Towards Batched Linear Solvers on Accelerated Hardware Platforms

Azzam Haidar[1] Tingxing Dong[1] Piotr Luszczek[1] Stanimire Tomov[1] Jack Dongarra[1,2,3]

University of Tennessee, Knoxville, TN 37916, USA[1]; Oak Ridge National Laboratory, USA[2]; University of Manchester, UK[3]

{haidar, tdong, luszczek, tomov, dongarra}@eecs.utk.edu

Abstract

As hardware evolves, an increasingly effective approach to develop energy efficient, high-performance solvers, is to design them to work on many small and independent problems. Indeed, many applications already need this functionality, especially for GPUs, which are known to be currently about four to five times more energy efficient than multicore CPUs for every floating-point operation. In this paper, we describe the development of the main one-sided factorizations: LU, QR, and Cholesky; that are needed for a set of small dense matrices to work in parallel. We refer to such algorithms as batched factorizations. Our approach is based on representing the algorithms as a sequence of batched BLAS routines for GPU-contained execution. Note that this is similar in functionality to the LAPACK and the hybrid MAGMA algorithms for large-matrix factorizations. But it is different from a straightforward approach, whereby each of GPU's symmetric multiprocessors factorizes a single problem at a time. We illustrate how our performance analysis together with the profiling and tracing tools guided the development of batched factorizations 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 on a two-sockets, Intel Sandy Bridge server. Compared to a batched LU factorization featured in the NVIDIA's CUBLAS library for GPUs, we achieves up to 2.5-fold speedup on the K40 GPU.

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

General Terms Algorithms, Experimentation, Measurement, Performance

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

1. Parallel Swapping on GPUs

Profiling the batched LU reveals that more than 60% of the time is spent in the swapping routine. We can observe on the trace that the classic 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 for this kernel to be the most time consuming is because the nb row interchanges are performed in a sequential manner, and the data of a row is not coalesced, thus the thread warps do not read/write in parallel. It is clear that the main bottleneck here is the memory access. Indeed, slow memory access compared to high compute capability 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. GPUs on the other hand, in addition to cache memory, use thread level parallelism (TLP) whereby threads are grouped into warps and multiple warps assigned for execution on the same SMX unit. When a warp issues an access to the device memory, it stalls until the memory returns a value, while the GPU'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 followed by the dtrsm. 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. Our experiments show that this reduces the time spent in the kernel from 60% to around 10% of the total elapsed time. As a result, the performance gain obtained is about $1.8\times$ as shown by the purple curve of Figure 1. We report each of the proposed optimization for the LU factorization in Figure 1 but we would like to mention that the percentage of improvement obtained for the Cholesky and QR

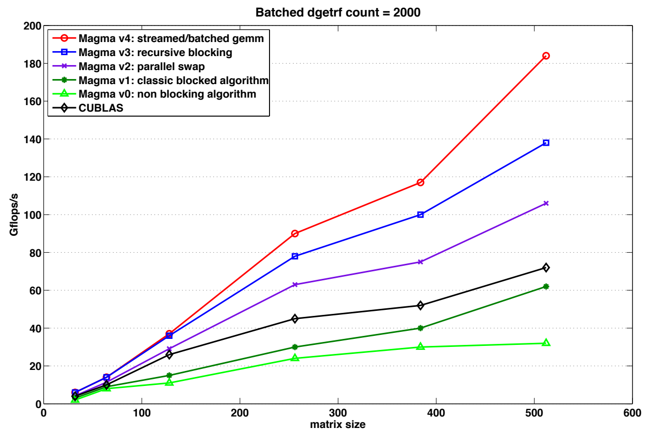

Figure 1. Performance in Gflops/s of the different versions of our batched LU factorizations compared to the CUBLAS implementation for square matrix sizes.

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA.
Copyright © 2015 ACM 978-1-4503-3205-7/15/02…$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Algorithm 1: Classical implementation of the dlarft routine.

```
for $j \in \{1, 2, \ldots, nb\}$ 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 $T_{1:j-1,j} = T_{1:j-1,1:j-1} \times \widehat{T}_{1:j-1,j}$
    $T(j,j) = tau(j)$
```

factorization is similar and to simplify we report 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.

2. Recursive Nested Blocking

The panel factorizations operate on 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 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. Profiling 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 needs 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. To overcome this, we improve the efficiency of the panel and reduce the memory access by using a recursive blocking technique depicted in Figure 2. 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 1 for the LU factorization. The same trend has been observed for both the Cholesky and the QR factorization.

3. Trading Extra Computation for Performance

For batched problems there is a need to minimize the use 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

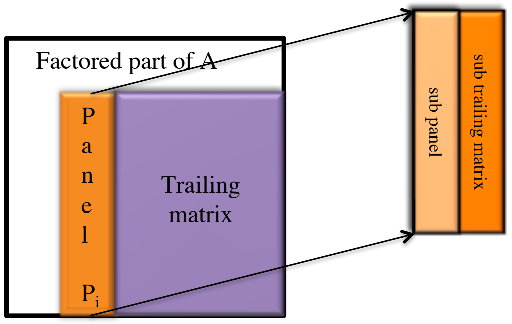

Figure 2. Recursive nested blocking

Algorithm 2: 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 $j \in \{1, 2, \ldots, nb\}$
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.
```

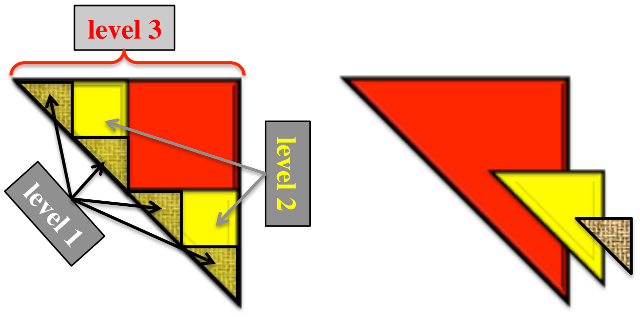

Figure 3. The shape of the matrix T for different level of the recursion during the QR decomposition.

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. The use of dgemm results in higher performance than dtrmm that only access either lower or upper portion of the diagonal blocks. Tests show that our implementation of dsyrk is twice as fast as the dgemm kernel for the same matrix size. Our dsyrk is optimized in order to reach the performance of dgemm (twice as slow due to twice as many 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 $A_{22} \leftarrow (I - VT^HV^H)A_{22} \leftarrow (I - A_{21}T^HA^H_{21})A_{22}$. The upper triangle of V is an identity. In the classic dlarfb A_{21} is available and it 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. A batched dlarfb 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 the overall time spent in the new dlarfb even with the extra computation is around 10% less than the one using the dtrmm.

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

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. LAPACK computes T column by column in a loop over the nb columns as described in Algorithm 1. Such an implementation takes up to 50% of the total QR factorization time.