GPU Acceleration of Small Dense Matrix Computation of the One-Sided Factorizations

Tingxing Dong, Mark Gates, Azzam Haidar, Piotr Luszczek, Stanimire Tomov

I. Introduction

Various scientific applications use Gaussian elimination or Cholesky or QR factorization to solve dense linear systems. For an important class of problems, a relatively large number of small size systems is generated and must be solved. Typically, the order of these linear systems is up to a few hundred, and their number is from a few thousand to millions. For example, subsurface transportation simulations have a number of reaction systems to solve. Each system involves computing a Jacobian matrix and iteratively applying Gaussian elimination until an outer solver converges. The system size is typically around 100.

As another example, consider an astrophysics ODE solver with Newton-Raphson iteration [1]. Multiple zones are simulated in one MPI task and each zone corresponds to a small linear system with each one resulting in multiple sequential solves [1]. A sparse direct solver called MA48 solves a sparse nonsymmetric system of m linear equations in n unknowns using Gaussian elimination. The typical matrix size is 150 by 150. If the matrix is symmetric and definite, the problem is reduced to batched Cholesky factorization [2]. Other examples include hydrodynamic simulations, e.g., where the need is to compute thousands of matrix-matrix multiplies (dgemm) for dimensions well below 100 by 100 [3].

The one-sided factorizations such as the Cholesky, LU, and QR factorizations are based on block outer-product updates of the trailing matrix. Algorithmically, this corresponds to a sequence of two distinct phases: the *panel factorization* and the *trailing matrix update*.

The panel factorization is latency and memory-bound due to its predominant reliance on the Level 2 BLAS operations. The implementation from the MAGMA library performs the panel factorization on the CPU and only uses the GPU to update the trailing matrix. A data transfer of the factorized panel from the CPU to the GPU is required at each step of the outermost loop.

However, in the batched LU implementation we cannot 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, especially due to the fact that the data for transfer are not continuous in the memory but instead are stored with a stride called a leading dimension. Another challenge in achieving good performance is the pivoting, which is a source of thread divergence and non-coalesced memory accesses. This is the result of consecutive threads accessing the matrix elements with a stride of one column instead of a single-element stride when the matrix is stored in column-major format.

II. Batched Implementation

We target matrices of size less than or equal to 512, since most application candidates for batched execution are of this size [1], [4]. In a batched problem, each matrix is a separate problem that is solved independently. All of the routines discussed are batched and denoted by the corresponding LAPACK routine name. We have implemented the routines in the four standard floating-point precisions. For convenience, we use the double precision routine name throughout.

III. Various factor impacts on the performance

A. Parallel Pivot Swapping

We analyzed and evaluated the implementation as described above to find that more than 60% of the factorization time is spent in the pivot swapping routine. 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 coalesced, thus the threads do not read/write it in parallel. 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, uses thread level parallelism (TLP) where threads are grouped into blocks (e.g., of 32 threads) and multiple blocks assigned for execution on the same SMX unit. In order to overcome the bottleneck of swapping, we proposed to modify the kernel in order to apply all nb row swaps in parallel. This modification will also allow the writes of the first nb rows of the matrix to be coalesced. So we changed the algorithm to generate two pivot vectors, where the first vector gives the final destination row indices for the first nb rows of the panel, and the second gives the row indices of the nb rows that must become the first nb rows of the panel. Our experiments show that this reduces the time spent in the kernel from 60% to around 10%. As a result, the gain obtained in terms of performance is around 50%.

B. Nested and Multi-Level Blocking of the Panel Factorization

The panel factorization goes over the nb columns and factorizes them 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 hand side of the factorized column i (this operation is done by the dger kernel). Since we cannot load the entire panel into the shared memory of the GPU, the right hand side vectors are loaded back and forth from the main memory at every step. Thus, one can expect that the rank-1 operation is the most time consuming of the panel factorization. A detailed analysis using the profiler reveals that the dger kernel consists of more than 80% of the panel factorization time, and around 40% of the total LU factorization time. Similarly to the swapping kernel described above, the main bottleneck here is the memory access. To this end, we propose to improve the efficiency of this kernel and to reduce the memory access by using a recursive blocking technique. In principle, the panel can be blocked recursively until a single element. In practice, two to three blocked levels are sufficient to achieve high performance. The above routines must be optimized for each blocked level, which complicates the implementation. The boost in performance obtained by this optimization is around 25%.

C. Streamed dgemm

Our main goal is to achieve higher performance and to accomplish this we performed deep analysis of every kernel of the algorithm. We found that 70% of the time is spent in the batched dgemm kernel. An evaluation of the performance of the dgemm kernel using either batched or streamed dgemm concludes that the streamed dgemm was performing better than the batched one for some cases, e.g., for $k = 32$ when the matrix size is of order of $m > 200$ and $n > 200$. We note that the performance of the batched dgemm is stable and does not depend 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. Thus, we propose to use the streamed dgemm whenever it is faster, and to roll back to the batched one otherwise. The use of the streamed dgemm (when the size allows it) can speed up the factorization by about 20%.

D. Multi-level Blocking of the Trailing Matrix Update

The performance of the streamed dgemm kernel is highly dependent on the size of the matrices. In particular, this affects the trailing matrix updates in the LU factorization which consist of rank-k operations. The performance of the streamed dgemm kernel is around twice higher for $k = 128$ than for $k = 32$. Since our panel size is limited to 32, the performance of the trailing matrix update is limited by the performance of the dgemm for $k = 32$. However, in order to achieve higher performance, we use a multi-level blocking of the trailing matrix update. This means that at step i we update only the next panel and delay the subsequent portion of the update till step $i+l$, where we reach a value of k that is acceptable to perform the whole update of the delayed portion and then start over again.

We can observe that for this range of small matrices, increasing the value of acceptable "k" for example to 128 gives us the advantage of performing dgemm at higher speed but it reduce the number of such dgemm operations. The performance observed is similar for both $k = 64$ and $k = 128$ for matrices of size 512 while $k = 64$ is always outperforming $k = 128$ for smaller sizes. As a result, a trade-off value of k needs to be chosen depending on the matrix size. The improvement obtained by this technique is around 15%.

IV. Performance Results

We conducted our experiments on a NVIDIA K40c card with 11.6 GB of GDDR memory per card running at 825 MHz. The cards were connected to the host via two PCIe I/O hubs with 6 GB/s bandwidth. On the CPU side, we used the MKL (Math Kernel Library) [5] and on the GPU we used CUDA version 5.5.

CUBLAS version 5.5 features a dgetrfBatched routine. By comparison, our batched LU is up to 2.5× faster than the CUBLAS routine. The slowest code in the figure has performance below 60 Gflop/s and is marked as "classic" — it corresponds to the performance of the MAGMA library, which was optimized for large matrices. The *classic* implementation is improved upon by CUBLAS' dgetrfBatched version and the performance exceeds 70 Gflop/s. To go beyond 100 Gflop/s, we used the code that optimizes pivoting with *parallel swap*. The next step in performance improvement is the use of variable blocking (also called *recursive blocking* getf2), which enables performance levels that go slightly above 130 Gflop/s. The final two improvements are *streamed/batched gemm*, which moves the performance level beyond 160 Gflop/s, and finally, a *two-level blocking update* completes the set of optimizations and takes the performance beyond the 180 Gflop/s mark. Each of these optimizations is described in detail in Section II.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] "Intel Math Kernel Library," http://software.intel.com/intel-mkl/.