Batched Matrix Computations on Hardware Accelerators Based on GPUs

Azzam Haidar[*], Tingxing Dong[*], Piotr Luszczek[*], Stanimire Tomov[*], and Jack Dongarra[*,†,‡]

[*]University of Tennessee
[†]Oak Ridge National Laboratory
[‡]University of Manchester

November 1, 2014

Abstract

Scientific applications require solvers that work on many small size problems that are independent from each other. At the same time, the high-end hardware evolves rapidly and becomes ever more throughput-oriented and thus there is an increasing need for effective approach to develop energy efficient, high-performance codes for these small matrix problems that we call *batched factorizations*. The many applications that need this functionality could especially benefit from the use of GPUs, which currently are four to five times more energy efficient than multicore CPUs on important scientific workloads. This paper, consequently, describes the development of the most common, one-sided factorizations: Cholesky, LU, and QR for a set of small dense matrices. The algorithms we present together with their implementations are, by design, inherently parallel. In particular, our approach is based on representing the process as a sequence of batched BLAS routines that are executed entirely on a GPU. Importantly, this is unlike the LAPACK and the hybrid MAGMA factorization algorithms that work under drastically different assumptions of hardware design and efficiency of execution of the various computational kernels involved in the implementation. Thus, our approach is more efficient than what works for a combination of multicore CPUs and GPUs for the problems sizes of interest of the application use cases. The paradigm where upon a single chip (a GPU or a CPU) factorizes a single problem at a time is not at all efficient for in our applications' context. We illustrate all these claims through a detailed performance analysis. With the help of profiling and tracing tools, we guide our development of batched factorizations to achieve up to two-fold speedup and three-fold better energy efficiency as compared against our highly optimized batched CPU implementations based on MKL library. The tested system featured two sockets of Intel Sandy Bridge CPUs and we compared to a batched LU factorizations featured in the CUBLAS library for GPUs, we achieve as high as 2.5× speedup on the NVIDIA K40 GPU.

1 Introduction

An improved data reuse is what drives the design of algorithms to work well on small problems, which, in the end, delivers higher performance. When working on small problems it is possible to improve the reuse as the input data gets loaded into the fast memory, it can be used presumably many times until the completion of the task. Many numerical libraries as well as applications already use this functionality but it needs to be further developed. For example, the tile algorithms from the area of dense linear algebra [2], various register and cache blocking techniques for sparse computations [11], sparse direct multifrontal solvers [29], high-order FEM [7], and numerous applications including astrophysics [17], hydrodynamics [7], image processing [18], signal processing [5], are examples of this trend.

The lack of linear algebra software for small problems is especially noticeable for GPUs. The development for CPUs, as pointed out in Sections 2 and 4.1, can be done easily using existing software infrastructure. On the other hand, GPUs, due to their throughput-oriented design, are efficient for large data parallel computations, and therefore have often been used in combination with CPUs, where the CPU handles the small and difficult to parallelize tasks. The need to overcome the challenges of solving small problems on GPUs is also related to the GPU's energy efficiency often four to five times better than the one for multicore CPUs. To take advantage of it, codes ported to GPUs must exhibit high efficiency. This is one of the main goals of this work: to develop GPU algorithms and their implementations on small problems in order to outperform multicore CPUs in raw performance and energy efficiency. In particular, we target the main one-sided factorizations – LU, QR, and Cholesky – for a set of small dense matrices of the same size.

Figure 1 gives a schematic view of the batched problem considered. Basic block algorithms, as the ones in LAPACK [4], factorize at step i a block of columns, denoted by panel P_i, followed by the application of the transformations accumulated in the panel factorization to the trailing sub-matrix A_i.

$$
\begin{array}{lll}
\text{Cholesky}: \texttt{dpotrf}(A^{(1)}) \rightarrow LL^{\mathsf{T}} & \text{LU}: \texttt{dgetrf}(A^{(1)}) \rightarrow P^{-1}LU & \text{QR}: \texttt{dgeqrf}(A^{(1)}) \rightarrow QR \\
\text{Cholesky}: \texttt{dpotrf}(A^{(2)}) \rightarrow LL^{\mathsf{T}} & \text{LU}: \texttt{dgetrf}(A^{(2)}) \rightarrow P^{-1}LU & \text{QR}: \texttt{dgeqrf}(A^{(2)}) \rightarrow QR \\
\ldots & \ldots & \ldots \\
\text{Cholesky}: \texttt{dpotrf}(A^{(k)}) \rightarrow LL^{\mathsf{T}} & \text{LU}: \texttt{dgetrf}(A^{(k)}) \rightarrow P^{-1}LU & \text{QR}: \texttt{dgeqrf}(A^{(k)}) \rightarrow QR
\end{array}
$$

Figure 1: Schematic view of a batched one-sided factorization problem for a set of k dense matrices. An approach based on batched BLAS factorizes the matrices simultaneously.

Interleaved with the algorithmic work are questions on what programing and execution model is best for small problems, how to offload work to the GPUs, and what should be the interaction with the CPUs if any. The offload-based execution model and the accompanying terms *host* and *device* have been established by the directive-based programming standards: OpenACC [21] and OpenMP 4 [22]. While these specifications are *host-centric*, in the context of dense linear algebra computations, we recognize three distinctly different modes of operation: hybrid, native, and batched execution. The first one employs both the host CPU and the device accelerator, be it a GPU

or an Intel coprocessor, that cooperatively execute on a particular algorithm. The second one offloads the execution completely to the accelerator. The third one is the focus of this writing and involves execution of a multitude of small problems on the accelerator while the host CPU only sends the input data and receives the computed result in a pipeline fashion to alleviate the overheads of the dearth of PCIe bandwidth and comparatively long latency of the transfers.

2 Related Work

Small problems can be solved efficiently on single CPU core, e.g., using vendor supplied libraries such as MKL [14] or ACML [3], because the CPU's memory hierarchy would back a "natural" data reuse (small enough problems can fit into small fast memory). Besides memory reuse, to further speedup the computation, vectorization to use SIMD processor supplementary instructions can be added either explicitly as in the Intel Small Matrix Library [13], or implicitly through the vectorization in BLAS. Batched factorizations then can be efficiently computed for multicore CPUs by having a single core factorize a single problem at a time (see Section 4.1). However, as we show, the energy consumption is higher than the GPU-based factorizations.

For GPU architectures, prior work has been concentrated on achieving high-performance for large problems through hybrid algorithms [26]. Motivation came from the fact that the GPU's compute power can not be used on panel factorizations as efficiently as on trailing matrix updates [27]. As a result, various hybrid algorithms were developed where panels are factorized on the CPU while the GPU is used for trailing matrix updates (mostly GEMMs) [1, 10]. For large enough problems the panel factorizations and associated with it CPU-GPU data transfers can be overlapped with GPU work. For small problems however, this is not possible, and our experience has shown that hybrid algorithms would not be as efficient as they are for large problems.

Indeed, targeting very small problems (of size up to 128), Villa et al. [23], [24] obtained good results for batched LU developed entirely for GPU execution, where a single CUDA thread, or a single thread block, was used to solve one system at a time. Similar techniques, including the use of single CUDA thread warp for single factorization, were investigated by Wainwright [28] for LU with full pivoting on matrices of size up to 32. Although the problems considered were often small enough to fit in the GPU's shared memory, e.g., 48 KB on a K40 GPU, and thus to benefit from data reuse (n^2 data for $\frac{2}{3}n^3$ flops for LU), the results showed that the performance in these approaches, up to about 20 Gflop/s in double precision, did not exceed the maximum performance due to memory bound limitations (e.g., 46 Gflop/s on a K40 GPU for DGEMV's $2n^2$ flops on n^2 data; see also the performance analysis in Section 5.2).

Here we developed an approach based on batched BLAS plus some batched-specific algorithmic improvements that exceeds in performance the memory bound limitations mentioned above. A batched LU based on batched BLAS has been also recently developed and released through CUBLAS [20], but has lower performance compared to our approach when the algorithmic improvements are added.

3 Algorithmic Background

In this section, we present a brief overview of the linear algebra aspects for development of either Cholesky, Gauss, or the Householder QR factorizations based on block outer-product updates of the trailing matrix. Conceptually, one-sided factorization maps a matrix A into a product of matrices X and Y:

$$\mathcal{F}: \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \mapsto \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \times \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix}.$$

Algorithmically, this corresponds to a sequence of in-place transformations of A, whose storage is overwritten with the entries of matrices X and Y (P_{ij} indicates currently factorized panels):

$$\begin{bmatrix} A_{11}^{(0)} & A_{12}^{(0)} & A_{13}^{(0)} \\ A_{21}^{(0)} & A_{22}^{(0)} & A_{23}^{(0)} \\ A_{31}^{(0)} & A_{32}^{(0)} & A_{33}^{(0)} \end{bmatrix} \rightarrow \begin{bmatrix} P_{11} & A_{12}^{(0)} & A_{13}^{(0)} \\ P_{21} & A_{22}^{(0)} & A_{23}^{(0)} \\ P_{31} & A_{32}^{(0)} & A_{33}^{(0)} \end{bmatrix} \rightarrow$$

$$\rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & A_{22}^{(1)} & A_{23}^{(1)} \\ X_{31} & A_{32}^{(1)} & A_{33}^{(1)} \end{bmatrix} \rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & P_{22} & A_{23}^{(1)} \\ X_{31} & P_{32} & A_{33}^{(1)} \end{bmatrix} \rightarrow$$

$$\rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & XY_{22} & Y_{23} \\ X_{31} & X_{32} & A_{33}^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & X_{22} & Y_{23} \\ X_{31} & X_{32} & P_{33} \end{bmatrix} \rightarrow$$

$$\rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & XY_{22} & Y_{23} \\ X_{31} & X_{32} & XY_{33} \end{bmatrix} \rightarrow \begin{bmatrix} XY \end{bmatrix},$$

where XY_{ij} is a compact representation of both X_{ij} and Y_{ij} in the space originally occupied by A_{ij}.

There are two distinct phases in each step of the transformation from $[A]$ to $[XY]$: *panel factorization* (P) and trailing matrix update $A^{(i)} \rightarrow A^{(i+1)}$. Implementation of these two phases leads to a straightforward iterative scheme shown in Algorithm 1.

Algorithm 1 is called block algorithm since every panel P is of size nb which allows the trailing matrix update to use the Level 3 BLAS routines. Note that if $nb = 1$ the algorithm falls back to the standard algorithm introduced by LINPACK in the 80's. The factorization of each panel is accomplished by a non-blocked routine. Table 1 shows the BLAS and the LAPACK routines that should be substituted for the generic routines named in the algorithm. Most of the current libraries focus on large matrices by using hybrid (CPU-GPU) algorithms [12]. Because the panel factorization is considered a latency-bound workload, which faces a number of inefficiencies on throughput-oriented GPUs, it was preferred to perform its factorization on the CPU. Due to their high performance rate exhibited on the update operation, and the fact that the update requires the majority of floating-point operations, the GPU has to perform the trailing matrix update. Note that a

	Cholesky	Householder	Gauss
PanelFactorize	xPOTF2 xTRSM	xGEQF2	xGETF2
TrailingMatrixUpdate	xSYRK2 xGEMM	xLARFB	xLASWP xTRSM xGEMM

Table 1: Panel factorization and trailing matrix update routines.

Algorithm 1: Two-phase implementation of a one-sided factorization.

for $P_i \in \{P_1, P_2, ..., P_n\}$ **do**
  PanelFactorize(P_i)
  TrailingMatrixUpdate($A^{(i)}$)

data transfer of the panel to and from the CPU is required at each step of the loop. The classical implementation as described in Algorithm 1 lacks of efficiency because either the CPU or the GPU is working at a time. The MAGMA library modified further the algorithm to overcome this issue and to achieve closer to optimal performance. In fact, the ratio of the computational capability between the CPU and the GPU is orders of magnitude, and thus the common technique to alleviate this imbalance and keep the GPU loaded is to use lookahead.

Algorithm 2: Lookahead of depth 1 for the two-phase factorization.

for $P_i \in \{P_1, P_2, ..., P_n\}$ **do**
  CPU: PanelFactorize(P_i)
  GPU: TrailingMatrixUpdate of only next panel of ($A^{(i)}$ which is P_2)
  CPU and GPU work in parallel: CPU go to the next loop while GPU continue the update
  GPU: continue the TrailingMatrixUpdate of the remaining ($A^{(i-1)}$) using the previous panel (P_{i-1})

Algorithm 2 shows a very simple case of lookahead of depth 1. The update operation is split into an update of the next panel, and an update of the rest of the trailing matrix. The splitting is done to overlap the communication and the factorization of the panel with the update operation. This technique let us hide the memory bound operation of the panel factorization and also keep the GPU loaded by the trailing matrix update.

In the batched implementation, however, we can not afford such a memory transfer at any step, since the trailing matrix is small and the amount of computation is not sufficient to overlap it in time with the panel factorization. Many small data transfers will take away any performance advantage enjoyed by the GPU. In the next section, we describe our proposed implementation and optimization for the batched algorithm.

4 Batched One-Sided Factorizations

The purpose of batched routines is to solve a set of independent problems in parallel. When the matrices are large enough to fully load the device with work, there is no need for batched routines: the set of independent problems can be solved in serial as a sequence of problems. Moreover, it is preferred to solve it in serial, and not in batched fashion, to better enforce locality of data and increase the cache reuse. However, when matrices are small (for example matrices of size less than or equal to 512), the amount of work needed to perform the factorization cannot saturate the device, either CPU or GPU), and thus there is a need for batched routines.

4.1 Batched Factorizations for Multicore CPUs

In broad terms, there are two main ways to approach batched factorization on multicore CPU. The first one is to parallelize each small factorization across all the cores and the second one is to execute each factorization sequentially on a single core with all the cores working independently on their own input data. With these two extremes clearly delineated, it is easy to see the third possibility: the in-between solution where each matrix is partitioned among a handful of cores and multiple matrices are worked on at a time as the total number of available cores permits.

The tall-and-skinny matrix factorization scenarios were studied before [8, 9, 16] which has some relation on batched factorization on multicore CPUs. The problem can either be of reduced size and be fully cache-contained even for Level 1 cache in which case the algorithm becomes compute-bound because the cache can fully satisfy the issue rate of the floating-point units. For our target matrix sizes, the cache containment condition does not hold and, consequently, the most efficient scheme is to employ fixed matrix partitioning schemes with communication based on cache coherency protocols to achieve nearly linear speedup over purely sequential implementation [8, 9, 16]. To our knowledge, this work constitutes nearly optimal implementation scenario that by far exceeds the state-of-the-art vendor and open source implementations currently available. Unfortunately, the bandwidth still remains the ultimate barrier: the achieved performance could be a multiple times better than the next best solution but it is still a fraction of the peak performance of the processor.

For batched operations, the cache partitioning techniques did not work well in our experience because of the small size of matrices which is not the intended target for this kind of optimization. We tested various levels of nested parallelism to exhaust all posiblities

of optimization available on CPUs. The two extremes mentioned above get about 40 Gflop/s (one outer task and all 16 cores working on a single problem at a time – 16-way parallelism for each matrix) and 100 Gflop/s (16 outer tasks with only a single core per task – sequential execution each matrix), respectively. The scenarios that between these extremes achieve somewhere in between in terms of performance. For example, with 8 outer tasks with 2 cores per task we achieve about 50 Gflop/s. Given these results and to increase clarity of the presentation, we only report the extreme setups in the results shown below.

4.2 Batched Factorizations for GPUs

One approach to the batched factorizations problem for GPUs is to consider that the matrices are small enough and to therefore factor them using the non blocked algorithm. The implementation in this case is simple but the performance obtained turns out to be unacceptably low. Thus the implementation of the batched factorization must also be blocked, and thus follow the same iterative scheme (*panel factorization* and *trailing matrix update*) shown in Algorithm 1. Note that the trailing matrix update consists of Level 3 BLAS operations (Xsyrk for Cholesky, Xgemm for LU and Xlarfb for QR) which are compute intensive and thus can perform very well on the GPU. Thus, the most difficult phase of the algorithm is the panel factorization.

A recommended way of writing efficient GPU kernels is to use the GPU's shared memory – load it with data and reuse that data in computations as much as possible. The idea behind this is to do the maximum amount of computation before writing the result back to the main memory. However, the implementation of such technique may be complicated for the small problems considered as it depends on the hardware, the precision, and the algorithm. Moreover, our experience showed that this procedure provides very good performance for simple GPU kernels but is not that appealing for batched algorithm for two main reasons. First, the current size of the shared memory is 48 KB per streaming multiprocessor (SMX) for the newest Nvidia K40 (Kepler) GPUs, which is a low limit for the amount of the batched problems data that can fit at once. Second, completely saturating the shared memory per SMX can decrease the performance of memory bound routines, since only one thread-block will be mapped to that SMX at a time. Indeed, due to a limited parallelism in the factorization of a small panel, the number of threads used in the thread block will be limited, resulting in low occupancy, and subsequently poor core utilization. In our study and analysis we found that redesigning the algorithm to use small amount of shared memory per kernel (less than 10KB) not only provides an acceptable data reuse but also allows many thread-blocks to be executed by the same SMX in parallel, and thus taking a better advantage of its resources. As a results the performance obtained is more than 3× better than the one where the entire shared memory is used. Since the CUDA warp consists of 32 threads, it is recommended to develop CUDA kernels that use multiple of 32 threads per thread-block. For our batched algorithm, we discovered empirically that the best value for nb is 32.

Below we describe our batched routines based on batched BLAS – the way they are implemented, and all the relevant optimizations that have been incorporated in order to achieve performance. All routines are batched and denoted by the corresponding LAPACK routine names. We have implemented them in the four standard floating-point precisions – single real, double real, single complex, and double complex. For convenience, we use the double precision routine name throughout the paper.

4.2.1 Methodology Based on Batched BLAS

In a batched problem solution methodology that is based on batched BLAS, there are many small dense matrices that must be factorized simultaneously (as illustrated in Figure 1). This means that all the matrices will be processed simultaneously by the same kernel. Yet, each matrix problem is still solved independently, identified by a unique batch ID. We follow this model in our batched implementations and developed the following set of new batched CUDA kernels:

- **Cholesky panel:** Provides the batched equivalent of LAPACK's dpotf2 routine. At step j of a panel of size (m,nb), the column vector $A(j:m,j)$ must be computed. This requires a dot-product using row $A(j,1:j)$ to update element $A(j,j)$, followed by a dgemv $A(j+1,1)\ A(j,1:j) = A(j+1:m,j)$, and finally a dscal on column $A(j+1:m,j)$. This routine involves two Level 1 BLAS calls (dot and scal), as well as a Level 2 BLAS dgemv. Since there are nb steps, these routines are called nb times, and thus one can expect that the performance depends on the performances of Level 2 and Level 1 BLAS operations. Hence, it is a slow, memory bound algorithm. We used shared memory to load both row $A(j,1:j)$ and column $A(j+1:m,j)$ to reuse them, and wrote a customized batched dgemv kernel to read and write these vectors from/into the shared memory.
- **LU panel:** This provides the batched equivalent of LAPACK's dgetf2 routine to factorize panels of size $m \times nb$ at each step of the batched LU factorizations. It consists of three Level 1 BLAS calls (idamax, dswap and dscal) and one Level 2 BLAS call (dger). The dgetf2 procedure proceeds as follow: Find the maximum element of the i^{th} column, then swap the i^{th} row with the row owning the maximum, and scale the i^{th} column. To achieve higher performance and minimize the effect on the Level 1 BLAS operation, we implemented a tree reduction to find the maximum where all the threads contributes to find the max. Since it is the same column that is used to find the max then scaled, we load it to the shared memory. This is the only data that we can reuse within one step.
- **QR panel:** This provides the batched equivalent of LAPACK's dgeqr2 routine to perform the Householder panel factorizations. It consists of nb steps where each step calls a sequence of the dlarfg and the dlarf routines. At every step (to compute one column), the dlarfg involves a norm computation followed by a dscal that uses the results of the norm computation in addition to some underflow/overflow checking. The norm computation is a sum reduce and thus a synchronisation step. To accelerate it, we implemented a two-layer tree reduction where for sizes larger than 32, all 32 threads of a warp progress to do a tree reduction similar to the MPI_REDUCE operation, and the latest 32 element are reduced by only one thread. Another optimization is to allow more than one thread-block to execute the dlarfg kernel which means the kernel needs to be split over two – one for norm and one for scaling in order to guarantee the synchronisation. Custom batched implementations of both dlarfg and the dlarf have been developed.
- **Trailing matrix updates:** The trailing matrix updates are mainly Level 3 BLAS operations. However, for small matrices it might be difficult to extract performance from very small Level 3 BLAS kernels. The dgemm is the best Level 3 BLAS kernel – it is GPU friendly, highly optimized, and achieves the highest performance among BLAS. For that, high-performance can be achieved if we redesign our update kernels to be represented by dgemms. For Cholesky, the update consists of the dsyrk routine. It performs a rank-nb update on

either the lower or the upper portion of A_{22}. Since CUBLAS does not provide a batched implementation of this routine, we implemented our own. It is based on a sequence of customized dgemms in order to extract the best possible performance. The trailing matrix update for the Gaussian elimination (LU) is composed of three routines: the dlaswp that swaps the rows on the left and the right of the panel in consideration, followed by the dtrsm to update $A_{12} \leftarrow L_{11}^{-1} A_{12}$, and finally a dgemm for the update $A_{22} \leftarrow A_{22} - A_{21} L_{11}^{-1} A_{12}$. The swap (or pivoting) is required to improve the numerical stability of the Gaussian elimination. However, pivoting can be a performance killer for matrices stored in column major format because rows in that case are not stored continuously in memory, and thus can not be read coalescently. Indeed, a factorization stored in column-major format can be 2× slower (depending on hardware and problem sizes) than implementations that transpose the matrix in order to internally use a row-major storage format [27]. Nevertheless, experiments showed that this conversion is too expensive in the case of batched problems. Moreover, the swapping operations are serial, that is row by row. This limits the parallelism. To minimize this penalty, we propose a new implementation that emphasizes a parallel swap and allows coalescent read/write. We also developed a batched dtrsm. It loads the small $nb \times nb$ L_{11} block into shared memory, inverts it with the dtrtri routine, and then the A_{12} update is accomplished by a dgemm. Generally, computing the inverse of a matrix may suffer from numerical stability, but since A_{11} results from the numerically stable LU with partial pivoting and its size is just $nb \times nb$, or in our case 32×32, we do not have this problem [6]. For the Householder QR decomposition the update operation is referred by the dlarfb routine. We implemented a batched dlarfb that is composed of three calls to the batched dgemm: $A_{22} \leftarrow (I - V T^H V^H) A_{22} \equiv (I - A_{21} T^H A_{21}^H) A_{22}$.

4.3 Techniques for High-Performance Batched Factorizations

4.3.1 Parallel Swapping

Profiling the batched LU reveals that more than 60% of the time is spent in the swapping routine. Figure 2 shows the execution trace of the batched LU for 2,000 matrices of size 512. We can observe on the top trace that the classical dlaswp kernel is the most time consuming part of the algorithm. The swapping consists of nb successive interchanges of two rows of the matrices. The main reason that this kernel is the most time consuming is because the nb row interchanges are performed in a sequential order, and that the data of a row is not coalescent, thus the thread warps do not read/write it in parallel. It is clear that the main bottleneck here is the memory access. Indeed, slow memory accesses compared to high compute capabilities have been a persistent problem for both CPUs and GPUs. CPUs for example alleviate the effect of the long latency operations and bandwidth limitations by using hierarchical caches. Accelerators on the other hand, in addition to hierarchical memories, use thread level parallelism (TLP) where threads are grouped into warps and multiple warps assigned for execution on the same SMX unit. The idea is that when a warp issues an access to the device memory, it stalls until the memory returns a value, while the accelerator's scheduler switches to another warp. In this way, even if some warps stall, others can execute, keeping functional units busy while resolving data dependencies, branch penalties, and long latency memory requests. In order to overcome the bottleneck of swapping, we propose to modify the kernel to apply all nb row swaps in parallel. This modification will also allow the coalescent write back of the top nb rows of the matrix. Note that the first nb rows are those used by the dtrsm kernel that is applied right after the dlaswp, so one optimization is to use shared memory to load a chunk of the nb rows, and apply the dlaswp folloed by the dtrsm at the same time. We changed the algorithm to generate two pivot vectors, where the first vector gives the final destination (e.g. row indices) of the top nb rows of the panel, and the second gives the row indices of the nb rows to swap and bring into the top nb rows of the panel. Figure 2 depicts the execution trace (bottom) when using our parallel dlaswp kernel. The experiment shows that this reduces the time spent in the kernel from 60% to around 10% of the total elapsed time. Note that the colors between the top and the bottom traces do not match each other; this is because the Nvidia profiler puts always the most expensive kernel in green. As a result, the performance gain obtained is about 1.8× as shown by the purple curve of Figure 3. We report each of the proposed optimization for the LU factorization in Figure 3 but we would like to mention that the percentage of improvement obtained for the Cholesky and QR factorization is similar and to simplify we reported the LU factorization only. Note that starting from this version we were able to be faster than the CUBLAS implementation of the batched LU factorization.

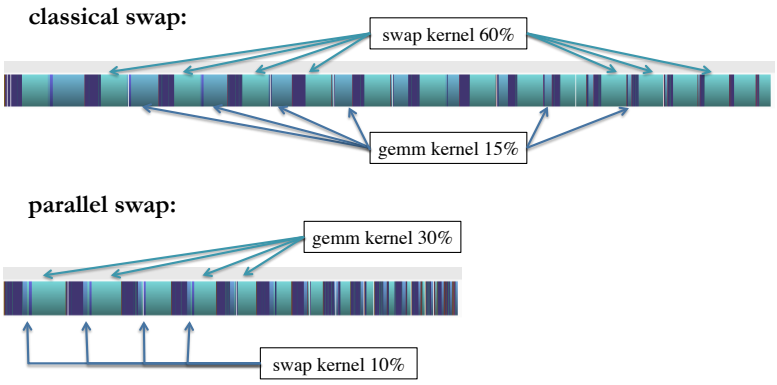

Figure 2: Execution trace of the batched LU factorization using either classical swap (top) or our new parallel swap (bottom).

4.3.2 Recursive Nested Blocking

The panel factorizations described in Section 4.2.1 factorize the nb columns one after another, similarly to the LAPACK algorithm. At each of the nb steps, either a rank-1 update is required to update the vectors to the right of the factorized column i (this operation is done

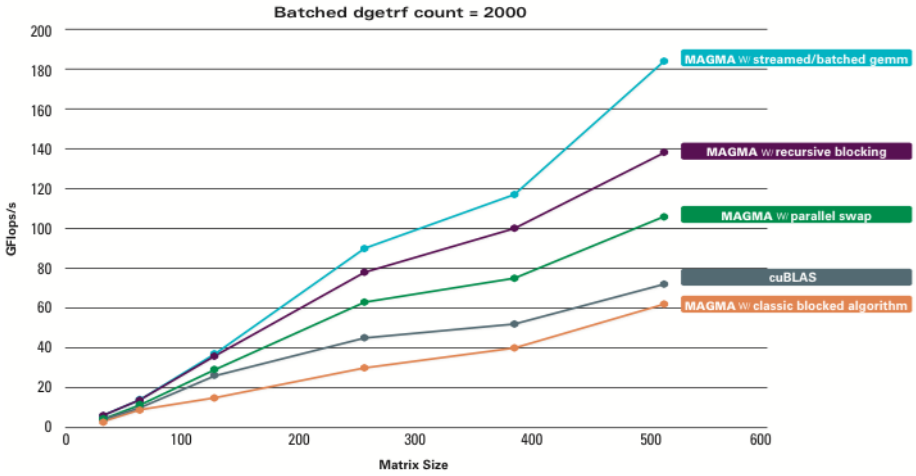

Figure 3: Performance in Gflops/s of the different versions of our batched LU factorizations compared to the CUBLAS implementation for different matrix sizes where $m = n$.

by the dger kernel for LU and the dlarf kernel for QR), or a left looking update of column i by the columns on its left, before factorizing it (this operation is done by dgemv for the Cholesky factorization). Since we cannot load the entire panel into the shared memory of the GPU, the columns to the right (in case of LU and QR) or to the left (in case of Cholesky) are loaded back and forth from the main memory at every step. Thus, one can expect that this is the most time consuming part of the panel factorization. A detailed analysis using the profiler reveals that the dger kernel requires more than 80% and around 40% of the panel time and of the total LU factorization time respectively. Similarly for the QR decomposition, the dlarf kernel used inside the panel computation need 65% and 33% of the panel and the total QR factorization time respectively. Likewise, the dgemv kernel used within the Cholesky panel computation need around 91% and 30% of the panel and the total Cholesky factorization time respectively. This inefficient behavior of these routines is also due to the memory access. For that, to overcomes this bottleneck, we propose to improve the efficiency of the panel and to reduce the memory access by using a recursive level of blocking technique as depicted in Figure 4. In principle, the panel can be blocked recursively until a single element. Yet, in practice, 2-3 blocked levels are sufficient to achieve high performance. The above routines must be optimized for each blocked level, which complicates the implementation. More than 30% boost in performance is obtained by this optimization, as demonstrated in Figure 3 for the LU factorization. The same trend has been observed for both the Cholesky and the QR factorization.

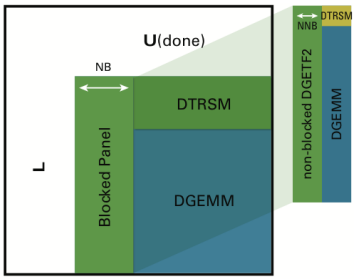

Figure 4: Recursive nested blocking

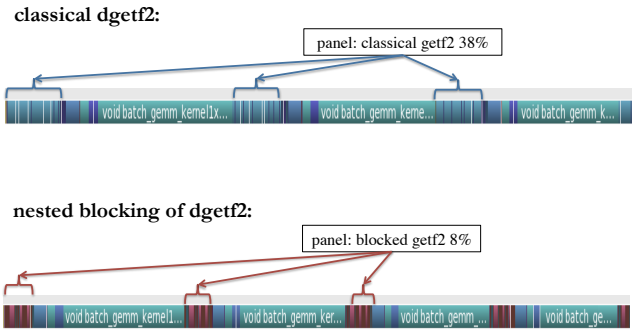

Figure 5: Execution trace of the batched LU factorization using either classical getf2 (top) or our recursive getf2 (bottom).

4.3.3 Trading Extra Computation for Higher Performance

The challenge discussed here is the following: for batched problems there is a need to minimize the use of low performance kernels on the GPU even if they are Level 3 BLAS. For the Cholesky factorization this concerns the dsyrk routine that is used to update the trailing matrix. The performance of dsyrk is important to the overall performance, since it takes a big part of the run-time. We implemented the batched dsyrk routine as a sequence of dgemm routines, each of size $M = m, N = K = nb$. In order to exclusively utilize the dgemm kernel, our implementation writes both the lower and the upper portion of the $nb \times nb$ diagonal blocks of the trailing matrix. This results in nb^3 extra operations for the diagonal block. However, since nb is small (e.g., $nb = 32$) these extra operations can be considered free. In practice the extra operation allows us to use dgemm and thus achieve higher performance than the one that touches the lower/upper portion of the $nb \times nb$ diagonal blocks. Tests show that our implementation of dsyrk is twice faster than the dgemm kernel for the same matrix size. This shows that our dsyrk is very well optimized in order to reach the performance of dgemm (which is twice slower as it computes twice more flops).

We applied the same technique in the dlarfb routine used by the QR decomposition. The QR trailing matrix update uses the dlarfb routine to perform $A_{22} = (I - VT^H V^H)A_{22} = (I - A_{21}T^H A_{21}^H)A_{22}$. The upper triangle of V is zero with ones on the diagonal. In the clasical dlarfb what is available is A_{21} that stores V in its lower triangular part and R (part of the upper A) in its upper triangular part. Therefore, the above is computed using dtrmm for the upper part of A_{21} and dgemm for the lower part. Also, the T matrix is an upper triangular and therefore the classical dlarfb implementation uses dtrmm to perform the multiplication with T. Thus, if one can guarantee that the lower portion of T is filled with zeroes and the upper portion of V is filled zeros and ones on the diagonal, the dtrmm can be replaced by dgemm. Thus we implemented a batched dlarfb that uses three dgemm kernels by initializing the lower portion of T with zeros, and filling up the upper portion of V with zeroes and ones on the diagonal. Note that this brings $3nb^3$ extra operations, but again, the overall time spent in the new dlarfb update using the extra computation is around 10% less than the one using the dtrmm.

Similarly to dsyrk and dlarfb, we implemented the batched dtrsm (that solves $AX = B$) by inverting the small $nb \times nb$ block A and using dgemm to get the final results $X = A^{-1}B$.

4.3.4 Block Recursive dlarft Algorithm

The dlarft is used to compute the upper triangular matrix T that is needed by the QR factorization in order to update either the trailing matrix or the right hand side of the recursive portion of the QR panel. The classical LAPACK computes T column by column in a loop over the nb columns as described in Algorithm 3. Such implementation takes up to 50% of the total QR factorization time. This is due to the fact that the kernels needed – dgemv and dtrmv – require implementations where threads go through the matrix in different directions (horizontal vs. vertical, respectively). An analysis of the mathematical formula of computing T allowed us to redesign the algorithm to use Level 3 BLAS and to increase the data reuse by putting the column of T in shared memory. One can observe that the loop can be split into two loops – one for dgemv and one for dtrmv. The dgemv loop that computes each column of $\widehat{T}$ can be replaced by one dgemm to compute all the columns of $\widehat{T}$ if the triangular upper portion of A is zero and the diagonal is made of ones. For our implementation that is already needed for the trailing matrix update in order to use dgemm in the dlarfb, and thus can be exploited here as well. For the dtrmv phase, we load the T matrix into shared memory as this allows all threads to read/write from/into shared memory during the nb steps of the loop. The redesign of this routine is depicted in Algorithm 4. Since we developed recursive blocking algorithm, we have to compute the T matrix for every level of the recursion. Nevertheless, the analysis of Algorithm 4 let us conclude that the portion of the Ts computed in the lower recursion level are the same as the diagonal blocks of the T of the upper level (yellow diagonal blocks in Figure 6), and thus we can avoid their (re-)computation. For that we modified Algorithm 4 in order to compute either the whole T or the upper rectangular portion that is missed (red/yellow portions in Figure 6).

Algorithm 3: Classical implementation of the dlarft routine.

for $j \in \{1,2,...,nb\}$ **do**
- dgemv to compute $\widehat{T}_{1:j-1,j} = A^H_{j:m,1:j-1} \times A_{j:m,j}$
- dtrmv to compute $T_{1:j-1,j} = T_{1:j-1,1:j-1} \times \widehat{T}_{1:j-1,j}$
- $T(j,j) = tau(j)$

Algorithm 4: Block recursive dlarft routine.

dgemm to compute $\widehat{T}_{1:nb,1:nb} = A^H_{1:m,1:nb} \times A_{1:m,1:nb}$

load $\widehat{T}_{1:nb,1:nb}$ to the shared memory. **for** $j \in \{1,2,...,nb\}$ **do**
- dtrmv to compute $T_{1:j-1,j} = T_{1:j-1,1:j-1} \times \widehat{T}_{1:j-1,j}$
- $T(j,j) = tau(j)$

write back T to the main memory.

4.4 Streamed dgemm

As our main goal is to achieve higher performance, we performed deep analysis of every kernel of the algorithm. We discovered that 70% of the time is spent in the batched dgemm kernel after the previously described optimizations were applied. An evaluation of the

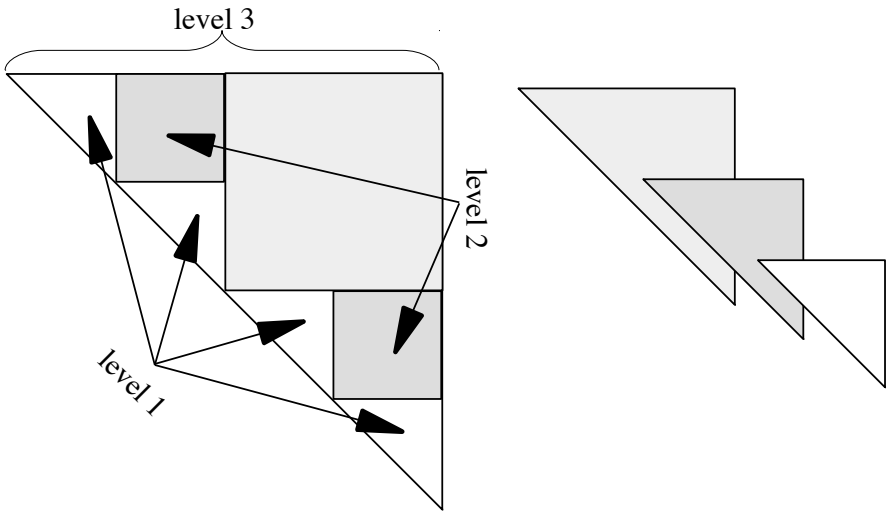

Figure 6: The shape of the matrix T for different level of the recursion during the QR decomposition.

performance of the dgemm kernel using either batched or streamed dgemm is illustrated in Figure 7. The curves let us conclude that the streamed dgemm is performing better than the batched one for some cases, e.g., for $k = 32$ when the matrix size is of order $m > 200$ and $n > 200$. We note that the performance of the batched dgemm is stable and does not dependent on k, in the sense that the difference in performance between $k = 32$ and $k = 128$ is minor. However it is bound by 300 Gflop/s. For that we propose to use the streamed dgemm whenever is faster, and to roll back to the batched one otherwise. Figure 8 shows the trace of the batched LU factorization of 2,000 matrices of size 512 using either the batched dgemm (top trace) or the combined streamed/batched dgemm (bottom trace). We can see that the use of the streamed dgemm (when the size allows it) can speed up the factorization by about 20%.

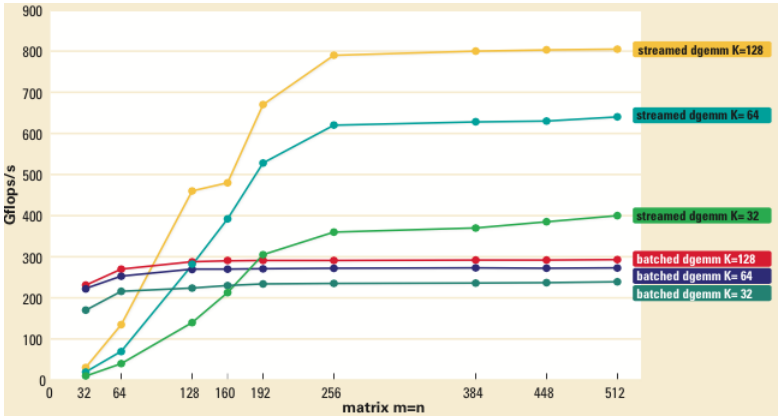

Figure 7: Performance comparison between the streamed and the batched dgemm kernel for different value of K and different matrix sizes where $m = n$.

5 Performance Results

5.1 Hardware Description and Setup

We conducted our experiments on Intel multicore system with two 8-cores socket Intel Xeon E5-2670 (Sandy Bridge) processors, each running at 2.6 GHz. Each socket had 20 MB of shared L3 cache, and each core had a private 256 KB L2 and 64 KB L1 cache. The system is equipped with 52 GB of memory and the theoretical peak in double precision is 20.8 Gflop/s per core, i.e., 332.8 Glop/s in total for the two sockets. It is also equipped with a NVIDIA K40c cards with 11.6 GB of GDDR memory per card running at 825 MHz. The theoretical peak in double precision is 1,689.6 Gflop/s. The cards are connected to the host via two PCIe I/O hubs with 6 GB/s bandwidth.

A number of software packages were used for the experiments. On the CPU side, we used the MKL (Math Kernel Library) [14] with the icc compiler (version 2013.sp1.2.144) and on the GPU accelerator we used CUDA version 6.0.37.

Related to power, we note that in this particular setup the CPU and the GPU have about the same theoretical power draw. In particular, the Thermal Design Power (TDP) of the Intel Sandy Bridge is 115 W per socket, or 230 W in total, while the TDP of the K40c GPU is 235 W. Therefore, we roughly expect that a GPU would have a power consumption advantage if it outperforms (in terms of time to solution)

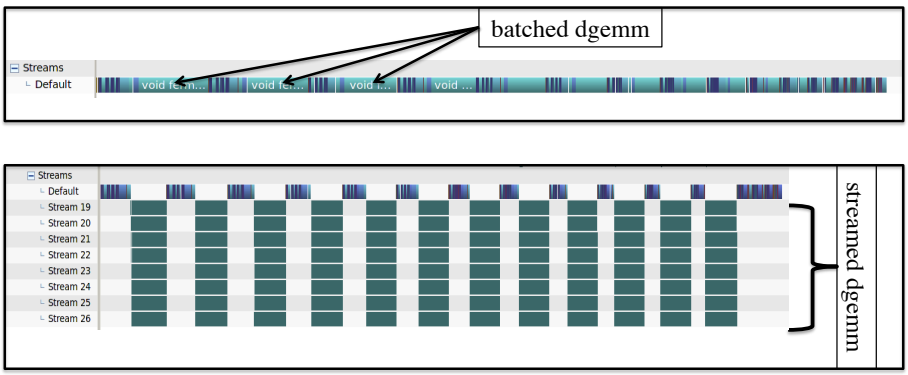

Figure 8: Execution trace of the batched LU factorization using either batched dgemm (top) or streamed/batched dgemm (bottom).

the 16 Sandy Bridge cores. Note that based on the theoretical peaks the GPU's advantage should be about 4 to 5×. This is observed in practice as well, especially for regular workloads on large data-parallel problems that can be efficiently implemented for GPUs.

5.2 Performance Analysis

The performance of the non-blocked versions can be bounded by the performance of the rank-1 update. Its Flops/Bytes ratio for double precision numbers is $\frac{3n}{16+16n}$ (for $m=n$). Therefore, a top performance for $n=500$ and read/write achievable bandwidth of 160 GFlop/s would be $160\times\frac{3\times500}{16+16\times500}=29.9$ GFlop/s. This shows for example that our non-blocking LU from Figure 3 achieves this theoretically best performance. This is the limit for the other non-blocking one-sided factorizations as well.

Similar analysis for a best expected performance can be done for the block algorithms as well. Their upper performances are bounded in general by the rank-nb performance, e.g., illustrated in Figure 7 for $nb=32$ and 64. Although we do not reach the asymptotic performance of 400 GFlop/s at $n=500$, as we show, our performances grow steadily with n, indicating that the $O(n^2)$ flops besides the rank-nb are slow and n needs to grow in order for their influence on the performance to become less significant compared to the rank-nb's $O(n^3)$ flops.

5.3 Comparison with CUBLAS on a K40c

Getting high performance across accelerators remains a challenging problem that we address with the algorithmic and programming techniques described in this paper. The efficient strategies used exploit parallelism and increase the use of Level 3 BLAS operations across the GPU. We highlighted this through a set of experiments that we performed on our system. We compare our batched implementations with the CUBLAS [20] library whenever possible (CUBLAS features only a dgetrfBatched routine). Our experiments were performed on batches of 2,000 matrices of different sizes going from 32×32 to 512×512.

Figure 9 shows the performance of the LU factorization. The dgetrfBatched version, marked as "CUBLAS", reaches a performance of around 70 Gflop/s for matrices size of 512×512. We first compare to a naive implementation that is based on the assumption that matrices of size (<512) are very small for block algorithms, and therefore uses the non blocked version. For example, for the case of LU this is the dgetf2 routine. The routine is very slow and the performance obtained reaches less than 30 Gflop/s, as shown in Figure 3. Note that although low, this is also the optimal performance achievable by this type of algorithms, as explained in Section 5.2.

Our second comparison is to the *classic* LU factorization, i.e., the one that follows LAPACK's two-phases implementation described in Algorithm 1. This algorithm achieves 63 Gflop/s as shown in Figure 3.

To reach beyond 100 Gflop/s, we used the technique that optimizes pivoting with *parallel swap*. Next step in performance improvement was the use of two-level blocking of the panel, which enables performance levels that go slightly above 130 Gflop/s. The final two improvements are *streamed/batched gemm*, which moves the performance beyond 160 Gflop/s, and finally, the *two-levels blocking update*, (also what we called *recursive blocking*) completes the set of optimizations and takes the performance beyond 180 Gflop/s. Thus our batched LU achieves up to 2.5× speedup compared to its counterpart from the CUBLAS library.

5.4 Comparison to Multicore CPU Solutions

Here we compare our batched LU to the two CPU implementations proposed in Section 4.1. The simple CPU implementation is to go in a loop style to factorize matrix after matrix, where each factorization is using the multi-thread version of the MKL Library. This implementation is limited in terms of performance and does not achieve more than 50 Gflop/s. The main reason for this low performance is the fact that the matrix is small – it does not exhibit parallelism and so the multithreaded code is not able to feed with work all 16 threads used. For that we proposed another version of the CPU implementation. Since the matrices are small (<512) and at least 16 of them fit in the L3 cache level, one of the best technique is to use each thread to factorize independently a matrix. This way 16 factorizations are conducted independently in parallel. We think that this implementation is one of the best optimized implementations for the CPU. This later implementation is twice faster than the simple implementation. It reaches around 100 Gflop/s in factorizing 2,000 matrices of size 512×512. Experiments show that our GPU batched LU factorization is able to achieve a speedup of 1.8× *vs.* the best CPU implementation using 16 Sandy Bridge cores, and 4× *vs.* the simple one.

The performances obtained for the Cholesky and QR factorizations are similar to the results for LU. A comparison againts the two CPU implementations for Cholesky and QR are given in Figures 10 and 11, respectively.

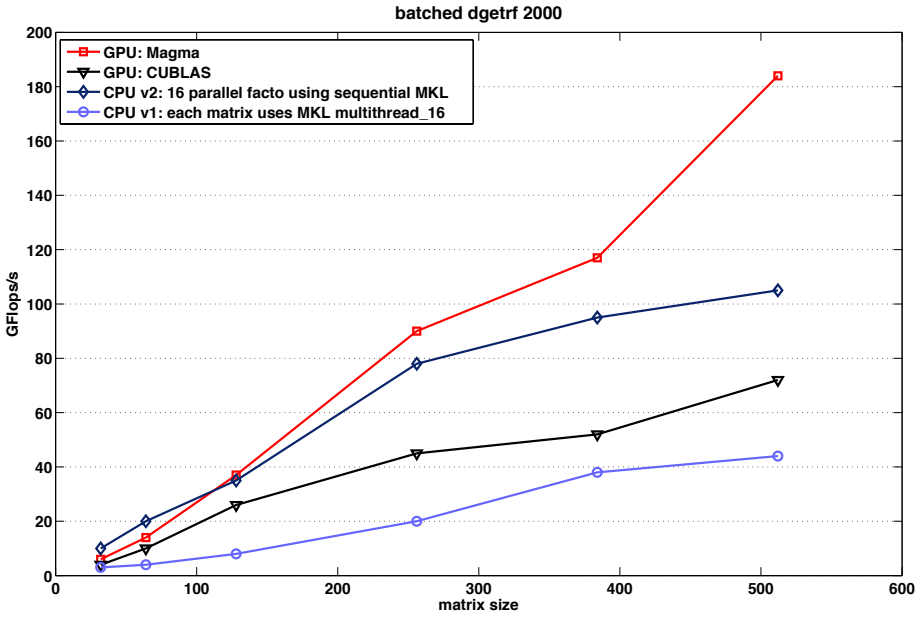

Figure 9: Performance in Gflops/s of our different version of the batched LU factorization compared to the CUBLAS implementation for different matrix sizes where $m = n$.

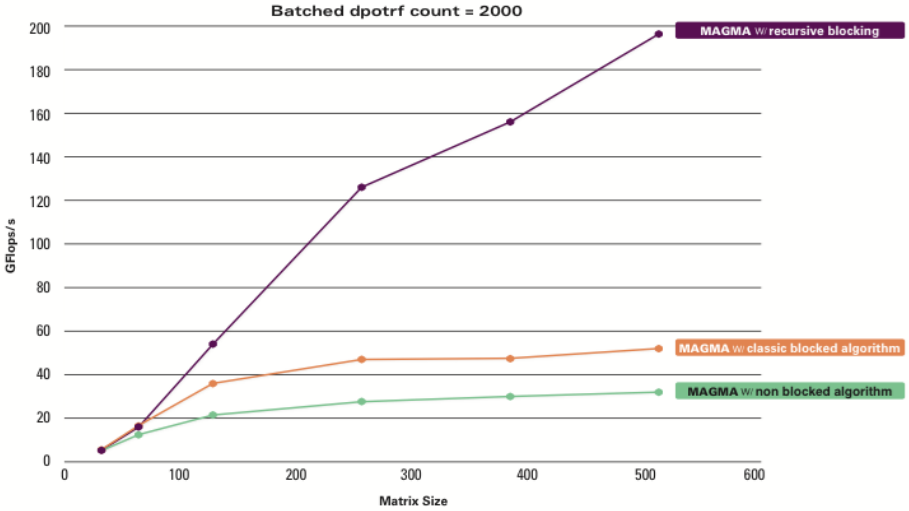

Figure 10: Performance in Gflops/s of the GPU *vs.* the CPU versions of our batched Cholesky factorizations for different matrix sizes where $m = n$.

Similarly to the LU, our first GPU implementation of the batched Cholesky factorization follows the classical LAPACK implementation. Compared to the non-blocking algorithm this version increases the use of shared memory and attains at $n = 500$ an upper bound of 60 Gflop/s. The different optimization techniques from Section 4.3 drive the performance of the Cholesky factorization up to 200 Gflop/s. The two CPU implementations behave similarly to the ones for LU. The simple CPU implementation achieves around 60 Gflop/s while the optimized one reaches 100 Gflop/s. This yields a speedup of 2× against the best CPU implementation using 16 Sandy Bridge cores.

The progress of our batched QR implementation over the different optimizations shows the same behavior. The classical block implementation does not attain more than 50 Gflop/s. The recursive blocking improves performance up to 105 Gflop/s, and the optimized computation of T draws it up to 127. The other optimizations (replacing dtrmm by dgemm in both dlarft and dlarfb), combined with the streamed/batched dgemm bring the GPU implementation to around 167 Gflop/s. The simple CPU implementation of the QR decomposition does not attain more than 50 Gflop/s while the optimized one gets 100 Gflop/s. Despite the CPU's hierarchical memory advantage, our GPU batched implementation is about 1.7× faster.

5.5 Energy Efficiency

For our energy efficiency measurements we use power and energy estimators built into the modern hardware platforms. In particular, on the tested CPU, Intel Xeon E5-2690, we use RAPL (Runtime Average Power Limiting) hardware counters [15, 25]. By the vendor's own admission, the reported power/energy numbers are based on a model which is tuned to match well the actual measurements for various

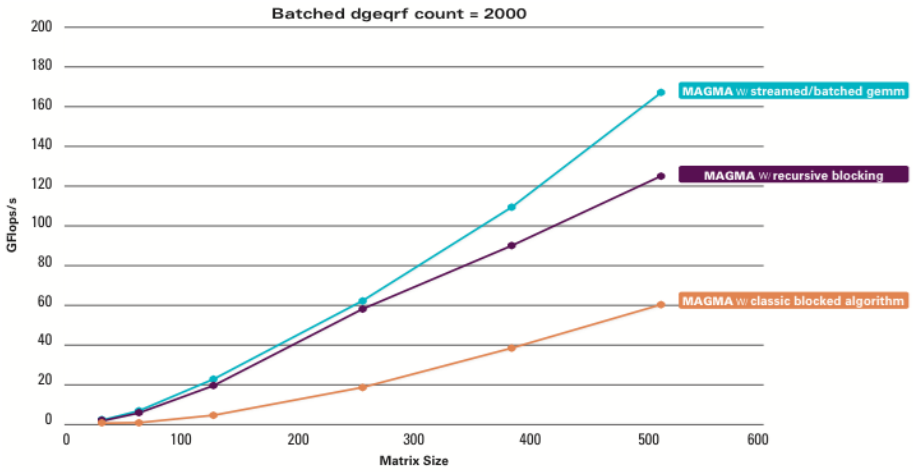

Figure 11: Performance in Gflops/s of the GPU *vs.* the CPU versions of our batched QR decomposition for different matrix sizes where $m = n$.

workloads. Given this caveat, we can report that the idle power of the tested Sandy Bridge CPU, running at fixed frequency of 2600 MHz, consumes about 20 W of power per socket. Batched operations raise the consumption to above 125 W-140 W per socket and the large dense matrix operations, that reach the highest fraction of the peak performance, raise the power draw to about 160 W per socket.

For the GPU measurements we use NVIDIA's NVML (NVIDIA Management Library) library [19]. NVML provides a C-based programmatic interface for monitoring and managing various states within NVIDIA Tesla GPUs. On Fermi and Kepler GPUs (like the K40c used) the readings are reported to be accurate to within +/-5% of current power draw. The idle state of the K40c GPU consumes about 20 W. Batched factorizations raise the consumption to about 150 – 180 W, while large dense matrix operations raise the power draw to about 200 W.

We depict in Figure 12 the comparison of the power consumption required by the three implementations of the batched QR decomposition: the best GPU and the two CPU implementations. The problem solved here is about 4,000 matrices of size 512×512 each. The green curve shows the power required by the simple CPU implementation. In this case the batched QR proceeds as a loop over the 4,000 matrices where each matrix is factorized using the multithreaded dgeqrf routine form the Intel MKL library on the 16 Sandy Bridge cores. The blue curve shows the power required by the optimized CPU implementation. Here, the code proceeds by sweep of 16 parallel factorizations each using the sequential dgeqrf routine form the Intel MKL library. The red curve shows the power consumption of our GPU implementation of the batched QR decomposition. One can observe that the GPU implementation is attractive because it is around 2× faster than the optimized CPU implementation, and moreover, because it consumes 3× less energy.

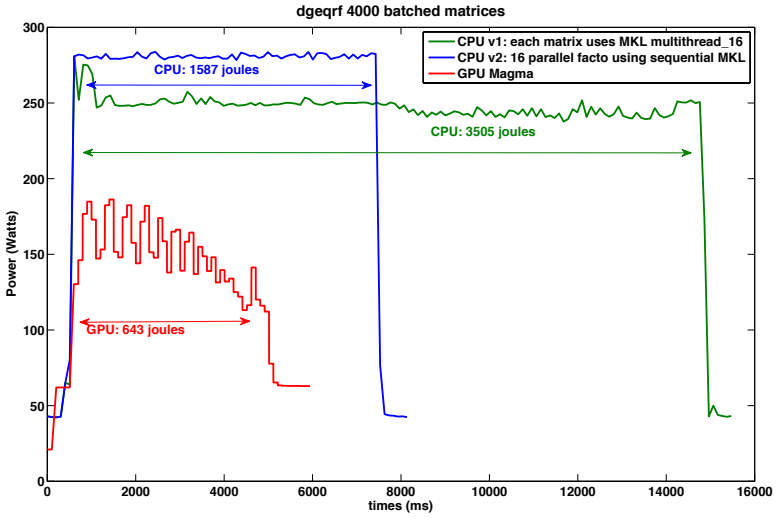

Figure 12: Comparison of the power consumption for the QR decomposition of 4,000 matrices of size 512×512.

According to the experiments we conduct to measure the power we found that the GPU implementations of all of the batched one-sided factorizations reach around 2× speedup over their best CPU counterpart and are 3× less expensive in term of energy.

6 Conclusions and future directions

Designing algorithms to work on small problems is a concept that can deliver higher performance through an improved data reuse. Many applications have relied on this design concept to get higher hardware efficiency, and users have requested it as a supported functionality in linear algebra libraries. Besides having the potential to improve the overall performance of applications with computational patterns ranging from dense to sparse linear algebra, developing these algorithms for the new low-powered and power-efficient architectures can bring significant savings in energy consumption. We demonstrated how to accomplish this in the case of batched dense solvers for GPU architectures.

We showed that efficient batched dense solvers can be implemented relatively easily for multicore CPUs, relying on existing high-performance libraries like MKL for building blocks. For GPUs, on the other hand, the development is not straightforward. Our literature review pointed out that the pre-existing solutions were either just memory-bound, or even if highly optimized, did not exceed in performance the corresponding CPU versions (if they are highly optimized as the ones developed in this work and use a number of cores scaled to ensure the same CPUs power draw as a GPU). We demonstrated that GPUs, with proper algorithmic enhancements and with the batched BLAS approach used, can have an advantage over CPUs on this workload. In particular, the algorithmic work on blocking, variations of blocking like in the recursive nested blocking, adding extra flops to improve parallelism and regularity of the computation, streaming, and other batched/algorithm-specific improvements as in the LU's parallel swapping, contributed most in enabling the GPUs to outperform the CPUs on a workload that was previously favored for execution on multicore CPU architectures due to their larger cache sizes and well developed memory hierarchy.

To illustrate the improvements, we compared the results obtained on current high-end GPUs and CPUs. In particular, we considered a single NVIDIA K40c GPU *vs.* two Intel Sandy Bridge CPUs (16 cores in total) as this configuration has the same accumulative power draw on the two systems. While the power draw is the same (around 240 W), the GPU has about 4× higher theoretical performance peak, and therefore is expected to have around $3-4\times$ advantage in both performance and energy efficiency. Indeed, improvements like these have been observed on large classical numerical algorithms in both dense and sparse linear algebra, where efficient GPU implementations are possible. In this paper, we demonstrated that one can take advantage of the GPUs for small batched linear solvers as well. In particular, we achieved around 2× speedup compared to our optimized CPU implementations and 3× better energy efficiency.

As the development of efficient small problem solvers gets more intricate on new architectures, we envision that users will further demand their availability in high-performance numerical libraries, and that batched solvers will actually become a standard feature in those libraries for new architectures. Our plans are to release and maintain this new functionality through the MAGMA libraries for NVIDIA GPU accelerators, Intel Xeon Phi coprocessors, and OpenCL with optimizations for AMD GPUs.

The batched algorithms and techniques can be used and extended to develop totally GPU implementations for stand-alone linear algebra problems. These would be useful, for example, to replace the hybrid CPU-GPU algorithms in cases where energy consumption, instead of higher-performance through use of all available hardware resources, is the top priority. Moreover, totally GPU implementations can have a performance advantage as well, when the host CPU becomes slower compared to the accelerator in future systems. For example, in mobile devices featuring ARM processors enhanced with GPUs, like the Jetson TK1, we have already observed that the totally GPU implementations have a significant advantage in both energy consumption and performance. This has motivated another future work direction – the development and release of a *MAGMA Embedded* library that would incorporate entirely GPU/coprocessor implementations for stand-alone, as well as batched, dense linear algebra problems.

Acknowledgment

The results were obtained in part with the financial support of the Russian Scientific Fund, Agreement N14-11-00190.

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

References

[1] 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.

[2] 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.

[3] ACML - AMD Core Math Library, 2014. Available at http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml.

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

[5] M. Anderson, D. Sheffield, and K. Keutzer. A predictive model for solving small linear algebra problems in gpu registers. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, 2012.

[6] D. Croz, J. J. Dongarra, and N. J. Higham. Stability of methods for matrix inversion. *IMA J. Numer. Anal.*, 12(119), 1992.

[7] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014.

[8] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Exploiting fine-grain parallelism in recursive LU factorization. In *ParCo 2011 – International Conference on Parallel Computing*, Ghent, Belgium, August 30-September 2 2011.

[9] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Exploiting fine-grain parallelism in recursive LU factorization. *Advances in Parallel Computing, Special Issue*, 22:429–436, 2012. ISBN 978-1-61499-040-6 (print); ISBN 978-1-61499-041-3 (online).

[10] 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.

[11] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, Feb. 2004.

[12] Matrix algebra on GPU and multicore architectures (MAGMA), 2014. Available at http://icl.cs.utk.edu/magma/.

[13] Intel Pentium III Processor - Small Matrix Library, 1999. Available at http://www.intel.com/design/pentiumiii/sml/.

[14] Intel Math Kernel Library, 2014. Available at http://software.intel.com/intel-mkl/.

[15] Intel® 64 and IA-32 architectures software developer's manual, July 20 2014. Available at http://download.intel.com/products/processor/manual/.

[16] P. Luszczek and J. Dongarra. Anatomy of a globally recursive embedded LINPACK benchmark. In *Proceedings of 2012 IEEE High Performance Extreme Computing Conference (HPEC 2012)*, Westin Hotel, Waltham, Massachusetts, September 10-12 2012. IEEE Catalog Number: CFP12HPE-CDR, ISBN: 978-1-4673-1574-6.

[17] O. Messer, J. Harris, S. Parete-Koon, and M. Chertkow. Multicore and accelerator development for a leadership-class stellar astrophysics code. In *Proceedings of "PARA 2012: State-of-the-Art in Scientific and Parallel Computing."*, 2012.

[18] J. Molero, E. Garzón, I. García, E. Quintana-Ortí, and A. Plaza. Poster: A batched Cholesky solver for local RX anomaly detection on GPUs, 2013. PUMPS.

[19] Available at https://developer.nvidia.com/nvidia-management-library-nvml, 2014.

[20] CUBLAS, 2014. Available at http://docs.nvidia.com/cuda/cublas/.

[21] The OpenACC™ application programming interface version 1.0, November 2011.

[22] OpenMP application program interface, July 2013. Version 4.0.

[23] 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*, volume 8097 of *Lecture Notes in Computer Science*, pages 813–825, Aachen, Germany, August 26-30 2013.

[24] V. Oreste, N. A. Gawande, and A. Tumeo. Accelerating subsurface transport simulation on heterogeneous clusters. In *IEEE International Conference on Cluster Computing (CLUSTER 2013)*, Indianapolis, Indiana, September, 23-27 2013.

[25] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, March/April 2012. ISSN: 0272-1732, 10.1109/MM.2012.12.

[26] S. Tomov, R. Nath, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, Atlanta, GA, April 19-23 2014.

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

[28] I. Wainwright. Optimized LU-decomposition with full pivot for small batched matrices, April, 2013. GTC'13 – ID S3069.

[29] S. N. Yeralan, T. A. Davis, and S. Ranka. Sparse mulitfrontal QR on the GPU. Technical report, University of Florida Technical Report, 2013.