

# Framework for Batched and GPU-resident Factorization Algorithms Applied to Block Householder Transformations

Azzam Haidar<sup>1</sup>, Tingxing “Tim” Dong<sup>1</sup>, Stanimire Tomov<sup>1</sup>, Piotr Luszczek<sup>1</sup>, and Jack Dongarra<sup>1,2,3</sup>

<sup>1</sup> University of Tennessee, USA

<sup>2</sup> Oak Ridge National Laboratory, USA

<sup>3</sup> University of Manchester, UK

**Abstract.** As modern hardware keeps evolving, an increasingly effective approach to develop energy efficient and high-performance solvers is to design them to work on many small size and independent problems. Many applications already need this functionality, especially for GPUs, which are currently known to be about four to five times more energy efficient than multicore CPUs. We describe the development of the main one-sided factorizations that work for a set of small dense matrices in parallel, and we illustrate our techniques on the QR factorization based on Householder transformations. We refer to this mode of operation as a *batched factorization*. Our approach is based on representing the algorithms as a sequence of batched BLAS routines for GPU-only execution. The hybrid CPU-GPU algorithms rely heavily on using the multicore CPU for specific part of the workload. But in order to benefit from the GPU’s significantly higher energy efficiency, the primary design goal is to avoid the use of the multicore CPU and to exclusively rely on the GPU. Additionally, this will result in the removal of the costly CPU-to-GPU communication. Furthermore, we do not use a single symmetric multiprocessor (on the GPU) to factorize a single problem at a time. We illustrate how our performance analysis and the use of profiling and tracing tools guided the development and optimization of batched factorization to achieve up to 2-fold speedup and 3-fold better energy efficiency compared to our highly optimized batched CPU implementations based on the MKL library (when using two sockets of Intel Sandy Bridge CPUs). Compared to a batched QR factorization featured in the CUBLAS library for GPUs, we achieved up to 5 speedup on the K40 GPU.

## 1 Introduction

Accelerators and coprocessors have enjoyed widespread adoption in computational science, consistently producing many-fold speedups across wide range of scientific disciplines and important applications [13]. The typical method of utilizing the GPU accelerators is to increase the scale and resolution of an application, which in turn increases its computational intensity; this tends to be a good match for the steady growth in performance and memory capacity of this type of hardware.

Unfortunately, there are many important application types for which the standard approach turns out to be a poor strategy for improving hardware utilization<sup>4</sup>. Numerous modern applications tend to be cast in terms of a solution of *many small matrix operations*, e.g., computations that require tensor contraction such as quantum Hall effect, astrophysics [28], metabolic networks [26], CFD and resulting PDEs through direct and multifrontal solvers [45], high-order FEM schemes for hydrodynamics [10], direct-iterative preconditioned solvers [20], and some image [29] and signal processing [5]. That is, at some point in their execution, such programs must perform a computation that is cumulatively very large, but whose individual parts are very small; these parts cannot be efficiently mapped as separate individuals on to the modern accelerator hardware. Under these circumstances, the only way to achieve good performance is to find a way to group these small inputs together and run them in large “batches.”

The emergence of large-scale, heterogeneous systems with GPU accelerators and coprocessors has made the *near total absence of linear algebra software for such small matrix operations* especially noticeable. Due to the high levels of parallelism they support, accelerators or coprocessors, like GPUs, efficiently achieve very high performance on large data parallel computations, so they have often been used in combination with CPUs, where the CPU handles the small and difficult to parallelize tasks. Moreover, by using efficient batched codes, small problems can be solved on GPUs with four to five times more energy efficiency than one can get from multicore CPUs alone. For both of these reasons, and given the fundamental importance of numerical libraries to science and engineering applications of all types [25], the need for software that can perform batched operations on small matrices is acute. This work is towards filling this critical gap, both by providing a library that addresses a significant range of small matrix problems, and by driving progress toward a standard interface that would allow the entire linear algebra (LA) community to attack them together.

To better understand the problem, consider Figure 1 that shows a simple batched computation that factorizes a sequence of small matrices  $A_i$ . The word *small* is used in relative terms as the beneficial size of  $A_i$  will depend on the circumstances. A straightforward guideline to determine this size is the ability of processing the matrices in parallel rather than sequentially. To achieve this goal it is necessary to co-locate a *batch* of  $A_i$  in a fast GPU store – a cache or shared memory – and process it there with a better use of parallel execution units.

Now this kind of optimization cannot be made by the compiler alone for two primary reasons: the lack of standardized interfaces and the opaque implementation of the factorization routine. The former derives from the fact that, until recently,

```

for  $A_i \in A_1, A_2, \dots, A_k$  do
  | GenerateSmallLinearSystem( $A_i$ )
for  $A_i \in A_1, A_2, \dots, A_k$  do
  | Factorize( $A_i$ )

```

Fig. 1. Batched computation example.

<sup>4</sup> Historically, similar issues were associated with strong scaling [14] and were attributed to a fixed problem size.

batched computations were not the primary bottleneck for scientific codes because it was large size problems that posed a performance challenge. Once appreciable increases in the processing power and memory capacity of GPUs removed this bottleneck, the small size problems became prominent in the execution profile because they significantly increased the total execution time. The latter is a simple consequence of separation of concerns in the software engineering process whereby the computational kernels are packaged as standalone modules that are highly optimized and cannot be inlined into the batched loop in Figure 1. What we propose is to define the appropriate interfaces so that our implementation can seamlessly work with the compiler and use the *code replacement* technique so that the user has an option of expressing the computation as the loop shown in the figure or a single call to a routine from the new standard batch library.

Against this background, the goal of this work is two-fold: first, to deliver a high-performance numerical library for batched linear algebra subroutines tuned for the modern processor architecture and system designs. The library must include LAPACK routine equivalents for many small dense problems as well as routines for many small sparse matrix solvers, that must be constructed as much as possible out of calls to batched BLAS routines and their look-alikes required in sparse computation context. Second, and just as importantly, it must define modular interfaces so that our implementation can seamlessly work with the compiler and use *code replacement* techniques. This will provide the developers of applications, compilers, and runtime systems with the option of expressing computation as a loop (as shown in the figure), or a single call to a routine from the new batch operation standard.

As might be expected, batched BLAS form the foundation of the framework for batched algorithms proposed, with other tasks building up in layers above. The goal of this approach is to achieve portability across various architectures, sustainability and ease of maintenance, as well as modularity in building up the framework's software stack. In particular, on top of the batched BLAS (by algorithmic design), we illustrate the building of a batched LAPACK. The new batched algorithms are implemented and currently released through the MAGMA 1.6.1 library [21]. The framework will allow for future work extension to batched sparse linear algebra, and application-specific batched linear algebra. Finally, all the components are wrapped up in a performance and energy autotuning framework.

In terms of the framework's sustainability, it is important to note that batched operations represent the next generation of software that will be required to efficiently execute scientific compute kernels on self-hosted accelerators that do not have accompanying CPUs such as the next generation of Intel Xeon Phi processors, and on accelerators with a very weak host CPUs, e.g., various AMD APU models and NVIDIA Tegra platforms. For such hardware, performing any non-trivial work on the host CPU would slow down the accelerator dramatically, making it essential to develop new batched routines as a basis for optimized routines for accelerator-only execution.

## 2 Related work

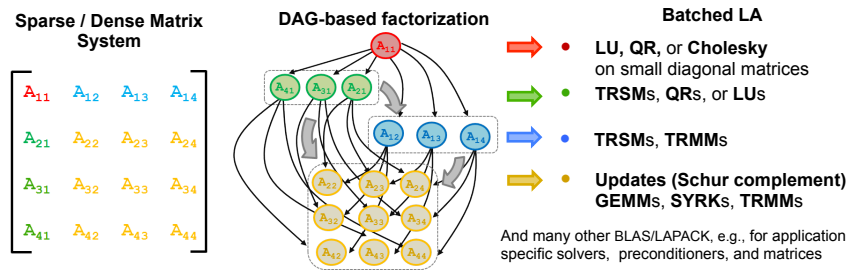
There is a lack of numerical libraries that cover the functionalities of batched computation for GPU accelerators and coprocessors. NVIDIA started to add certain batch functions in their math libraries; NVIDIA’s CUBLAS 6.5 [36] includes batched Level 3 BLAS for `gemm` and `trsm` (triangular matrix solver), the higher-level (LAPACK) LU and QR factorizations, matrix inversion, and a least squares solver. All of these routines are for uniform size matrices. AMD and Intel’s MKL do not provide batched operations yet. For higher-level routines, NVIDIA provides four highly-optimized LAPACK-based routines, but they do not address the variable sizes, extended functionality, portability and device-specific redesigns of the LAPACK algorithms. Our initial proof-of-concept work shows the potential of addressing these issues, e.g., as illustrated in this paper by our  $3\times$  speedup compared to the batch-optimized QR in CUBLAS.

Batched LA ideas can be applied to multicore CPUs as well. Indeed, small problems can be solved efficiently on single core, e.g., using vendor supplied libraries such as MKL [23] or ACML [4], 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 can be added to use SIMD processor supplementary instructions—either explicitly as in the Intel Small Matrix Library [22], or implicitly through the vectorization in BLAS. Batched factorizations can then be efficiently computed for multicore CPUs by having a single core factorize a single problem at a time. However, as we show, the energy consumption is higher than the GPU-based factorizations and that our GPU-based routine is about 2 times faster than the multicore implementation.

Despite the lack of support for batched operations, applications developers implemented particular routines for certain cases, trying various approaches. For example, targeting very small problems (of size up to 128), Villa et al. [37,38] 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 [43] 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 able to benefit from data reuse, the results showed that the performance in these approaches, up to about 20 Gflop/s in double precision, did not exceed the performance of memory bound kernels like `gemv` (which achieves up to 46 Gflop/s on a K40 GPU). Batched-specific algorithmic improvements were introduced for the Cholesky factorization [9] and the LU factorization [8,17] that exceed in performance the memory bound limitations mentioned above. Here we further develop and conceptualize an approach based on batched BLAS plus a number of batched-specific algorithmic innovations to significantly improve in performance the previously published results on batched linear algebra.

### 3 Methodology and Algorithmic Design

In a number of research papers [9,8,19], we have shown that high-performance batched algorithms can be designed so that the computation is performed by calls to batched BLAS kernels, to the extent possible by the current BLAS API. This is important since the use of BLAS has been crucial for the high-performance sustainability of major numerical libraries for decades, and therefore we can leverage the lessons learned from that success too. To enable the effective use of a batched BLAS based approach, there is a need to develop highly efficient and optimized batched BLAS routines that are needed by many high-level linear algebra algorithms such as Cholesky, LU, and QR, either in batched or classical fashion.



**Fig. 2.** Direct sparse or dense factorizations—a DAG approach that needs efficient computation of many small linear algebra tasks.

To put the proposed methodology in context, Figure 2 shows an illustration of our work on direct linear system solvers, be it sparse or dense, for many-core heterogeneous architectures. To provide parallelism in these solvers, the computation can be expressed as a Directed Acyclic Graph (DAG) of small tasks with labeled edges designating data dependencies, which naturally leads to the need to handle many small LA problems in parallel. Our work through vendor recognition centers, the efforts with collaborators from the HPC community, and the engagement with applications developers has resulted in the accumulation of expertise, technologies, and numerical software [6,27,42,31,1,42,40,41,32,3,30,12,7,15] that can be directly leveraged in the development of state-of-the-art, portable, cross-platform batched BLAS. The objective of our methodology is to minimize the development effort and have a parametrized kernels that can be used for tuning on different kind of architecture without the need to re-implement the kernel.

#### 3.1 Algorithmic Baseline

The QR factorization of an  $m$ -by- $n$  matrix  $A$  is of the form  $A = QR$ , where  $Q$  is an  $m$ -by- $m$  orthonormal matrix, and  $R$  is an  $m$ -by- $n$  upper-triangular matrix. The LAPACK routine **GEQRF** implements a right-looking QR factorization algorithm, whose first step consists of the following two phases:

- 1) **Panel factorization:** The first panel  $A_{:,1}$  is transformed into an upper-triangular matrix.
  1. **GEQR2** computes an  $m$ -by- $m$  Householder matrix  $H_1$  such that  $H_1^T A_{:,1} = \begin{pmatrix} R_{1,1} \\ 0 \end{pmatrix}$ , and  $R_{1,1}$  is an  $n_b$ -by- $n_b$  upper-triangular matrix.
  2. **LARFT** computes a block representation of the transformation  $H_1$ , i.e.,  $H_1 = I - V_1 T_1 V_1^H$ , where  $V_1$  is an  $m$ -by- $n_b$  matrix and  $T_1$  is an  $n_b$ -by- $n_b$  upper-triangular matrix.
- 2) **Trailing submatrix update:** **LARFB** applies the transformation computed by **GEQR2** and **LARFT** to the submatrix  $A_{:,2:n_t}$ :

$$\begin{pmatrix} R_{1,2:n_t} \\ \widehat{A} \end{pmatrix} := (I - V_1 T_1 V_1^H) \begin{pmatrix} A_{1,2:n_t} \\ A_{2:m_t,2:n_t} \end{pmatrix}.$$

Then, the QR factorization of  $A$  is computed by recursively applying the same transformation to the submatrix  $\widehat{A}$ . The transformations  $V_j$  are stored in the lower-triangular part of  $A$ , while  $R$  is stored in the upper-triangular part. Additional  $m$ -by- $n_b$  storage is required to store  $T_j$ .

### 3.2 Optimized and parametrized batched BLAS kernels

We developed the most needed and performance-critical Level 3 and Level 2 batched BLAS routines. Namely, we developed the batched **gemm** (general matrix-matrix multiplication), **trsm** (triangular matrix solver), and **gemv** (general matrix-vector product) routines, as well as a number of Level 1 BLAS such as the **dot** product, the **norm** functionality and the **scal** scaling routine. There are a number of feasible design choices for batched BLAS, each best suited for a particular case. Therefore, to capture as many of them as possible, we propose to build a design space for batched BLAS that will include parametrized algorithms that are tuned for modern and future hardware and take into account the matrix size. We also propose a parametrized-tuned approach to find the optimal implementation within the confines of the said design space.

We developed a parametrized basic kernel, that uses multiple levels of blocking, including shared memory and register blocking as well as double buffering techniques to hide the data communication with the computation. This kernel served us to optimize and tune the MAGMA **gemm** routine for large matrix sizes originally for Fermi GPUs [32], and lately for Kepler architectures. Recently, we extended it to a batched **gemm** [19,18], now released through MAGMA 1.6.1 [21]. The extension was done by autotuning the basic kernel and adding one more thread dimension to account for the batch count. Our strategy is to develop optimized components that can be used easily as plug-in device routine to provide many of the Level 3 and Level 2 BLAS routines. Following these techniques, we also developed our **trsm** kernel. It consists of a sequence of calls to invert a  $16 \times 16$  diagonal block followed by a call to the **gemm** components which are already optimized and tuned.

Moreover, we developed the batched equivalent of LAPACK's **geqr2** routine to perform the Householder panel factorizations. For a panel of  $nb$  columns, it consists of  $nb$  steps where each step calls a sequence of the **larfg** and the **larf** routines.

At every step (to compute one column), the `larfg` involves a norm computation followed by a `scal` that uses the results of the norm computation in addition to some underflow/overflow checking. These Level 1 BLAS kernels have been developed as device component routine to plug them easily when needed. The norm computation is a sum reduce and thus a synchronization 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 last 32 elements are reduced by a single thread. Our parametrized technique let us run our autotuner and tune these kernels. As a result, custom batched implementations of both `larfg` and the `larf` have been developed. When the panel size is small enough, we use the shared memory to load the whole panel and to perform its computation in fast memory. For larger panel sizes, we load only the vector that is annihilated at each step meaning that the `norm`, `scal`, and thus the `larfg` computation operate on data in shared memory, as well as the `larf` read data from shared memory but write data in main memory since it cannot fit into the shared memory. When the panel is very large, and here it becomes out of the context of batched small matrices, the BLAS kernel operates using many thread-blocks and an atomic synchronization.

### 3.3 Development of Batched LAPACK algorithms

The development of batched LAPACK algorithms and implementations is our main example of how to use the batched BLAS for higher-level algorithms. Section 1 highlighted the main challenges and describes how our work overcomes them through new algorithmic designs and use of high-performance batched BLAS. In particular, we overcome the memory bound limitations that previous developers had on small problems, and exceed the performance of even state-of-the-art vendor implementations by up to 3×. Similarly to the batched BLAS, we build a design space for batched LAPACK that will include parametrized algorithms that are architecture and matrix size aware. An autotuning approach is used to find the best implementation within the provisioned design space.

We developed the performance-critical LAPACK routines to solve small dense linear systems or least squares problems. Namely, we developed the LU, and Cholesky factorizations in [9,8,19], and we present in detail our progress and development for the QR decomposition in this paper.

We developed technologies for deriving high-performance entirely GPU implementations to solve sets of small linear algebra problems (as in LAPACK) in parallel. Note that entirely GPU implementations have been avoided up until recently in numerical libraries, especially for small and difficult to parallelize tasks like the ones targeted by the batched factorization. Indeed, hybridization approaches were in the forefront of developing large scale solvers as they were successfully resolving the problem by using CPUs for the memory bound tasks [40,2,44,11,16]. For large problems, the panel factorizations (the source of memory bound, not easy to parallelize tasks) are always performed on CPU. 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. Therefore, we developed an approach based

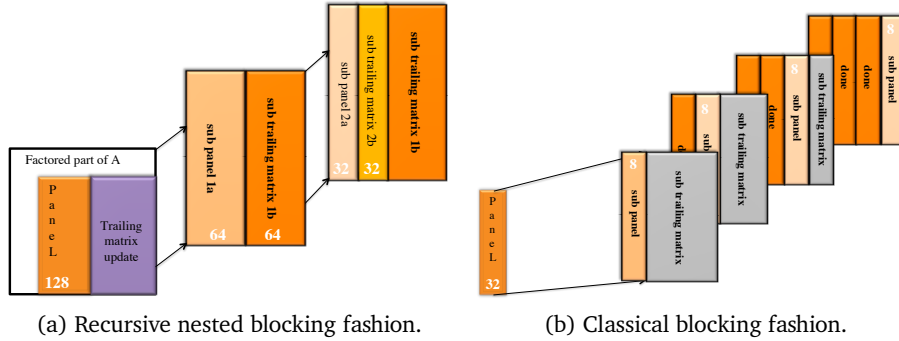
on a combination of 1) batched BLAS, 2) batched-specific, and 3) architecture-aware algorithmic improvements. Batched-specific algorithms that were different from LAPACK were needed since we could not outperform the NVIDIA-optimized LAPACK-based implementation by only using our own aggressive optimizations on top of the standard LAPACK algorithm. In particular, for our QR decomposition, besides high-performance batched BLAS, we also used batch-specific and architecture-aware algorithmic advances described below.

**Recursive Multilevel Nested Blocking.** The panel factorizations (`geqr2`) described above factorize the  $nb$  columns one after another, similarly to the LAPACK algorithm. At each of the  $nb$  steps, a rank-1 update is required to update the vectors to the right of the factorized column  $i$ . This operation is done by the `larf` kernel. Since we cannot load the entire panel into the shared memory of the GPU, the columns to the right are loaded back and forth from the main memory at every step except for the very small size cases (e.g., size less than  $32 \times 8$ ). Thus, one can expect that this is the most time consuming part of the panel factorization.

Our analysis using the NVIDIA Visual Profiler [33] shows that a large fraction of even a highly optimized batched factorization is spent in the panels, e.g., 40% of the time for the QR decomposition. The profiler reveals that the `larf` kernel requires more than 75% of the panel time by itself. The inefficient behavior of these routines is also due to the memory access. To resolve this challenge, we propose to improve the efficiency of the panel and to reduce the memory access by using a two-level nested blocking technique as depicted in Figure 3. First, we recursively split the panel to an acceptable block size  $nb$  as described in Figure 3a. In principle, the panel can be blocked recursively until a single element. Yet, in practice, 2-3 blocked levels (an  $nb = 32$  for double precision was the best) are sufficient to achieve high performance. Then, the routine that performs the panel factorization (`geqr2`) must be optimized, which complicates the implementation. This optimization can bring between 30% to 40% improvement depending on the panel and the matrix size. In order to reach our optimization goal, we also blocked the panel routine using the classical blocking fashion to small blocks of size  $ib$  ( $ib = 8$  was the optimized choice for double precision) as described in Figure 3b. More than 25% boost in performance is obtained by this optimization.

**Block Recursive `dlarft` Algorithm.** The `larft` 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 1. Such implementation takes up to 50% of the total QR factorization time. This is due to the fact that the kernels needed – `gemv` and `trmv` – 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 `gemv` and





**Fig. 3.** The recursive two-level nested blocking fashion used in our implementation to achieve high-performance batched kernels.

one for `trmv`. The `gemv` loop that computes each column of  $\widehat{T}$  (see the notation in Algorithm 1) can be replaced by one `gemm` 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 `gemm` in the `larfb`, and thus can be exploited here as well. For the `trmv` 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 2. Since we developed recursive blocking algorithm, we must compute the  $T$  matrix for every level of the recursion. Nevertheless, the analysis of Algorithm 2 let us conclude that the portion of the  $T$ 's 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 4), and thus we can avoid their (re-)computation. For that we modified Algorithm 2 in order to compute either the whole  $T$  or the upper rectangular portion that is missed (red/yellow portions in Figure 4). Redesigning the algorithm to block the computation using Level 3 BLAS, accelerated the overall algorithm on average by about 20 – 30% (depending on various parameters).

```

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

```

**Algorithm 1:** Classical implementation of the `darft` routine.

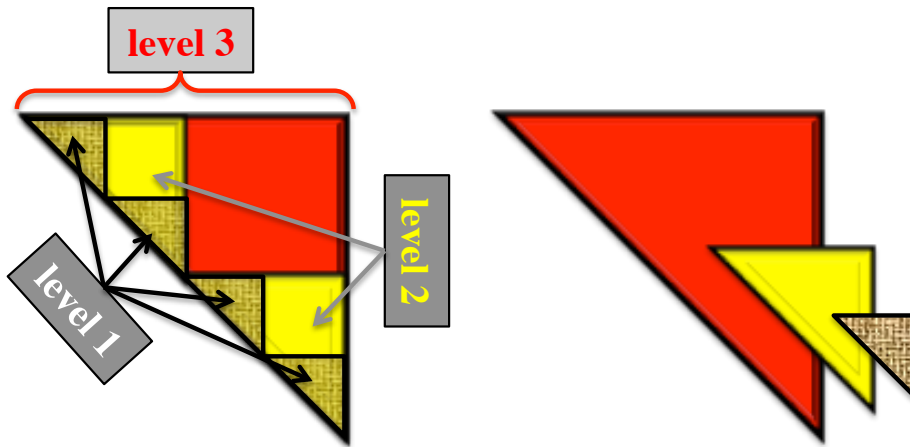
**Trading extra computation for higher performance.** The goal here is to replace the use of low performance kernels with higher performance ones—often for the

```

dgemm to compute  $\widehat{T}_{1:nb,1:nb} = A_{1:m,1:nb}^H \times A_{1:m,1:nb}$ 
load  $\widehat{T}_{1:nb,1:nb}$  to the shared memory. for  $j \in \{1, 2, \dots, nb\}$  do
┌   dtrmv to compute  $T_{1:j-1,j} = T_{1:j-1,1:j-1} \times \widehat{T}_{1:j-1,j}$ 
│    $T(j,j) = \text{tau}(j)$ 
└
write back  $T$  to the main memory.

```

**Algorithm 2:** Block recursive dlarft routine.



**Fig. 4.** The shape of the matrix  $T$  for different level of the recursion during the QR decomposition.

cost of more flops, e.g., `trmm` used by the `larfb` can be replaced by `gemm`. The QR trailing matrix update uses the `larfb` routine to perform  $A = (I - VT^H V^H)A$ . The upper triangle of  $V$  is zero with ones on the diagonal and also the matrix  $T$  is upper triangular. The classical `larfb` uses `trmm` to perform the multiplication with  $T$  and with the upper portion of  $V$ . If one can guarantee that the lower portion of  $T$  is filled with zeroes and the upper portion of  $V$  is filled with zeroes and ones on the diagonal, then the `trmm` can be replaced by `gemm`. Thus we implemented a batched `larfb` that uses three `gemm` kernels by initializing the lower portion of  $T$  with zeroes, and filling up the upper portion of  $V$  with zeroes and ones on the diagonal. Note that this brings  $3nb^3$  extra operations. The benefits again depend on various parameters, but on current architectures we observe an average of 10% improvement, and see a trend where its effect on the acceleration grows from older to newer systems.

## 4 Performance Results

### 4.1 Hardware Description and Setup

We conducted our experiments on an Intel two-socket multicore system with two 8-core Intel Xeon E5-2670 (Sandy Bridge) processors, each running at 2.6 GHz. Each socket has 20 MiB of shared Level 3 cache, and each core has a private 256 KiB Level 2 and 64 KiB Level 1 cache. The system is equipped with the total of 52 GiB of main 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 NVIDIA K40c cards with 11.6 GiB of GDDR memory per card running at 825 MHz. The theoretical peak in double precision is 1,689.6 Gflop/s per GPU. 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 MKL (Math Kernel Library) [23] with the `icc` compiler (version 2013.sp1.2.144) and on the GPU accelerator we used CUDA version 6.5.14.

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) 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.

### 4.2 Performance results

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 `dgeqrfBatched` routine from the CUBLAS [35] library. Our experiments were performed on batches of 1,000 matrices of different sizes going from  $32 \times 32$  to  $1024 \times 1024$ .

We also compare our batched QR to two CPU implementations. First is the simple CPU implementation which mean 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 90 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 ( $< 1024$ ) and at least 16 of them fit in the Level 3 cache, 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 believe 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 160 Gflop/s in factorizing 1,000 matrices of size  $1024 \times 1024$ .

The progress of our batched QR implementation over the different optimizations shows attractive behavior. For a 1000 matrix of size  $512 \times 512$  each, the classical block implementation does not attain more than 55 Gflop/s. The inner panel blocking makes the performance reach around 70 Gflop/s; and the recursive blocking alone improves performance up to 108 Gflop/s; combined, the two-level blocking brings performance up to around 136 Gflop/s. The optimized computation of  $T$  draws it up to 195 Gflop/s. The other optimizations (replacing `dtrmm` by `dgemm` in both `darft` and `darfb`), combined with the streamed/batched `dgemm` calls, bring the performance of the GPU implementation to around 221 Gflop/s. Despite the CPU's hierarchical memory advantage, our experiments show that our GPU batched QR factorization is able to achieve a speedup of  $2\times$  vs. the best CPU implementation using 16 Sandy Bridge cores, and  $4\times$  vs. the simple one. Moreover, our algorithm reaches around 334 Gflop/s for matrices of size  $1024 \times 1024$ , which is between  $5\times$  to  $20\times$  faster than the CUBLAS implementation for matrices in the range of 512 to 1024. We should mention that the CUBLAS implementation is well suited for very small matrices such as matrices of size less than  $64 \times 64$ . The performance of CUBLAS for these sizes outperforms our proposed algorithm as well as both of the CPU implementations.

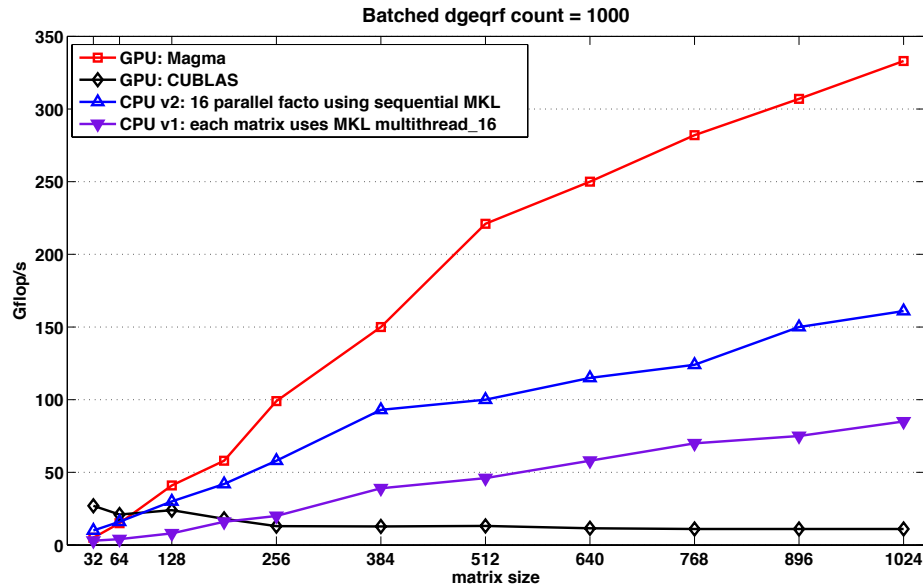


Fig. 5. Performance in Gflops/s of the GPU vs. the CPU versions of our batched QR decomposition for different matrix sizes where  $m = n$ .

### 4.3 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 [24,39]. 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 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  $125 \div 140$  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. We should mention that the CPU measurement do not include the power cost of the memory access while the GPU measurement includes it. In order to include the power for the CPU, we had to change in the BIOS and we were not allowed to do it on our testing machine. However, our previous results showed that the power of the CPU memory access can be estimated to be 40 W on average. On some systems, energy consumption numbers do not include the power consumed by the main memory as the memory modules do not report their voltage levels to the CPU’s memory controller on those systems, which renders RAPL ineffective for the purpose of estimating temporal power draw. However, based on estimates from similarly configured systems, we estimate that the power consumption for the main memory under load is between 30 W and 40 W depending on the memory size and configuration.

For the GPU measurements we use NVIDIA’s NVML (NVIDIA Management Library) library [34]. 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  $\pm 5\%$  of current power draw. The idle state of the K40c GPU consumes about 20 W. Batched factorizations raise the consumption to about  $150 \div 180$  W, while large dense matrix operations raise the power draw to about 200 W. For reference, it is worth noting that the active idle state draws 62 W.

We depict in Figure 6 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 1,000 matrices of size  $1024 \times 1024$  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 1,000 matrices where each matrix is factorized using the multithreaded `dgeqrf` routine from 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 from 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  $2\times$  faster than the optimized CPU implementation, and moreover, because it consumes  $3\times$  less energy.

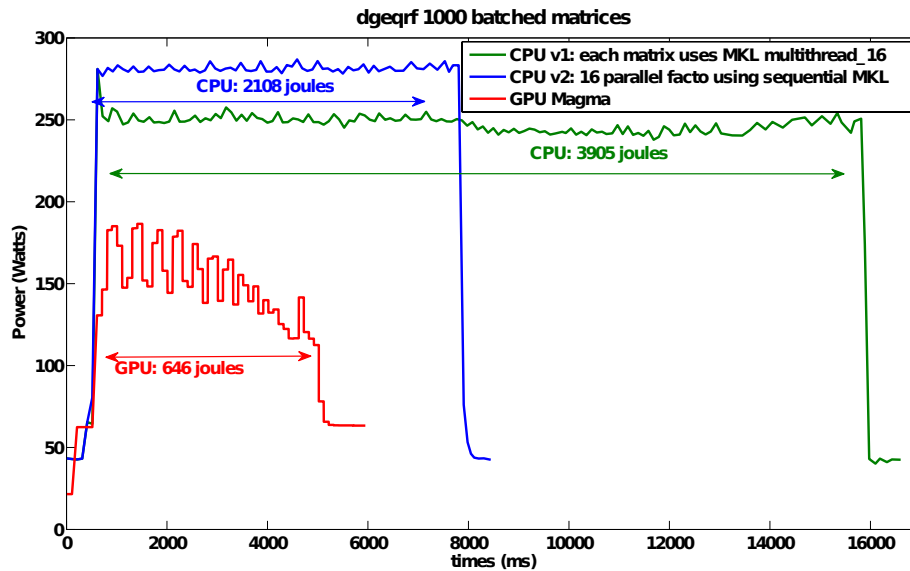


Fig. 6. Comparison of the power consumption for the QR decomposition of 1,000 matrices of size  $1024 \times 1024$ .

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  $2\times$  speedup over their best CPU counterpart and are  $3\times$  less expensive in term of energy.

## 5 Conclusions an Future Work

Designing algorithms to work on small problems is a concept that can deliver higher performance through 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 as well, as we showed. Therefore, by solving the technical issues and providing the needed batched LA tools, the future development on the framework presented will also address the following long term goals: (1) Define a new standard for the use of small matrix computations in applications; (2) Provide a methodology to solve many small size LA problems and an initial implementation that is portable at all levels of the platform pyramid, from embedded devices to supercomputers; (3) Establish grounds for the next generation of innovations in HPC applications and sustainability of high-performance numerical libraries.

Future work extensions include building batched sparse, and application-specific batched linear algebra capabilities. Of specific interest will be the effect of the batched framework on high-performance numerical libraries and run-time systems. Current approaches, e.g., in dense tiled algorithms, are based on splitting algorithms into small tasks that get inserted into, and scheduled for execution by, a run-time system. This often amounts to splitting large gemms into many small gemms, which is known to encounter overheads for scheduling and saving parameters (although most are the same). The batched approach, besides providing high performance for small tasks, will be a natural fit to extend these and similar libraries, as well as provide a new hierarchical scheduling model for queuing jobs, organizing run-time systems, and interacting with accelerators/co-processors.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. ACI-1339822, the Department of Energy, and Intel. The results were obtained in part with the financial support of the Russian Scientific Fund, Agreement N14-11-00190.

## References

1. 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.
2. Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.
3. Emmanuel Agullo, Jack Dongarra, Rajib Nath, and Stanimire Tomov. Fully empirical autotuned qr factorization for multicore architectures. *CoRR*, abs/1102.5328, 2011.
4. ACML - AMD Core Math Library, 2014. Available at <http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml>.
5. M.J. 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. Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julien Langou, Piotr Luszczek, and Stanimire Tomov. The impact of multicore on math software. In Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Wasniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing, 8th International Workshop, PARA*, volume 4699 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2006.
7. C. Cao, J. Dongarra, P. Du, M. Gates, P. Luszczek, and S. Tomov. clMAGMA: High performance dense linear algebra with OpenCL. In *The ACM International Conference Series*, Atlanta, GA, may 13-14 2013. (submitted).
8. T. Dong, A. Haidar, P. Luszczek, A. Harris, S. Tomov, and J. Dongarra. LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU. In *Proceedings of 16th IEEE International Conference on High Performance and Communications (HPCC 2014)*, August 2014.

9. T. Dong, A. Haidar, S. Tomov, and J. Dongarra. A fast batched Cholesky factorization on a GPU. In *Proceedings of 2014 International Conference on Parallel Processing (ICPP-2014)*, September 2014.
10. Tingxing Dong, Veselin Dobrev, Tzanio Kolev, Robert Rieben, Stanimire Tomov, and Jack Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014.
11. 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.
12. Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.*, 38(8):391–407, August 2012.
13. Oak Ridge Leadership Computing Facility. Annual report 2013-2014, 2014. [https://www.olcf.ornl.gov/wp-content/uploads/2015/01/AR\\_2014\\_Small.pdf](https://www.olcf.ornl.gov/wp-content/uploads/2015/01/AR_2014_Small.pdf).
14. J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.
15. 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).
16. Azzam Haidar, Chongxiao Cao, Asim Yarkhan, Piotr Luszczek, Stanimire Tomov, Khairul Kabir, and Jack Dongarra. Unified development for mixed multi-gpu and multi-coprocessor environments using a lightweight runtime environment. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14*, pages 491–500, Washington, DC, USA, 2014. IEEE Computer Society.
17. Azzam Haidar, Tingxing Dong, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Batched matrix computations on hardware accelerators based on GPUs. *International Journal of High Performance Computing Applications*, doi:10.1177/1094342014567546, 02/2015.
18. Azzam Haidar, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Optimization for performance and energy for batched matrix computations on gpus. In *8th Workshop on General Purpose Processing Using GPUs (GPGPU 8) co-located with PPOPP 2015*, PPOPP 2015, San Francisco, CA, 02/2015 2015. ACM, ACM.
19. Azzam Haidar, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Towards batched linear solvers on accelerated hardware platforms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2015*, San Francisco, CA, 02/2015 2015. ACM, ACM.
20. Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, February 2004.
21. Matrix algebra on GPU and multicore architectures (MAGMA), MAGMA Release 1.6.1, 2015. Available at <http://icl.cs.utk.edu/magma/>.
22. Intel Pentium III Processor - Small Matrix Library, 1999. Available at <http://www.intel.com/design/pentiumiii/sml/>.
23. Intel Math Kernel Library, 2014. Available at <http://software.intel.com/intel-mkl/>.
24. Intel® 64 and IA-32 architectures software developer's manual, July 20 2014. Available at <http://download.intel.com/products/processor/manual/>.
25. David Keyes and Valerie Taylor. NSF-ACCI task force on software for science and engineering, December 2010.



26. J.C. Liao Khodayari A., A.R. Zomorodi and C.D. Maranas. A kinetic model of escherichia coli core metabolism satisfying multiple sets of mutant flux data. *Metabolic engineering*, 25C:50–62, 2014.
27. Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *Proceedings of the 2009 International Conference on Computational Science, ICCS'09*, Baton Rouge, LA, May 25-27 2009. Springer.
28. O.E.B. Messer, J.A. Harris, S. Parete-Koon, and M.A. 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.
29. J.M. Molero, E.M. Garzón, I. García, E.S. Quintana-Ortí, and A. Plaza. Poster: A batched Cholesky solver for local RX anomaly detection on GPUs, 2013. PUMPS.
30. R. Nath, S. Tomov, T. Dong, and J. Dongarra. Optimizing symmetric dense matrix-vector multiplication on GPUs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, nov 2011.
31. R. Nath, S. Tomov, and J. Dongarra. Accelerating GPU kernels for dense linear algebra. In *Proceedings of the 2009 International Meeting on High Performance Computing for Computational Science, VECPAR'10*, Berkeley, CA, June 22-25 2010. Springer.
32. Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved magma gemm for fermi graphics processing units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, November 2010.
33. Nvidia visual profiler.
34. Available at <https://developer.nvidia.com/nvidia-management-library-nvml>, 2014.
35. CUBLAS, 2014. Available at <http://docs.nvidia.com/cuda/cublas/>.
36. CUBLAS 6.5, January 2015. Available at <http://docs.nvidia.com/cuda/cublas/>.
37. Villa Oreste, Massimiliano Fatica, Nitin A. Gawande, and Antonino 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.
38. Villa Oreste, Nitin A. Gawande, and Antonino Tumeo. Accelerating subsurface transport simulation on heterogeneous clusters. In *IEEE International Conference on Cluster Computing (CLUSTER 2013)*, Indianapolis, Indiana, September, 23-27 2013.
39. Efraim Rotem, Alon Naveh, Doron Rajwan, Avinash Ananthakrishnan, and Eliezer 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](https://doi.org/10.1109/MM.2012.12).
40. S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Comput. Syst. Appl.*, 36(5-6):232–240, 2010. DOI: [10.1016/j.parco.2009.12.005](https://doi.org/10.1016/j.parco.2009.12.005).
41. S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010. IEEE Computer Society. DOI: [10.1109/IPDPSW.2010.5470941](https://doi.org/10.1109/IPDPSW.2010.5470941).
42. Stanimire Tomov and Jack Dongarra. Dense linear algebra for hybrid gpu-based systems. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, *Scientific Computing with Multicore and Accelerators*. Chapman and Hall/CRC, 2010.
43. Ian Wainwright. Optimized LU-decomposition with full pivot for small batched matrices, April, 2013. GTC'13 – ID S3069.
44. Ichitaro Yamazaki, Stanimire Tomov, and Jack Dongarra. One-sided dense matrix factorizations on a multicore with multiple GPU accelerators. *Procedia Computer Science*, 9(0):37 – 46, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

45. Sencer N. Yeralan, Tim A. Davis, and Sanjay Ranka. Sparse multifrontal QR on the GPU. Technical report, University of Florida Technical Report, 2013.