since these factorizations are rich in compute intensive tasks, they can take advantage of massively parallel architectures. Establishing the use of GPUs in the HPC market was due to GPUs' ability to outperform multicore CPUs in such tasks. GPUs represent a throughput-oriented architecture. They possess relatively slow processing cores (compared to CPUs), but they compensate that by having orders of magnitude more cores (typically thousands instead of few dozens). In addition, they prove to provide better energy efficiency (i.e. the number of floating point operations (`FLOPs`) per unit power) for compute intensive tasks. These reasons led to mature GPU software for dense linear algebra. Examples include cuBLAS [4] and cuSOLVER [5] (vendor-supplied libraries), ViennaCL [6], and MAGMA [7]. art for GPU-accelerated dense linear solvers.

However, GPUs are not the favorite architecture for latency-sensitive or inherently sequential tasks. An example for such tasks is the panel factorization in one-sided decompositions. This is why libraries such as MAGMA, offload the panel factorization steps to the CPU, while the GPU performs the trailing matrix updates [8]. The CPU activity and the CPU-GPU communication is hidden using a *lookahead* technique, where the next panel is sent/factorized on the CPU while the current update is running on the GPU. Although this strategy achieves very high performance on GPU-accelerated systems, we point out some weaknesses:

1) Hybrid designs require an "optimal configuration", meaning a large enough problem, high-end multicore CPU, and a highly optimized CPU software for the panel factorization. A relatively slow CPU (compared to the GPU) or a slow CPU software makes it impossible, or requires very large matrix sizes, to reach asymptotic peak performance due to performance penalties associated with partial overlap of the panel factorization.
2) Hybrid designs use all cores of the CPU to factorize the panel as fast as possible. Applications that have a mix of different workloads will experience a low task concurrency.
3) A hybrid CPU-GPU factorization is, by design, suboptimal from the perspective of energy efficiency. The CPU and the GPU are working together on the same problem, with the CPU being fully engaged in a task with limited parallelism (the panel factorization). In addition, moving data across the interconnect is energy-consuming.

We provide an alternative solution that is fully GPU based. Our choice is motivated by the weaknesses men-

tioned above, as well as by the advances in the GPU architectures over the past five years. We show that a loss of less than 10% in performance can lead to a 50% gain in energy efficiency against hybrid designs with "optimal configuration". We also show that, in the absence of such an optimal configuration, the full GPU solution outperforms the hybrid solution. We address the challenges of optimizing the panel factorization on the GPU and provide novel designs that save as much memory traffic as possible. The developed solutions do not require a high-end multicore CPU nor do they require optimized CPU software. In fact, they require a single CPU thread, which leaves the rest of the host cores available to perform other useful work. Our case study includes Cholesky factorization, and LU factorization with partial pivoting. Against competitive deigns provided by cuSOLVER [5], our developed solutions achieve 5-10% performance improvements for Cholesky factorization. As for the LU factorization, the performance improvements are in the range of 30-50% in single precision, and 15-50% in double precision. The new developed solutions are lined up for integration into the MAGMA library.

2 Related Work

The study of GPU accelerated algorithms for dense linear algebra has been active for years. In fact, there are many software packages that are GPU-enabled in this field. Some of these libraries mainly provide `BLAS` functionality [9], such as NVIDIA's cuBLAS [4], KBLAS [10], and ASPEN.K2 [11]. Some other packages provide a more comprehensive set of `BLAS` and `LAPACK` [1] routines. For example, MAGMA [12] is an open source library that provides various `BLAS` kernels and `LAPACK` routines using hybrid CPU-GPU algorithms [8]. ViennaCL [6] is another package that provides few functionalities for dense matrices, such as triangular solves and LU factorization without pivoting. NVIDIA's cuSOLVER [5] also provides some GPU-powered `LAPACK` algorithms. Among all these packages, cuSOLVER is the only library that provides one-sided factorization routines that use the GPU only.

One of the early efforts to develop hybrid CPU-GPU solutions for dense linear solvers was provided by Barrachina et al. [13] and Baboulin et al. [14]. The motivation back then was that the time to transfer the panel, perform the factorization on the CPU, and transfer the panel back, can be shorter than performing the factorization on the GPU. The detailed panel factorization analysis and optimizations by Volkov and Demmel [15] also concluded that in practice, it is best to exploit the heterogeneity of the system by using both the GPU and the CPU. Such finding triggered the development of hybrid CPU-GPU solutions, not only for one-sided factorization [8], but also for singular and Eigenvalue problems [16], [17], [18]. In addition, many of these algorithms support multi-GPU systems [19], [20], [21]. Also, there was a strong interest in developing hybrid algorithm for other type of accelerated platform such as using AMD GPU [22] or Intel Xeon Phi co-processors [23], [24], [25].

In general, accelerators were used as suppliers of high performance compute-intensive `BLAS` routines. The development of GPU-only (accelerators-only) dense linear algebra algorithms was avoided in the past because: (1) accelerators were not friendly for latency-sensitive tasks (panel factorization), and (2) Hybrid algorithms were much faster, since the CPU outperforms the GPU in such tasks. However, the recent advances in accelerators architectures as well as the need for small matrix computations [26] [27] have opened the door for fully accelerators-based algorithms. Some recent work involved developing accelerators-only routine for Intel Xeon-Phi [28], [29]. We also have illustrated this in earlier work on the QR and Cholesky factorizations [30], [31] for Nvidia-GPU. Here we extend the analysis and the design techniques, and add the previously missing case for the LU factorization. We also show that the dependencies required by the hybrid solutions to achieve high performance (fast CPU and optimized software) are not always available. Moreover, the considerations for energy efficiency (currently in favor of GPUs over CPUs) are increasingly important in modern HPC platforms. These are our main motivations to investigate entirely GPU-based algorithms.

3 Contributions

This section summarizes the contributions of this paper:

1) A new design strategy for one-sided factorization that offloads the entire workload to the GPU, instead of a hybrid CPU-GPU design. Such strategy challenges the common thinking that GPUs are not efficient in latency sensitive tasks. It also provides a solution that is more energy efficient than hybrid algorithms.
2) Eliminating the need for a high-end multicore CPU in GPU-accelerated systems opens the door for building more energy efficient HPC clusters and supercomputers by using lightweight CPU cores.
3) We present a detailed study of dense Cholesky/LU factorizations that follows a bottom-up methodology. We discuss each individual component of the factorization and how we can design or tune it to operate at high speed on the GPU.
4) Three GPU kernels that perform the panel factorizations on the GPU (one kernel for Cholesky, two kernels for LU). Our strategy in the three kernels is to save as much as possible of the global memory traffic, which matters the most when optimizing for energy efficiency. One of the LU kernels uses inter-block communication among thread blocks to optimize data reuse.
5) An auxiliary pivot vector technique that enables fast row permutations in LU, instead of pairwise serial row-swapping, which improves the performance of the whole factorization.

4 Background

According to `LAPACK` [1], the LU factorization with partial pivoting of a general matrix $\mathtt{A}_{\mathtt{M}\times\mathtt{N}}$ is defined as $\mathtt{A} = \mathtt{P} \times \mathtt{L} \times \mathtt{U}$, where `P` is a permutation matrix, `L` is unit lower triangular (lower trapezoidal if $\mathtt{M} > \mathtt{N}$), and `U` is upper triangular (upper trapezoidal if $\mathtt{M} < \mathtt{N}$). The permutation matrix is stored in a condensed form using a *pivot vector* (`IPIV`).

Algorithm 1 Blocked LU factorization. Assume nb fully divides min(M, N).

1: **for** j = 0 to min(M,N) **Step** nb **do**
2: Panel: DGETF2($A_{j:M-1,j:j+nb-1}$)
3: LEFT SWAP: DLASWP($A_{j:M-1,0:j-1}$)
4: RIGHT SWAP: DLASWP($A_{j:M-1,j+nb:N-1}$)
5: Triangular Solve (DTRSM):
$A_{j:j+nb-1,j+nb:N-1}$ *= $A^{-1}_{j:j+nb-1,j:j+nb-1}$
6: Update (DGEMM):
$A_{j+nb:M-1,j+nb:N-1}$ -= $A_{j+nb:M-1,j:j+nb-1}$ * $A_{j:j+nb-1,j+nb:N-1}$
7: **end for**

Symmetric positive definite matrices are often factorized using Cholesky factorization, A = L × L^T, where L is a lower triangular matrix.

Algorithm 2 Blocked left-looking Cholesky factorization. Assume nb fully divides N.

1: **for** j = 0 to N **Step** nb **do**
2: **if** (j > 0) **then**
3: Update current panel (DSYRK, DGEMM):
$A_{j:j+nb-1,j:j+nb-1}$-=$A_{j:j+nb-1,0:j-1}A^T_{j:j+nb-1,0:j-1}$
$A_{j+nb:,j:j+nb-1}$ -= $A_{j+nb:,0:j-1}A^T_{j:j+nb-1,0:j-1}$
4: **end if**
5: Panel: DPOTF2($A_{j:j+nb-1,j:j+nb-1}$)
6: Triangular Solve (DTRSM):
$A_{j+nb:,j:j+nb-1}$ *=$A^{-1}_{j:j+nb-1,j:j+nb-1}$
7: **end for**

The key to performance in LAPACK relies on factorizing a *panel* of width nb > 1 (called the *blocking size*), in order to perform the updates using compute-bound kernels (e.g. GEMM). Algorithm 1 shows a blocked dense LU factorization with partial pivoting, which is equivalent to the standard DGETRF routine in LAPACK. It is mainly dominated by two compute-bound operations; triangular solve (DTRSM), and matrix multiplication (DGEMM). Algorithm 2 also shows a blocked Cholesky factorization. We consider the left-looking variant of the algorithm since it is dominated by the same two operations as the LU factorization (DTRSM and DGEMM). The detailed comparison between the left-looking and the right-looking variants of the algorithm is outside the scope of this paper. The internal steps of the panel factorization are discussed in Section 6.

5 Data Layout

Coalesced data access is a major key for high performance on modern GPUs. The lack of coalesced accesses in a GPU kernel results in an under-utilization of the memory bandwidth (more data are moving around than actually required), which in turn results in more power consumption and less performance. Most LAPACK algorithms assume a column major layout for dense matrices. This is convenient for Cholesky factorization, due to the absence of explicit row accesses in the algorithm. However, the situation is different in LU factorization with partial pivoting. While a column major layout is friendly for the pivot search (IDAMAX), it is not convenient for row interchanges (DLASWP). The situation is the exact opposite if row major layout is used. In this regard, we consider two layouts for the LU factorization. The first one uses a column major layout like LAPACK, and mitigates the impact of row interchanges by introducing auxiliary pivot information that enables fast row permutations (Section 6.6). Figure 1 shows the computational steps of a single iteration on a column major layout.

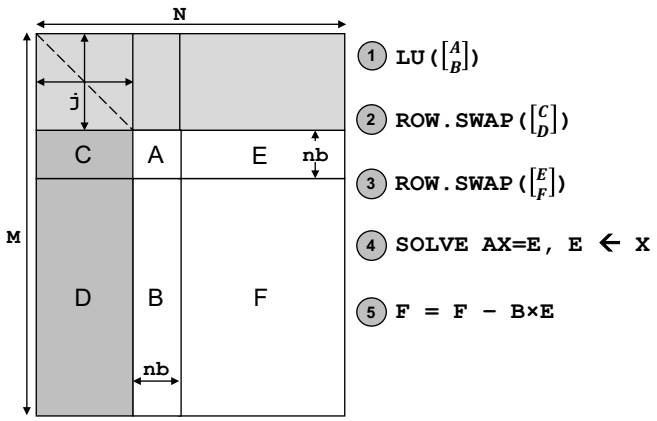


Fig. 1. Computational steps of one iteration of the LU factorization on a column major layout.

The second design still accepts a column major layout (for consistency with LAPACK), but internally transposes the matrix in order to efficiently perform the row interchanges (as in [32]). During the panel factorization, the panel is transposed back to a column major layout to efficiently perform the pivot search. Figure 2 shows the computational steps of a single iteration on the transposed matrix. Note that it is no longer needed to perform two swap operations (left and right), since the factorized panel is in a separate workspace, and is copied back after the swapping is finished. The extra steps required in this mode of operations are: (1) Two transposition steps at the beginning and at the end of the algorithm. This applies to the entire matrix, and (2) Two transposition steps at every factorization iteration. This applies to the panel only. The transposition steps are bandwidth limited. As Section 7 shows, the transposition overhead is amortized by the performance gains obtained by performing the factorization on the transposed matrix.

6 Algorithmic Design

This section discusses the design of Cholesky and LU factorizations on GPUs using a bottom-up approach. We study each individual building block of both algorithms, which gives insights on the performance of different designs and the best tuning parameters for each algorithm.

6.1 Hardware Setup

Before describing our methodology, we first list the specifications of the three systems on which our experiments are conducted.

SYSTEM 1 is configured with two sockets of a 10-core Intel Haswell CPU (Intel Xeon E5-2650 v3, running at 2.3 GHz), and a Pascal generation GPU (Tesla P100 with 1.189

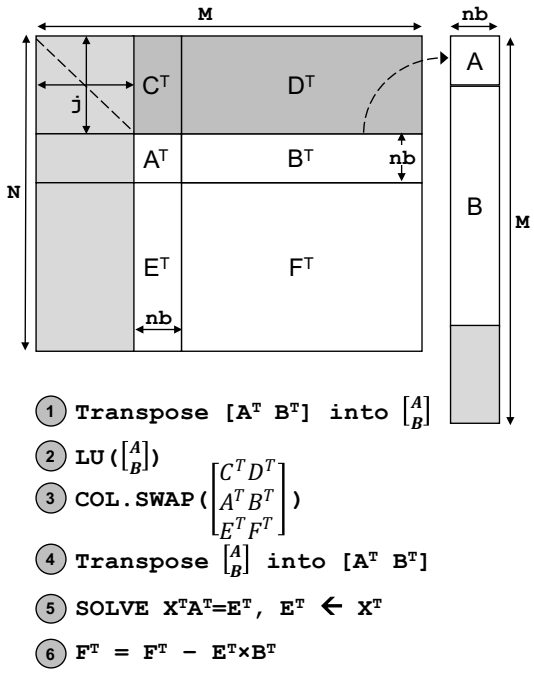


Fig. 2. Computational steps of one iteration of LU factorization on a transposed matrix.

GHz multiprocessor clock). The latter is the PCIe model, which has 16GB of HBM2 memory, and 56 multiprocessors. Each multiprocessor has 64 cores and 32 double precision units. The vendor advertises a 9.3/4.7 Tflop/s theoretical peak performance for single/double precision, respectively, and a 250W of maximum power consumption. Installed on the machine are CUDA 8.0, and MKL 11.3.0. We installed MAGMA on this system to use MKL as the provider of the CPU LAPACK. We use **SYSTEM 1** to show intermediate results for incremental improvements throughout the paper.

SYSTEM 2 is a self-hosted 68-core Intel Knights Landing (KNL) architecture (Intel Xeon Phi CPU 7250, running at 1.40GHz). We use this system to show the final results of MKL 11.3.0, and to compare the energy efficiency of the KNL architecture against the GPU architecture.

SYSTEM 3 is a single node of the ORNL's Summitdev cluster, and is also used to show our final performance results. A single node of Summitdev is equipped with two sockets of 10-core IBM POWER8 CPU (POWER8NVL (raw) with altivec, running at 2.094 GHz), and four Pascal generation GPUs (Tesla P100-SXM2 with 1.48 GHz multiprocessor clock). Each GPU has similar specifications to the one installed on **SYSTEM 1** except for the faster clock and the maximum power (300W). We run MAGMA with the ESSL-SMP library as the provider of the CPU-optimized LAPACK.

6.2 Matrix Multiplication and Triangular Solve

The GEMM and TRSM kernels are the dominant steps in both the LU and the left-looking Cholesky. We use the kernels provided by cuBLAS 8.0 [4]. The triangular solve kernel has been optimized in the latest cuBLAS release based on a recursive design that uses GEMM internally [33]. This means that we can turn our attention to the GEMM kernel only.

While the cuBLAS GEMM kernel achieves its peak performance on square multiplications, it is important to point out that this case rarely appears in one-sided factorizations. In fact, the GEMM performance relies on the input sizes, and the transposition (if any) of the input matrices. In the left-looking Cholesky factorization for example, the GEMM updates the tall and skinny matrices C (step 5 in Algorithm 2), such that $C_{m\times nb} = C_{m\times nb} - A_{m\times k} \times (B^T)_{k\times nb}$, where nb is the blocking size, while m and k keep changing as the factorization proceeds. On the other hand, LU factorization invokes the GEMM kernel to compute rank-nb matrix updates $C_{m\times n} = C_{m\times n} - A_{m\times nb} \times B_{nb\times n}$, i.e., A is often tall and skinny, B is short and wide, and C is square if the original matrix is square.

We conducted a benchmark to test the asymptotic performance of the GEMM kernel on the typical call configurations that are found in Cholesky and LU factorizations. The purpose of this experiment is to empirically decide the minimum blocking size that can get close to the peak performance (e.g., within 90%). Figure 3 shows the asymptotic behavior of the cuBLAS GEMM kernel under the two configurations using different blocking sizes. The figure shows that a minimum blocking size of 256 is required for double precision, while single precision requires at least a blocking size of 1024. These values put insights for us when we design our panel factorization kernels.

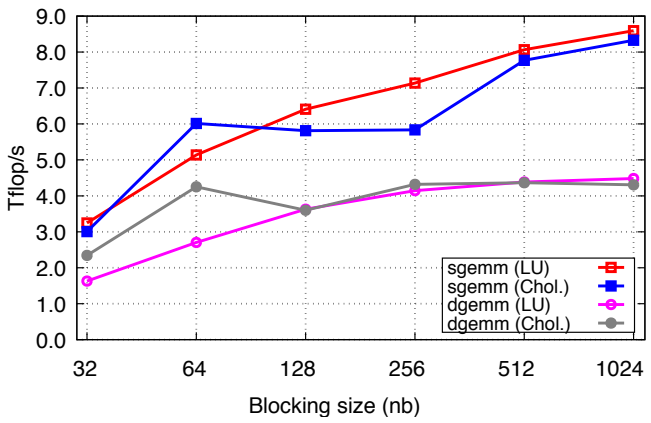


Fig. 3. Performance of the cuBLAS GEMM kernel in typical call configurations of the Cholesky/LU algorithms. Results are obtained on a Tesla P100 GPU.

6.3 Recursive Panel Factorization

The panel factorization is an unblocked implementation (i.e. column by column) of the algorithm. It is a memory-bound operation, exposes very little data reuse, and is often offloaded to the CPU (as in MAGMA) [12], [34]. On the contrary, we developed optimized GPU kernels for Cholesky and LU panel factorizations. All the developed kernels are compliant with LAPACK in terms of error checking, which means that the kernels check for non-positive-definiteness in Cholesky factorization, and for singularity in LU factorization.

However, the previous section shows that we need relatively large blocking sizes in order to get the best GEMM

performance. Even an optimized kernel should not be used to factorize a panel of width 1024 or even 256. The panel factorization becomes very time consuming, and remains bound by the memory bandwidth. This is why we follow a *recursive panel design* in order to involve compute-bound operations in the panel step. Figure 4 shows a sketch of the recursive panel in both algorithms. The width of the panel (rnb) at which recursion stops is a tuning parameter that varies according to the GPU architecture.

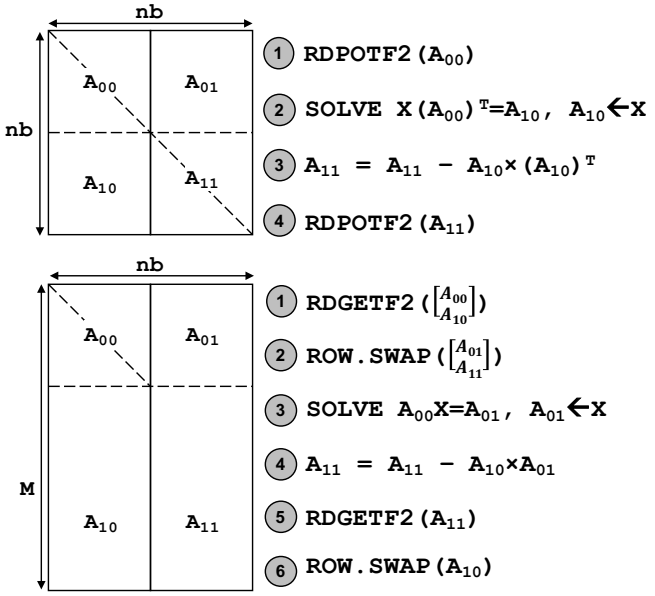


Fig. 4. Recursive panel of Cholesky (RDPOTF2) and LU (RDGETF2) factorizations.

The RDPOTF2 and RDGETF2 routines have a stopping condition if the width of the panel becomes less than a certain value (typically 16 to 32), where the unblocked POTF2 or GETF2 factorization routines are executed. If the width of the panel is larger than the stopping condition, the steps shown in Figure 4 are executed. The recursive Cholesky panel factorization starts by a recursive call on the A_{00} submatrix. The next step is to update A_{10} inplace using a triangular solve with respect to the factorized A_{00} submatrix. A_{11} is then rank-updated before passing it to another recursive factorization. On the other hand, the RDGETF2 routine starts by a recursive call on A_{x0}. The necessary row interchanges are then performed on A_{x1} before updating A_{01} using an inplace triangular solve with respect to A_{00}. A_{11} is then rank-updated and recursively factorized. The last step is to perform the necessary row interchanges, resulting from factorizing A_{11}, on the A_{10} submatrix.

6.4 Cholesky Panel Factorization (DPOTF2)

We developed an optimized GPU kernel to factorize an $rnb \times rnb$ square matrix. Unlike LU factorization, the panel is relatively small, and can be factorized using one thread block with rnb threads. We deviate from a fully unblocked implementation and use in-kernel blocking with fused customized BLAS functions in order to maximize data reuse. Figure 5 shows the design concept of the kernel. Following a left-looking scheme, the kernel iterates over the matrix using an internal blocking size ib, and holds an $(rnb-j) \times ib$ subpanel in shared memory, where $j = 0, ib, 2ib, \cdots, etc.$ For every subpanel, the kernel performs an in-shared-memory update ($C = C - AB^T$) using a customized DGEMM sub-kernel. Note that this operation overwrites the upper triangular part of C in shared memory, but ignores it when writing the subpanel back to the global memory. This extra computation allows a simpler kernel design and less thread divergence during the update. The customized DGEMM sub-kernel takes advantage of the overlap between A and B by reading a portion of A of tunable width kb, and transposing the corresponding part of B in shared memory. It also incorporates register double buffers to perform the updates using overlapping stages, for example by prefetching a_1 (which includes b_1) while computing $a_0 \times b_0^T$. After the update is complete, the kernel reuses the subpanel in shared memory and executes the unblocked factorization sub-kernel. The factorized panel is finally written to the global memory, and can also be reused for the update of the next subpanel. We point out that, with such a kernel design, our solution for Cholesky factorization involves three layers of blocking before executing an unblocked factorization (LAPACK blocking → recursive blocking → kernel blocking → unblocked).

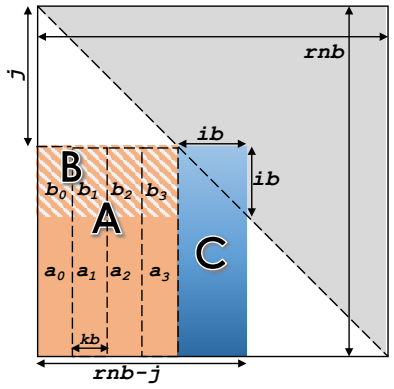


Fig. 5. Design of the Cholesky panel factorization.

6.5 LU Panel Factorization (DGETF2)

The LU panel factorization is more challenging to optimize on GPUs. This is because blocking reduces only the columns in a panel, and thus, can not make a panel small enough to be factorized entirely in "fast memory". Indeed, following Algorithm 1, the recursive panel sizes are $(M - j) \times rnb$, where $j = 0, nb, 2nb, \cdots, etc.$ Since we cannot make any assumptions about the memory footprint size of the input panel, which is unlike the Cholesky algorithm, we have to provide a generic design (denoted below by D_1) that works for any size. However, we can take advantage of the panel if its height is below some threshold T, where we can explore more chances of data reuse. We adopt this strategy and develop two different designs for the LU panel factorization. The basic idea is, as the panel gets shorter, we are able to launch a more optimized kernel D_2 to perform the factorization. The threshold T is a tunable parameter that depends on the GPU architecture. We point out that large-scale matrix

factorizations also benefit from Design D_2. Namely, when the height of the unfactorized part becomes less than T, our recursive panel routine switches automatically from D_1 to D_2.

6.5.1 Design D_1

This design works for any panel size. It launches multiple kernels that correspond to the computational steps described in Algorithm 3, except for the DSCAL and DGER operations, which are fused into one kernel to increase data reuse. At each iteration, Design D_1 starts by launching the IDAMAX kernel, which uses one thread block of IDAMAX_{TX} threads (typically 256 to 512). The kernel loops over the current column in segments of IDAMAX_{TX} elements, with each thread computing its local maximum. The next stage is a tree reduction in shared memory, after which the pivot location is determined and written to IPIV. The DSWAP kernel is a simple one that just exchanges two rows of the current panel. Since this kernel does not respect coalesced memory access, we keep the the panel width as small as $8-16$.

Algorithm 3 Unblocked LU factorization

```
1: for j = 0 to min(M, N) do
2:   Locate Pivot (IDAMAX):
     piv = IPIV[j] = j + IDAMAX(A[j:, j])
3:   if piv≠j then
4:     SWAP (DSWAP): A[j, 0:] with A[piv, 0:]
5:   end if
6:   SCALE (DSCAL):
     A[j+1:,j] *= 1/A[j, j]
7:   UPDATE (DGER):
     A[j+1:,j+1:] -= A[j+1: ,j]*A[j,j+1:]
8: end for
```

The DSCAL and DGER steps are fused into one kernel in order to reuse the data from the scaled column. The kernel launches a 1D array of thread blocks, each with DSCALGER_{TX} threads (typically $128-512$). Figure 6 shows an example that uses three thread blocks. Each thread block reads the pivot row and stores it in shared memory. It then uses the pivot element to scale its respective segment of "a" and, while writing it into global memory, reuses it in the outer product to update the respective part of C in registers.

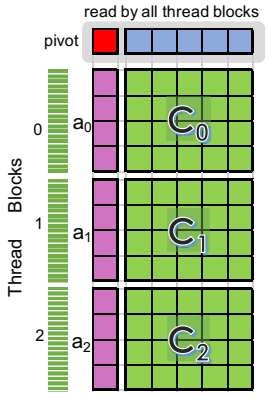


Fig. 6. The fused DSCAL/DGER kernel.

6.5.2 Design D_2

As the factorization progresses, the height of the panel gets smaller than some threshold (T), and we can take advantage of that by introducing an alternate design D_2 that increases data reuse in fast memory levels. This reduces the memory traffic and increases the energy efficiency. We set a target to make T as large as possible, by introducing a design that fuses all the computational steps of Algorithm 3 into a single kernel. The kernel uses multiple thread blocks and inter-block communication among thread blocks when sharing of information is needed. Each thread block holds an entire column of the panel in registers, which enables T to be as large as 50k/25k in single/double precisions, respectively.

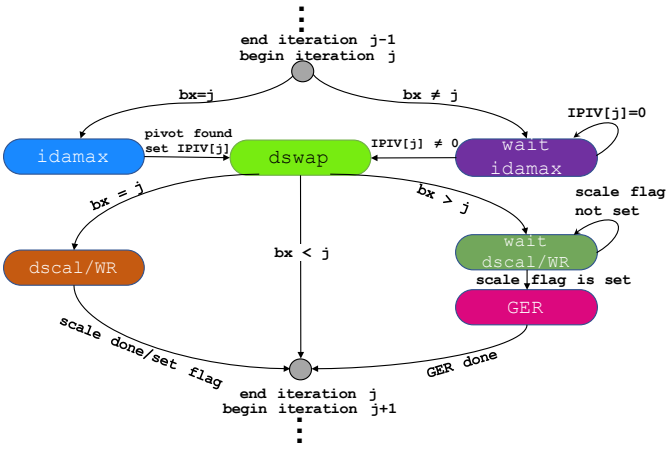


Fig. 7. A FSM for a single iteration for Design D_2

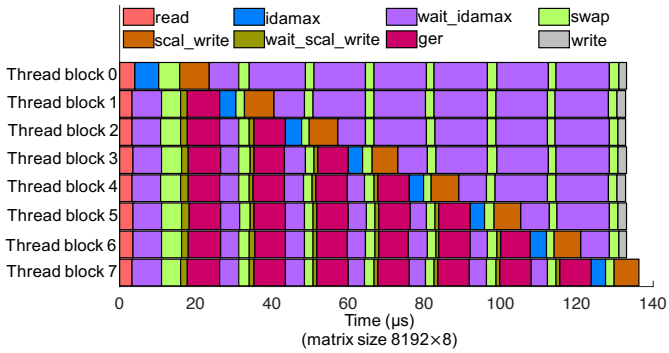
CUDA (versions 8.0 and earlier) does not natively support inter-block communication. It also adopts a weakly-ordered memory model, meaning that the order of reads and writes to the global memory does not necessarily follow the order specified in the developed code. The inter-block communication of Design D_2 is realized using three components. The first is global memory flags that enable thread blocks to synchronize and share data. We adopt a simple mechanism by assigning a separate set of flags for each thread block, in order to avoid race conditions or deadlocks. Each set of flags is readable by all thread blocks, but may be written by only a unique thread block. The extra workspace needed to hold these flags is negligible even if the original matrix is small. The second component is atomic operations for variables that are writable by all thread blocks, such as the info parameter that is used to report singularity. We use atomic exchange operations in order to set such variables, which also ensures that the new written value is visible to all thread blocks. The third component is memory fence functions, which are used to make sure that data writes to the global memory are visible to all thread blocks. This is typically needed to trigger the update operation, when the scaled column is written to the global memory.

The design of D_2 uses a *finite state machine (FSM)* to coordinate the factorization among independent thread blocks. Figure 7 shows the state machine of the j^{th} iteration of

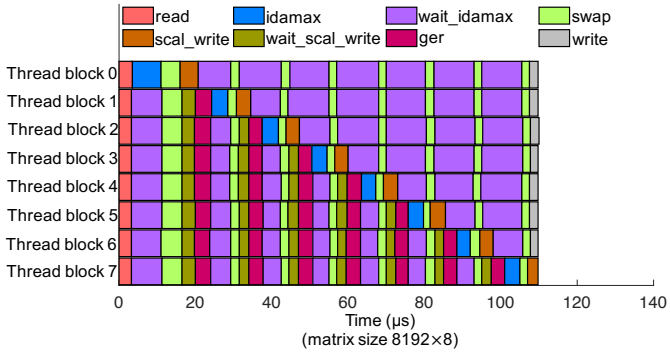
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2842785, IEEE Transactions on Parallel and Distributed Systems

Design D_2. The thread block who has an ID bx equal to j performs the IDAMAX step on its column, while other thread blocks wait for the j^{th} entry of the pivot vector to be updated to a non-zero value. Once the pivot is found, all thread blocks perform the DSWAP step in registers and then select one of three paths, based on their respective IDs. Thread blocks with ID less than j move to the next iteration and wait for a new pivot. The j^{th} thread block scales its column and writes it to global memory, so that thread blocks with ID $>$ j (that are meanwhile waiting) can begin the update operation DGER. All operations occur on the register level, and swapping does not lead to non-coalesced memory accesses. Data reuse is significantly increased as well, compared to Design D_1, since all columns are kept in registers for the lifetime of the kernel.

The kernel driver makes sure that all thread blocks are simultaneously live in order to avoid deadlocks. This is done by launching a number of thread blocks that is less than or equal to the number of multiprocessors on the GPU. We also force the runtime to schedule exactly one thread block per multiprocessor by allocating more than half the shared memory available. The driver reads, at run time, the number of multiprocessors of the GPU, and tunes the value of rnb accordingly. For example, on a Pascal P100, we run at least 32 thread blocks, since the GPU has 56 multiprocessors. Each thread block is a 1D array of $D2_{TX}$ threads, typically $256 - 512$ based on the GPU architecture. Each column is stored in registers using multiple segments, each of length $D2_{TX}$.



(a) Design D_2 with greedy update



(b) Design D_2 with lazy update

Fig. 8. Execution traces of thread blocks inside the D_2 kernel with greedy/lazy updates. Matrix size is 8192×8. GPU is Tesla P100.

6.5.3 *Tracing Execution for Design D_2*

We conducted an experiment that enables tracing the execution of thread blocks within the D_2 kernel. The purpose of this trace is two fold. First, it generates a realistic visualization of the state machine shown in Figure 7. Second, it allows assessing a tradeoff between a *greedy update* and a *lazy update*. The former overlaps the DSCAL and DGER operations by triggering an update for every written segment of the factorized column. The latter waits for the entire column to be written and then triggers the update. Therefore, the tradeoff is between having more parallelism versus less synchronization among thread blocks.

We modified the D_2 kernel in order to allow thread blocks log their progress in an auxiliary workspace in the global memory. Each thread block records the value of the special GPU register %globaltimer at the times it enters/leaves a certain state. Figure 8 shows two visualizations of the log information for a matrix of size 8192×8. The time scale is the same for the two traces. In either case, eight thread blocks are launched. They follow the state machine shown in Figure 7, except in the way they execute the update. A greedy update (Figure 8a) pipelines the DSCAL and the DGER operations, thus seeking more parallelism. Although the overlap between the two stages is apparent, the greedy update suffers from frequent synchronization among thread blocks, which is required after scaling each segment of the current column. This causes the DSCAL and the DGER operations to take longer times than they do in Figure 8b, which shows that a traditional "lazy" update is faster due to the use of fewer synchronization points.

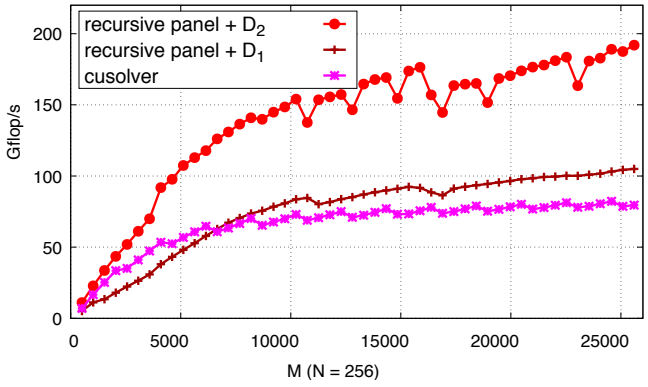


Fig. 9. Performance comparison of Designs D_1 and D_2 on a Tesla P100 in double precision arithmetic.

6.5.4 *Performance Comparison between D_1 and D_2*

Figure 9 compares the impacts of Designs D_1 and D_2 on the performance of the recursive panel factorization of Figure 4. Shown also is the performance of cuSOLVER. The experiment uses tall and skinny matrices, where the panel factorization kernels become more dominant than the update kernels. We use a fixed panel width of 256, while varying the number of rows up to $\approx$ 25k. The recursive panel using Design D_1 trails cuSOLVER for panel heights up to 7k, but steadily outperforms cuSOLVER as the panel becomes taller.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2842785, IEEE Transactions on Parallel and Distributed Systems

The recursive panel with Design D2 outperforms the other two implementations for every size, achieving asymptotic speedups of 2.4× against cuSOLVER, and 1.8× against the recursive panel calling D_1. We also observe oscillations in the best performing design. Our profiling results show that the oscillations are due to other components of the recursive panel (i.e. BLAS kernels operating on small sizes). These oscillations still appear when Design D_1 is used, but they are less apparent since the factorization kernels are more dominant.

Unfortunately, the higher performance comes at the cost of increased register pressure as the panel grows taller. We empirically observe that Design D_2 is limited to panels shorter than 25k in double precision. Considering single precision, we are able to factorize panels as tall as 40k, which is the maximum size used in the reported results of this paper. As mentioned before, Design D_1 is used whenever the D_2 limit is reached. The recursive panel transparently makes a choice between the two designs every time the factorization is done.

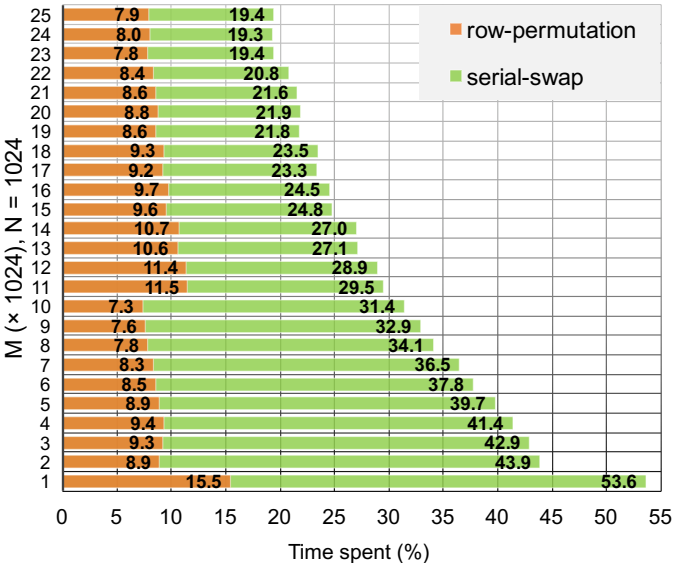


Fig. 10. Percentage time spent on row interchanges on a Tesla P100 GPU. Matrix width is fixed at 1024. Results are for double precision arithmetic.

6.6 Fast Row Permutations

The standard LU factorization in LAPACK uses a pivot vector IPIV to store the permutation matrix P in a condensed form. The LAPACK documentation describes IPIV as an integer vector that stores pivot indices, such that for 1 ≤ j ≤ min(M,N), row j of the matrix was interchanged with row IPIV[j]. When a panel of width nb is factorized, the corresponding nb entries in the pivot vector are used to perform the row interchanges to the left and right of the panel. The row interchanges are required in the design shown in Figure 1, and also in the recursive panel design shown in Figure 4. This operation leads to many non-coalesced memory accesses, and is purely memory-bound, since no FLOPs are performed.

However, the way the pivots are stored forces a *serial pairwise row swapping*. In other words, the DLASWP routine must loop over the pivot vector, interchanging two rows at a time for each entry. For example, consider a factorization of an 8×8 matrix that results in a pivot vector of [7, 7, 4, 4, 5, 8, 7, 8]. The order of swapping is fixed, e.g., row 7 has been selected as pivot in the first two iterations, which means that, generally, a row can change its place multiple times during the row interchanges. This leads to a serial behavior and redundant non-coalesced memory traffic.

We developed an auxiliary pivot vector XIPIV that mitigates this effect. The use of XIPIV is internal only, and we still provide the same pivot vector that is provided by the LAPACK software. The new pivot vector describes "*permutations*" rather than "*a series of pairwise swaps*". It stores the final destination of each row, so that each row changes its place once, and permutations occur in parallel. For example, the equivalent XIPIV vector to the example discussed above is [7, 1, 4, 3, 5, 8, 2, 6]. We can read it as "the first row of the new matrix is the seventh row of the original matrix", and "the second row of the new matrix is the first row in the original matrix", and so on. We build the two vectors as the factorization goes on, and use XIPIV to perform the permutations. Two GPU kernels have been developed for this purpose. The first converts from IPIV to XIPIV, and the second performs the permutations using XIPIV. Note that writing back the permuted nb rows can now be done using coalesced memory accesses.

Figure 10 compares the percentage time spent in row interchanges between the LAPACK-style swapping and the new permutation mechanism. For the latter, we include the times of both the conversion and the permutations. Results are shown for matrices of sizes $M \times 1024$, which are typical panel sizes. We observe that row permutations are at least 2.4× faster than the serial swapping, and can be up to 4.9× faster. Row permutations take around $7 - 8\%$ of the time of the whole factorization, compared to $19 - 20\%$ in the case of serial swapping. The increase in the percentage time after size 10k for row permutations is due to a switch-of-kernels when building up the XIPIV vector. Below size 10k, we are able to build the new vector in shared memory and write it once to the global memory. Another kernel is used for larger sizes, where XIPIV is built directly in global memory.

7 Experimental Results

This section presents the final performance and energy efficiency results for the developed Cholesky and LU factorizations, which we call *magma-native*. Comparisons are made against (1) the MAGMA-hybrid routines (on **SYSTEMs 1 & 3**), (2) cuSOLVER (on **SYSTEMs 1 & 3**), which is provided by the vendor and uses a full GPU design, (3) the MKL library running on **SYSTEM 1** (which we call *mkl-haswell*), (4) the MKL library running on **SYSTEM 2** (which we call *mkl-knl*), and (5) the ESSL-SMP library running on **SYSTEM 3** (which we call *essl-power8*). MAGMA hybrid routines uses the multithreaded MKL/ESSL for the panel factorization on **SYSTEM 1/2**, with the number of threads set to 20. Experiments on the KNL hardware set the value of OMP_NUM_THREADS to 68. These five competitors represent

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2842785, IEEE Transactions on Parallel and Distributed Systems

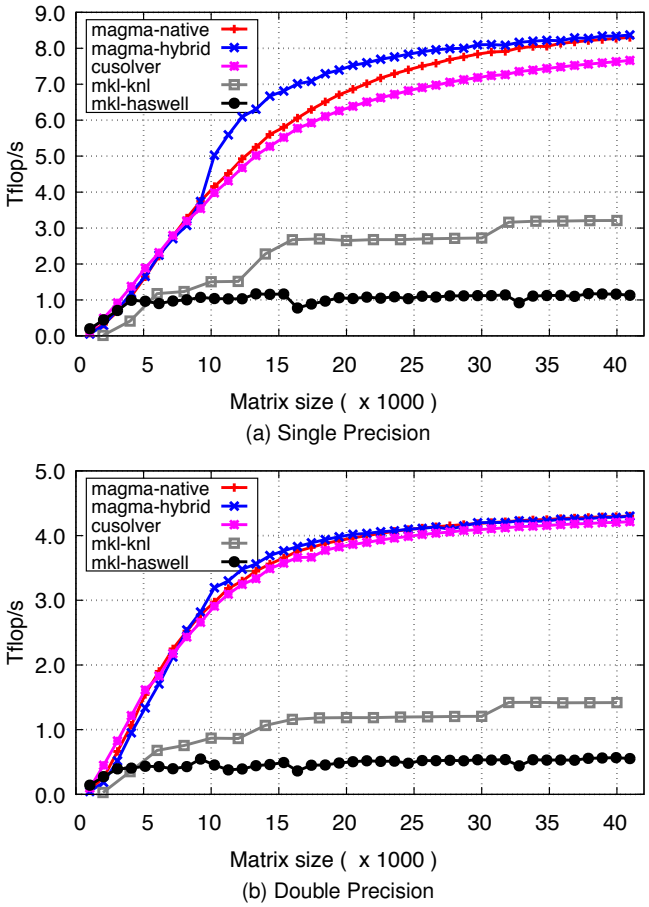


Fig. 11. Performance of Cholesky factorizationon on **SYSTEMs 1 & 2**.

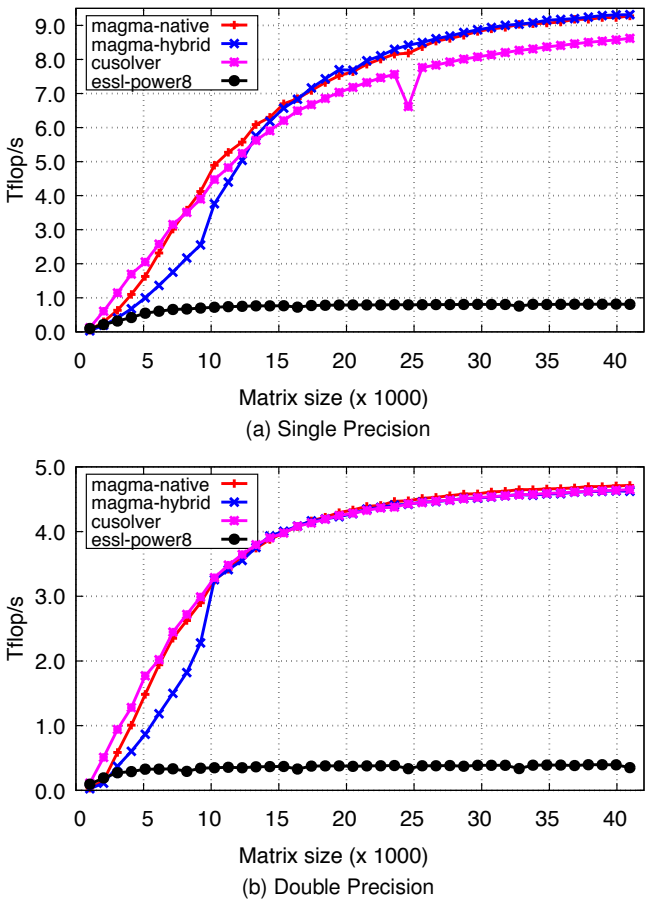


Fig. 12. Performance of Cholesky factorization on **SYSTEM 3**.

the state of the art in their respective categories (CPU+GPU, GPU only, and CPU (and KNL) only).

7.1 Performance

7.1.1 Cholesky Factorization

Figure 11 shows the performance of the Cholesky factorization on **SYSTEMs 1 & 2**. In single precision (Figure 11a), the *magma-native* solution asymptotically matches the performance of *magma-hybrid*. This is where the time of the panel factorization becomes almost negligible with respect to the rest of the computational steps. This means that large matrices do not have to use a hybrid CPU+GPU solutions to achieve high performance. However, the midrange observes an advantage of the hybrid design that is up to 20%. This advantage comes at the cost of increased energy consumption, as we show later in Section 7.2. The native solution slightly outperforms *magma-hybrid* and cuSOLVER for matrices of size less than 8K, and outperforms cuSOLVER afterwards by up to 10%. In double precision (Figure 11b) the performances of *magma-native*, *magma-hybrid*, and cuSOLVER are very similar across all sizes. We also observe that a full GPU solution for Cholesky factorization can be up to $8\times/7\times$ faster than a CPU solution in single/double precision. Against MKL running on the KNL platform, the asymptotic speedups of *magma-native* are $2.5\times/3.0\times$ in single/double precision.

Figure 12 shows the performance of the Cholesky factorization on **SYSTEM 3**. We observe that *magma-hybrid* is not competitive until sizes beyond 10k in both precisions. This behavior indicates that the CPU panel is not as fast as it is on **SYSTEM 1**. In addition, while cuSOLVER has a slight advantage for small sizes, *magma-native* outperforms cuSOLVER in single precision by an asymptotic speedup of 10%, and is nearly identical to cuSOLVER and *magma-hybrid* in double precision for sizes larger than 10k. We also observe that *magma-native* is asymptotically more than $10\times$ faster than the pure CPU solution using the multithreaded ESSL library.

7.1.2 LU Factorization

Figure 13 shows the performance of the LU factorization on **SYSTEMs 1 & 2**. Recall that we provide two native solutions based on the data layout of the matrix. The *magma-native (NT)* graph corresponds to the design for column major layouts in Figure 1, while the *magma-native (T)* graph corresponds to the design of Figure 2. The performance of the latter includes all overheads of the transpositions and workspace allocations. In both precisions, a transposition of the input matrix achieves the better performance out of the two proposed solutions. However, we observe a very slight advantage of the non-transposed solution for sizes less than 4k/3k in single/double precision. For the rest of the discussion, we focus on the transposed solution only.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2842785, IEEE Transactions on Parallel and Distributed Systems

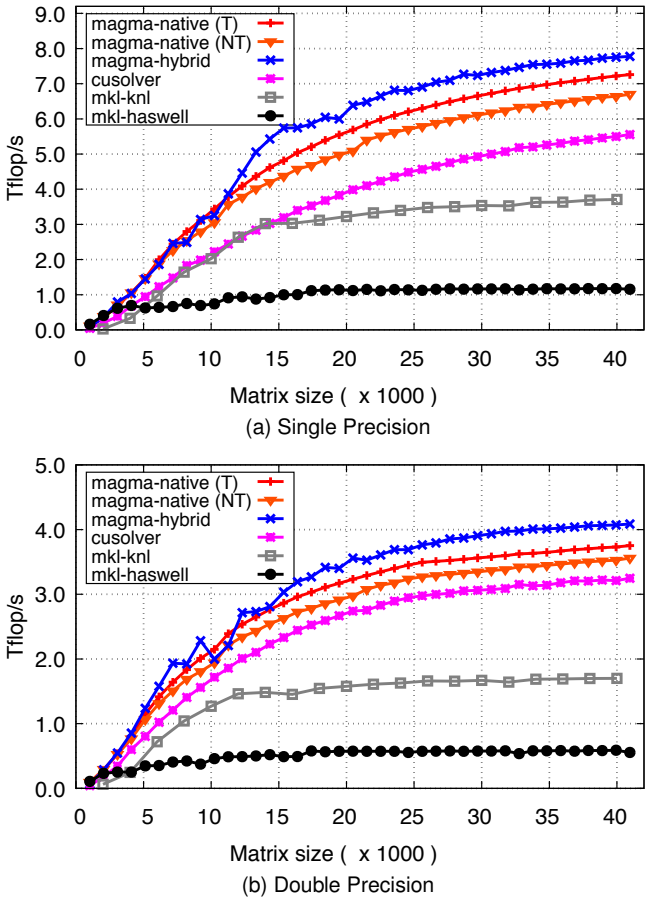


Fig. 13. Performance of LU factorization on **SYSTEMs 1 & 2**.

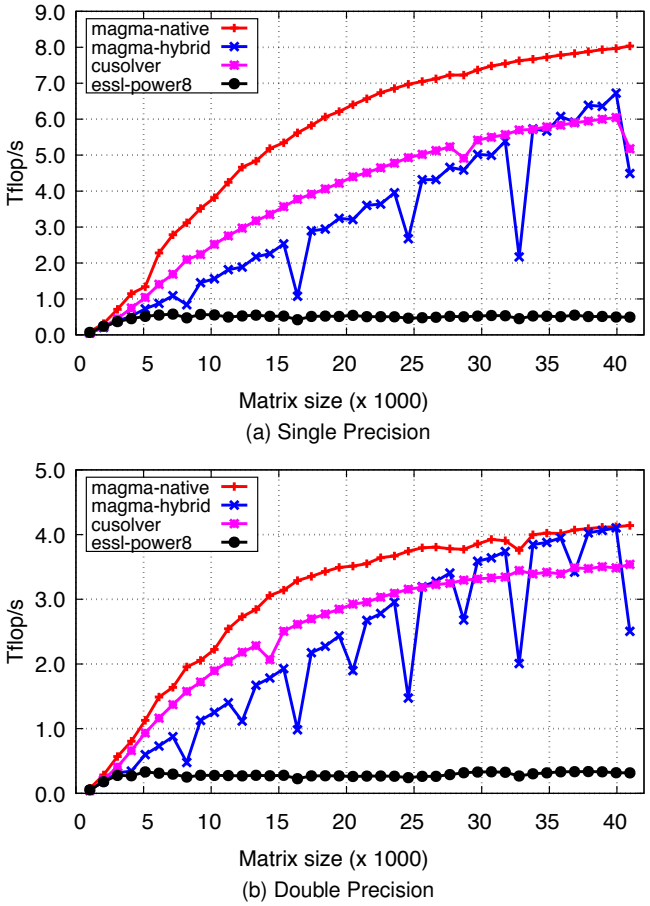


Fig. 14. Performance of LU factorization on **SYSTEM 3**.

In single precision, the *magma-native (T)* solution matches the *magma-hybrid* performance for sizes up to 10k, and mostly stays within 10% or less of the *magma-hybrid* performance for larger matrices. Note that the LU panel is much more expensive than the Cholesky panel, and its cost grows as the matrix becomes larger. This explains why our solution does not match the *magma-hybrid* performance asymptotically. We also outperform cuSOLVER by speedups that range between 30%-50%. In double precision, *magma-native (T)* nearly matches the performance of *magma-hybrid* up to size 15k, and then maintains a performance within at least 90% of the MAGMA performance. *Magma-native (T)* also scores speedups against cuSOLVER that range between 15%-50%. In both precisions, we show that a full GPU solution can be up to $6\times$ faster than a high performance CPU solution. It is also $1.9\times/2.2\times$ faster than MKL running on the KNL architecture for single/double precision.

Figure 14 shows the performance of the LU factorization on **SYSTEM 3**, where we observe a behavior opposite to Figure 13. Surprisingly, the native GPU routines (both *magma-native* and cuSOLVER) are generally faster than *magma-hybrid*. This is a scenario where the reliance on the CPU to factorize the panel does not pay off. Due to the lack of enough optimization and the oscillatory performance, the panel factorization routine on the CPU becomes a bottleneck, resulting in performance penalties. The *magma-native* routine is asymptotically 33%/17% faster than cuSOLVER in single/double precision. With the performance drops of *magma-hybrid* excluded, the speedups scored by *magma-native* are up to $2.6\times$ in single precision, and $2.03\times$ in double precision. In addition, *magma-native* is asymptotically $10\times/12\times$ faster than a pure CPU solution.

To better illustrate the contradicting behaviors of *magma-hybrid* between Figures 13 and 14, we profiled the factorization of a $10\text{k}\times10\text{k}$ matrix in double precision. Figures 15a and 15b show partial execution traces on **SYSTEMs 1 & 3**, respectively. On **SYSTEM 1**, the panel factorization on the CPU is fast enough to overlap the GEMM update on the GPU, resulting in the very good performance of Figure 13. On **SYSTEM 3**, however, and due to the lack of enough optimization, the CPU panel becomes a bottleneck, and the GPU has to wait for it to finish. This means that hybrid routines are not necessarily the best performing solution. On the absence of optimized CPU software, they can be outperformed by other solutions.

7.1.3 Performance Efficiency

While absolute performance comparisons show a general advantage for the MAGMA hybrid designs on **SYSTEM 1**, a comparison that is based on the *performance as percentage of the peak* shows the opposite. A hybrid design has a performance upper bound that is equal to the sum of the peak performances of the CPU and the GPU, while the upper bound of a full GPU design is the peak of the GPU plus the peak of one CPU core. Rather than the theoretical peak performance, we use a more practical bound that is

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2842785, IEEE Transactions on Parallel and Distributed Systems

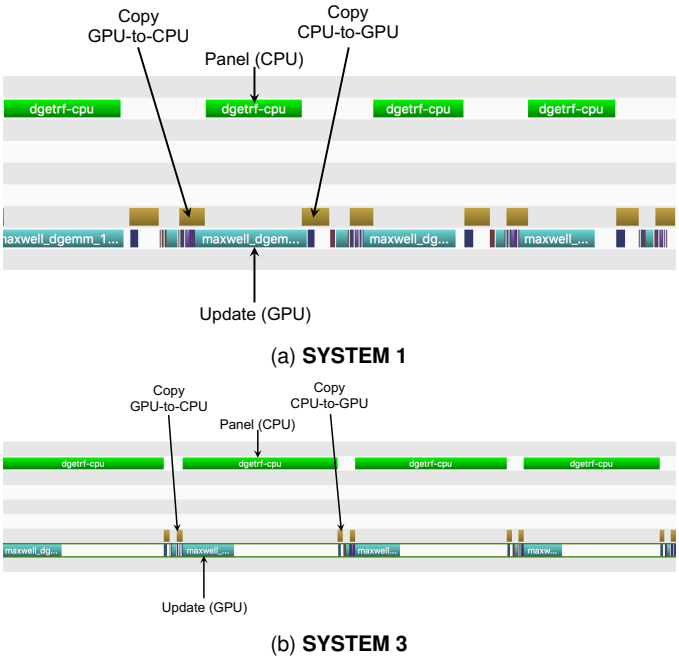


Fig. 15. Overlapping the CPU-based LU panel with the GPU-based updates. Matrix size is 10k.

equal to the peak GEMM performance. Our experiments show 8.9/4.55 Tflop/s peaks for the P100 GPU (using cuBLAS), and 1.2/0.6 Tflop/s for the 20-core Haswell CPU (using MKL). Figure 16 shows the asymptotic performance as a percentage of the aggregate GEMM peak. In every category, our solution achieves a better performance efficiency than the MAGMA hybrid designs, with more than 90% performance efficiency for Cholesky factorization, and more than 80% performance efficiency for LU factorization.

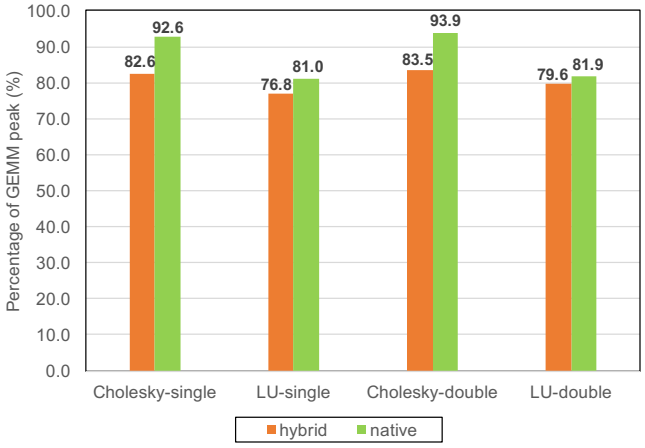


Fig. 16. Asymptotic performance as percentage of the aggregate GEMM peak (**SYSTEM 1**).

7.2 Energy Efficiency

The energy efficiency experiments are conducted on **SYSTEMs 1 & 2** only, due to the lack of power measurement software on Summitdev. Our energy measurements use the

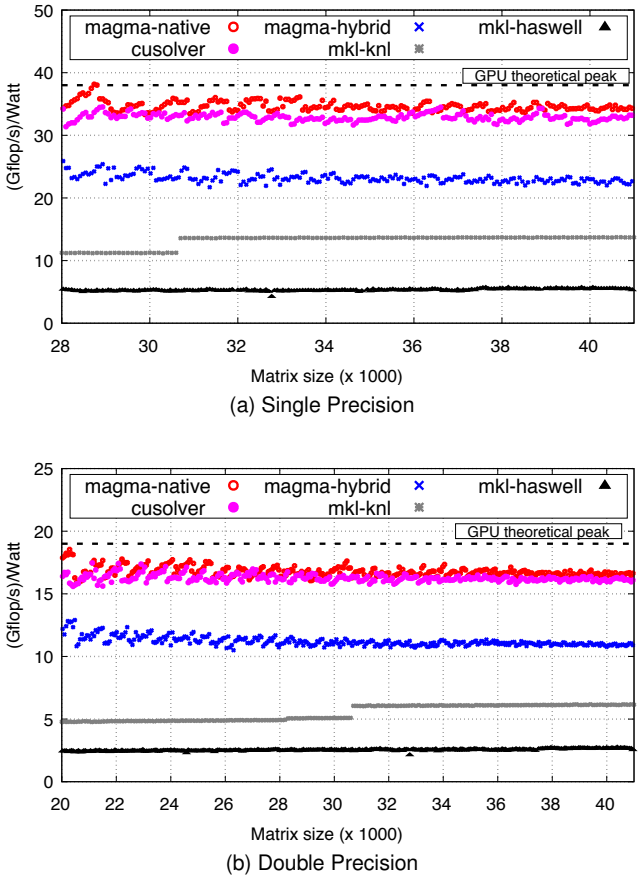


Fig. 17. Performance per Watt for Cholesky factorization on **SYSTEMs 1 & 2**.

Intel RAPL driver for the CPU and NVIDIA NVML library for the GPU. We use a sampling window of 100ms in order to get stable readings, as recommended in [35]. These software tools are validated to provide accuracy that is within +/- 5% of current power draw [36]. The energy efficiency is measured in (Gflop/s)/Watt. For every data point, we divide the achieved performance by the average power consumption based on the several samples collected during the computation. In order to have enough samples, we consider matrices that require at least a second to factorize. We also point out that, for cuSOLVER and *magma-native*, we add the power of one CPU socket to the GPU power. Table 1 summarizes the average performance per Watt for all solutions, while Figures 17 and 18 show more detailed results for every size that has been tested.

Solution Name	Cholesky		LU	
	SP	DP	SP	DP
MAGMA-native	38.91	17.70	29.40	14.64
cuSOLVER	36.97	17.17	21.85	12.09
MAGMA-hybrid	26.83	12.22	20.33	10.04
MKL-KNL	12.02	5.33	15.43	6.95
MKL-CPU	5.08	2.44	4.87	2.56

TABLE 1
The average (Gflop/s)/W scored on **SYSTEMs 1 & 2**.

Figure 17 shows the performance per Watt for Cholesky factorization. *Magma-native* shows a slight advantage over

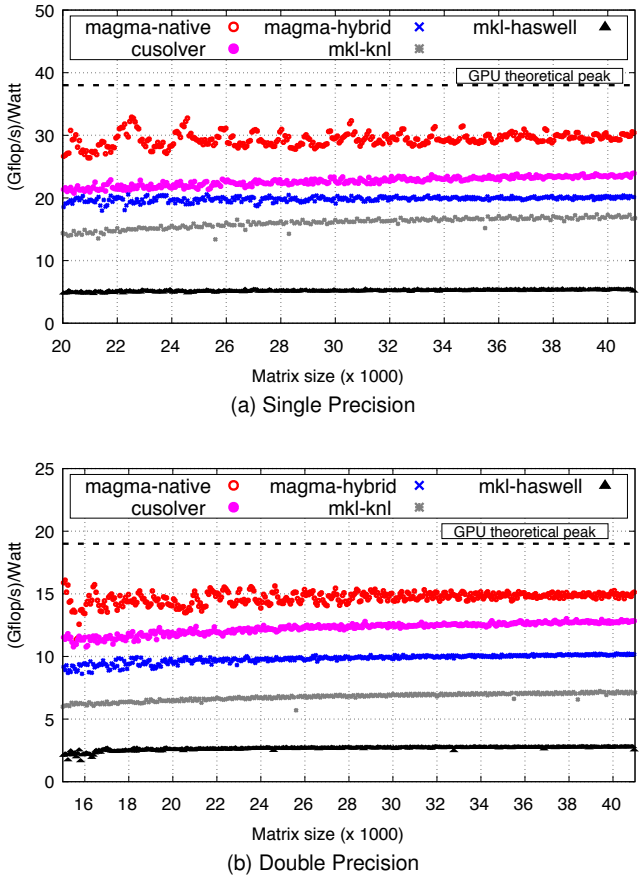


Fig. 18. Performance per Watt for LU factorization with partial pivoting on **SYSTEMs 1 & 2**.

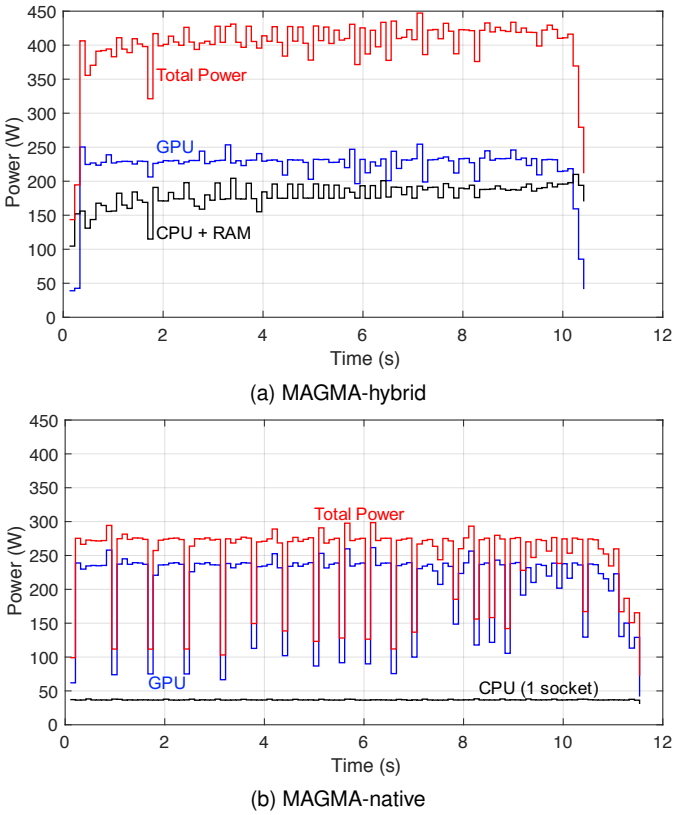


Fig. 19. Power traces for double precision LU factorization on a $40k \times 40k$ matrix (**SYSTEM 1**).

cuSOLVER in both precisions. More importantly, the proposed design achieves about $35/17$ (Gflop/s)/Watt in single/double precision, which is at least 50% better than the *magma-hybrid* implementation. The achieved performance per Watt is also within $90+\%$ of the theoretical peak, as advertised by the vendor. The discontinuity in the KNL results correspond to the performance spikes after size $30k$, which obviously leads to a better performance per Watt.

A similar behavior is observed for LU factorization, as shown in Figure 18. Not only does *magma-native* achieve a similar 50% improvement against *magma-hybrid*, but it also outperforms cuSOLVER by $25\%/15\%$ in single/double precision. Figures 17 and 18 show that a full GPU solution can achieve $6 - 7\times$ better performance per Watt than a 20-core Haswell CPU. They also show that the developed solution is $1.8 - 3.0\times$ more energy efficient than the KNL platform running MKL.

Figures 19a and 19b show a power trace for LU factorization of a $40k \times 40k$ matrix using *magma-hybrid* and *magma-native*, respectively. Both figures have the same scale in both axes. The figures show the main difference between the hybrid and the native designs. The former keeps the CPU operating at full power, performing the panel factorization and communicating with the GPU. The latter barely uses any CPU resources, and so the CPU in *magma-native* consumes at least $4\times$ less power. Figure 19b also shows an interesting behavior of nearly periodic drops in the GPU power consumption. These drops correspond to the panel factorization kernels. Such kernels are very memory-bound and lack compute-intensiveness, thus consuming way less power than the update steps.

8 Conclusion and Future Work

This paper presented an alternative design that is high performant and power efficient for dense one-sided factorization on GPUs. Rather than a hybrid design that uses the CPU and the GPU, we adopt a full GPU design to improve the energy efficiency. Our optimized GPU kernels for performing the panel factorization enables achieving 90% of the performance of the hybrid design, while scoring up to 50% improvement in performance per unit power. They also achieve 15%-60% performance improvements against a competitive GPU design by the vendor. In addition, we show that hybrid designs can be slower than the proposed solution in the absence of optimized CPU software.

Future directions include following a similar path for QR factorization, singular and Eigenvalue problems, and multi-GPU designs. The authors are also interested in optimizations for embedded systems, such as the Jetson TX2 platform, where hybrid designs fail to provide high performance due to the absence of server class CPUs and fast interconnects. Another work direction is to apply the GPU only factorizations to large problems that do not fit at once in the GPU memory. These problems are solved using out-of-GPU-memory algorithms [37], where a hybrid panel can be replaced by the GPU-only panel to seamlessly benefit from the new developments. Finally, we are currently working

on extending the techniques and the GPU-only panel factorizations presented to be used as essential building blocks in developing high-performance and energy-efficient factorizations for large scale heterogeneous distributed-memory supercomputers.

ACKNOWLEDGMENTS

This work is partially supported by NSF Grant No. CSR 1514286, NVIDIA, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nations exascale computing imperative.

REFERENCES

[1] "LAPACK - Linear Algebra PACKage," "http://www.netlib.org/lapack/".
[2] "Intel Math Kernel Library," available at http://software.intel.com/intel-mkl/.
[3] "Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL," available at https://www-03.ibm.com/systems/power/software/essl/.
[4] "NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS)," available at https://developer.nvidia.com/cublas.
[5] "NVIDIA cuSOLVER: A Collection of Dense and Sparse Direct Solvers," available at https://developer.nvidia.com/cusolver.
[6] K. Rupp, F. Rudolf, and J. Weinbub, "ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs," in *Intl. Workshop on GPUs and Scientific Applications*, 2010, pp. 51–56.
[7] 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.*, vol. 180, no. 1, 2009.
[8] S. Tomov, J. Dongarra, and M. Baboulin, "Towards Dense linear algebra for Hybrid GPU Accelerated Manycore Systems," *Parellel Comput. Syst. Appl.*, vol. 36, no. 5-6, pp. 232–240, 2010, DOI: 10.1016/j.parco.2009.12.005.
[9] "BLAS (Basic Linear Algebra Subprograms)," "http://www.netlib.org/blas/".
[10] A. Abdelfattah, D. Keyes, and H. Ltaief, "KBLAS: An Optimized Library for Dense Matrix-Vector Multiplication on GPU Accelerators," *ACM Trans. Math. Softw.*, vol. 42, no. 3, pp. 18:1–18:31, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2818311
[11] "Automatic Stabilizing and PErformaNce tuning of level 2 BLAS Kernels," available at http://www.aics.riken.jp/labs/lpnctrt/ASPENK2_e.html.
[12] "Matrix Algebra on GPU and Multicore Architectures (MAGMA)," 2016, available at http://icl.cs.utk.edu/magma/.
[13] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí, *Solving Dense Linear Systems on Graphics Processors*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 739–748. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85451-7_79
[14] M. Baboulin, , J. Dongarra, and S. Tomov, "Some Issues in Dense Linear Algebra for Multicore and Special Purpose Architectures," Electrical Engineering and Computer Science Department, Tech. Rep. UT-CS-08-615, May 6 2008, also available as LAPACK Working Note 200.
[15] V. Volkov and J. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.
[16] C. Vmel, S. Tomov, and J. Dongarra, "Divide and Conquer on Hybrid GPU-Accelerated Multicore Systems," *SIAM Journal on Scientific Computing*, vol. 34, no. 2, pp. C70–C82, 2012. [Online]. Available: https://doi.org/10.1137/100806783
[17] R. Solcà, A. Kozhevnikov, A. Haidar, S. Tomov, J. Dongarra, and T. C. Schulthess, "Efficient Implementation of Quantum Materials Simulations on Distributed CPU-GPU Systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 10:1–10:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807654
[18] A. Haidar, S. Tomov, J. Dongarra, R. Solca, and T. Schulthess, "A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks," *International Journal of High Performance Computing Applications*, September 2012, (accepted).
[19] A. Haidar, M. Gates, S. Tomov, and J. Dongarra, "Toward a scalable multi-gpu eigensolver via compute-intensive kernels and efficient communication," in *ICS'13: 27th International Conference on Supercomputing (submitted)*, Eugene, Oregon, USA, June 10-14 2013.
[20] A. Haidar, R. Solca, M. Gates, S. Tomov, T. Schulthess, and J. Dongarra, "Leading edge hybrid multi-GPU algorithms for generalized eigenproblems in electronic structure calculations," in *ICS'13: International Supercomputing Conference (submitted)*, Hamburg, Germany, June 2013.
[21] I. Yamazaki, T. Dong, R. Solc, S. Tomov, J. Dongarra, and T. Schulthess, "Tridiagonalization of a Dense Symmetric Matrix on Multiple GPUs and its Application to Symmetric Eigenvalue Problems," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 16, pp. 2652–2666, 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.3152
[22] C. Cao, M. Gates, A. Haidar, P. Luszczek, S. Tomov, and I. Yamazaki, "Performance and Portability with OpenCL for Throughput-Oriented HPC Workloads across Accelerators, Coprocessors, and Multicore Processors," in *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, Nov 2014, pp. 61–68.
[23] 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*, ser. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 491–500. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2014.58
[24] J. Dongarra, M. Gates, A. Haidar, Y. Jia, K. Kabir, P. Luszczek, and S. Tomov, "Portable HPC Programming on Intel Many-Integrated-Core Hardware with MAGMA Port to Xeon Phi," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 571–581.
[25] A. Haidar, P. Luszczek, S. Tomov, and J. Dongarra, "Heterogenous Acceleration for Linear Algebra in Multi-coprocessor Environments," in *High Performance Computing for Computational Science – VECPAR 2014*, M. Daydé, O. Marques, and K. Nakajima, Eds. Cham: Springer International Publishing, 2015, pp. 31–42.
[26] V. Oreste, M. Fatica, N. A. Gawande, and A. Tumeo, "Power/performance trade-offs of small batched LU based solvers on GPUs," in *19th International Conference on Parallel Processing, Euro-Par 2013*, ser. Lecture Notes in Computer Science, vol. 8097, Aachen, Germany, August 26-30 2013, pp. 813–825.
[27] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "On the Development of Variable Size Batched Computation for Heterogeneous Parallel Architectures," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*, 2016, pp. 1249–1258.
[28] 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 *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2016, pp. 1–7.
[29] K. Kabir, A. Haidar, S. Tomov, and J. Dongarra, "On the Design, Development, and Analysis of Optimized Matrix-Vector Multiplication Routines for Coprocessors," in *High Performance Computing*, J. M. Kunkel and T. Ludwig, Eds. Cham: Springer International Publishing, 2015, pp. 58–73.
[30] 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)*, Sept 2015, pp. 1–6.
[31] A. Haidar, A. Abdelfatah, S. Tomov, and J. Dongarra, "High-

performance Cholesky Factorization for GPU-only Execution," in *Proceedings of the General Purpose GPUs*, ser. GPGPU-10. New York, NY, USA: ACM, 2017, pp. 42–52. [Online]. Available: http://doi.acm.org/10.1145/3038228.3038237

[32] V. Volkov and J. W. Demmel, "LU, QR and Cholesky factorizations using vector capabilities of GPUs," University of California, Berkeley, Tech. Rep. UCB/EECS-2008-49, May 13 2008, also available as LAPACK Working Note 202.

[33] A. Charara, H. Ltaief, and D. E. Keyes, "Redesigning Triangular Dense Matrix Computations on GPUs," in *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, 2016, pp. 477–489.

[34] S. Tomov and J. Dongarra, "Dense Linear Algebra for Hybrid GPU-based Systems," in *Scientific Computing with Multicore and Accelerators*, J. Kurzak, D. A. Bader, and J. Dongarra, Eds. Chapman and Hall/CRC, 2010.

[35] H. McCraw, J. Ralph, A. Danalis, and J. Dongarra, "Power monitoring with PAPI for extreme scale architectures and dataflow-based programming models," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2014, pp. 385–391.

[36] NVIDIA Corporation, "Nvml api reference guide," 2017, available at https://docs.nvidia.com/deploy/nvml-api.

[37] I. Yamazaki, S. Tomov, and J. Dongarra, "One-sided dense matrix factorizations on a multicore with multiple GPU accelerators," *Procedia Computer Science*, vol. 9, no. 0, pp. 37 – 46, 2012, proceedings of the International Conference on Computational Science, ICCS 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050912001263

Ahmad Abdelfattah Ahmad Abdelfattah received his PhD in computer science from King Abdullah University of Science and Technology (KAUST) in 2015, where he was a member of the Extreme Computing Research Center (ECRC). He is currently a research scientist in the Innovative Computing Laboratory at the University of Tennessee. He works on optimization techniques for different linear algebra workloads in the MAGMA library. Ahmad has B.Sc. and M.Sc. degrees in computer engineering from Ain Shams University, Egypt.

Azzam Haidar Azzam Haidar received a Ph.D. in 2008 from CERFACS, France. He is Research Scientist at the Innovative Computing Laboratory at the University of Tennessee, Knoxville. His research interests focus on the development and implementation of parallel linear algebra routines for scalable distributed multi-core and GPU architectures, for large-scale dense and sparse problems, as well as new algorithms for singular value (SVD) and eigenvalue problems as well as approaches that combine direct and iterative algorithms to solve large linear systems.

Stanimire Tomov Stanimire Tomov received a M.S. degree in Computer Science from Sofia University, Bulgaria, and Ph.D. in Mathematics from Texas A&M University. He is a Research Director in ICL and Adjunct Assistant Professor in the EECS at UTK. Tomov's research interests are in parallel algorithms, numerical analysis, and high-performance scientific computing (HPC). Currently, his work is concentrated on the development of numerical linear algebra software for emerging architectures for HPC.

Jack Dongarra Jack Dongarra received a Bachelor of Science in Mathematics from Chicago State University in 1972 and a Master of Science in Computer Science from the Illinois Institute of Technology in 1973. He received his Ph.D. in Applied Mathematics from the University of New Mexico in 1980. He worked at the Argonne National Laboratory until 1989, becoming a Senior Scientist. He now holds an appointment as University Distinguished Professor of Computer Science in the Department of Electrical Engineering and Computer Science at the University of Tennessee, has the position of a Distinguished Research Staff member in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL), Turing Fellow in the Computer Science and Mathematics Schools at the University of Manchester, and an Adjunct Professor in the Computer Science Department at Rice University.