LU Factorization with Partial Pivoting for a Multi-CPU, Multi-GPU Shared Memory System

– LAPACK Working Note 266

Jakub Kurzak
Piotr Luszczek
Mathieu Faverge
Electrical Engineering and Computer Science, University of Tennessee

Jack Dongarra
Electrical Engineering and Computer Science, University of Tennessee
Computer Science and Mathematics Division, Oak Ridge National Laboratory
School of Mathematics & School of Computer Science, University of Manchester

Abstract

LU factorization with partial pivoting is a canonical numerical procedure and the main component of the High Performance LINPACK benchmark. This article presents an implementation of the algorithm for a hybrid, shared memory, system with standard CPU cores and GPU accelerators. Performance in excess of one TeraFLOPS is achieved using four AMD Magny Cours CPUs and four NVIDIA Fermi GPUs.

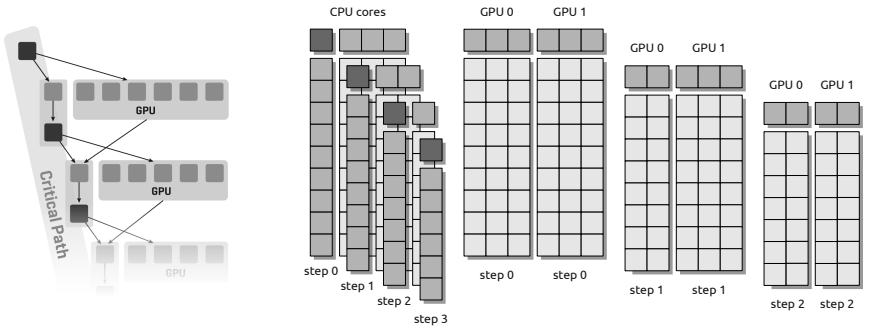
1 Introduction

This paper presents an implementation of the canonical formulation of the LU factorization, which relies on partial (row) pivoting for numerical stability. It is equivalent to the `DGETRF` function in the LAPACK numerical library. Since the algorithm is coded in double precision, it can serve as the basis for an implementation of the *High Performance LINPACK* benchmark (HPL) [1]. The target platform is a hybrid, multi-CPU, multi-GPU shared memory system.

2 Background

The LAPACK block LU factorization is the main point of reference here, and LAPACK naming convention is followed. The LU factorization of a matrix M has the form $M = PLU$, where L is a unit lower triangular matrix, U is an upper

triangular matrix and P is a permutation matrix. The LAPACK algorithm proceeds in the following steps: Initially, a set of *nb* columns (*the panel*) is factored and a pivoting pattern is produced (`DGETF2`). Then the elementary transformations, resulting from the panel factorization, are applied to the remaining part of the matrix (*the trailing submatrix*). This involves swapping of up to *nb* rows of the trailing submatrix (`DLASWP`), according to the pivoting pattern, application of a triangular solve with multiple right-hand-sides to the top *nb* rows of the trailing submatrix (`DTRSM`), and finally, application of matrix multiplication of the form $C = C - A \times B$ (`DGEMM`), where A is the panel without the top *nb* rows, B is the top *nb* rows of the trailing submatrix, and C is the trailing submatrix without the top *nb* rows. Then the procedure is applied repeatedly, descending down the diagonal of the matrix.



(a) Task Graph Scheduling

(b) Work Partitioning

Figure 1: Scheduling the task graph of the LU factorization, with fine-grained tasks on the critical path being dispatched to individual CPU cores and coarse-grained tasks outside of the critical path being dispatched to GPU devices.

3 The Solution

The main hybridization idea is captured on Figure 1 and relies on representing the work as a *Directed Acyclic Graph* (DAG) and dynamic task scheduling, with CPU cores handling the complex fine-grained tasks on the *critical path* (the longest path through the DAG), and GPUs handling the coarse-grained data-parallel tasks outside of the critical path. Some number of columns (*lookahead*) are assigned to the CPUs, and the rest of the matrix is assigned to the GPUs in a 1D block-cyclic fashion. In each step of the factorization, the CPUs factor a panel and update their portion of the trailing submatrix, while the GPUs update their portions of the trailing submatrix. After each step, one column of tiles shifts from the GPUs to the CPUs.

The implementation relies on a number of state-of-the-art solutions such as: tile data layout, block-cyclic data distribution, parallel recursive panel factorization, GPU kernel autotuning, the technique of *lookahead*, the use of superscalar scheduling and communication-computation overlapping.

3.1 Tile Data Layout

The matrix is laid out in square tiles on the CPU side (*host memory*), where each tile occupies a continuous region of memory. Tiles are stored in column-major and elements within tiles are stored in column-major. This layout, referred to as *Column-Column Rectangular Block* (CCRB) [2] is the native layout of the PLASMA library[3]. Tiles are transposed on the GPU side (*device memory*), i.e. the layout is translated to *Column-Row Rectangular Block* (CRRB), which is critical to the performance of the row swap (`DLASWP`) operation. This tile-wise transposition is trivial to code and fast to execute.

3.2 CPU Kernels

CPUs are responsible for the panel factorization and a portion of the update of the trailing submatrix. The update is relatively straightforward and requires three operations: row swap (`DLASWP`), triangular solve (`DTRSM`) and matrix multiplication (`DGEMM`). In the case of `DLASWP`, one core is responsible for swaps in one column of tiles. The LAPACK `DLASWP` function cannot be used, because of the use of tile layout, so `DLASWP` is hand-coded. In the case of `DTRSM` and `DGEMM` one core is responsible for one tile. Calls to Intel *Math Kernel Library* (MKL) are used, with layout set to column-major and the *leading dimension* set to tile size (*nb*).

The LAPACK panel factorization (`DGETF2`) is sequential and memory bound, and can deliver performance of roughly 0.5 Gflop/s, which is completely inadequate for a hybrid LU implementation. Running at such speed, panel factorizations would completely dominate the entire execution time. A fast alternative is absolutely critical. Here, the recursive-parallel panel factorization from the PLASMA library is used, providing an order of magnitude higher performance.

The application of recursion allows for a decrease in memory intensity by introducing some degree of level 3 BLAS operations [4]. Tiles of the panel are assigned to cores in a *round-robin* fashion and each core preserves the same set of tiles throughout all the steps of the panel factorization. At some point in the LU factorization, panels become short enough to fit in the aggregate cache of the designated cores, i.e., panel operations become cache-resident, which at some level resembles the technique of *Parallel Cache Assignment* (PCA) [5] currently employed by ATLAS. The cores are forced to work in lock-step, but can benefit from a high level of cache reuse. The ultra-fine granularity of operations requires very light-weight

synchronization. Synchronization is implemented using *busy-waiting* on volatile variables and works at the speed of hardware cache-coherency.

3.3 GPU Kernels

The update of the trailing submatrix on the GPUs requires kernels for three operations: row swap (`DLASWP`), triangular solve (`DTRSM`) and matrix multiplication (`DGEMM`). Also, a tile-wise transposition is required to convert between the CCRB layout in the host memory and the CRRB layout in the device memory. This transposition follows the transfer of each panel from the host memory to the device memory and precedes the transfer of each column returning from the device memory to the host memory.

`DLASWP` is implemented by creating *nb* (tile size) threads per multiprocessor and assigning one column to each thread. `DTRSM` (an in-place operation) is replaced by an inversion of the diagonal block (application of the L factor to identity) on a CPU, followed by a `DGEMM` on the GPUs (out-of-place). The transposition is implemented by spanning the column being transposed with a block-grid / thread-grid, such that each individual thread transposes one element (no loops in the kernel). These straightforward implementations are sufficient to make the impact of the operations negligible in comparison to the `DGEMM`.

The `DGEMM` kernels are produced using the *Automatic Stencil TuneR for Accelerators* (ASTRA) system [6], which follows the principles of *Automated Empirical Optimization of Software* (AEOS), popularized by the *Automatically Tuned Linear Algebra Software* (ATLAS) [7]. The same process is currently used to produce `DGEMM` kernels for the MAGMA project [8].

The kernel is expressed through a parametrized *stencil*, creating a large search space of possible implementations. The search space is aggressively pruned, using mostly constraints related to the usage of hardware resources. On NVIDIA GPUs, one of the main selection criteria is *occupancy*, i.e. the capability of the kernel to launch a big number of *Single Instruction Multiple Threads* (SIMT) threads. The pruning process identifies a few tens of kernels for each tile size. The final step of autotuning is benchmarking these kernels to find the best performing ones.

There are two differences between the kernels used here and the MAGMA kernels. MAGMA kernels operate on matrices in canonical FORTRAN 77 column-major layout, compliant with the *Basic Linear Algebra Subroutines* (BLAS) standard. The kernels used here operate on matrices in CRRB tile layout [2]. Also, MAGMA kernels are tuned for the case where all three input matrices are square, while the kernels used here are tuned for the *block outer product* operation in the LU factorization, i.e., $C = C - A \times B$, where the width of A and the height of B are equal to the matrix tile size *nb*.

DGEMM kernels achieve the best performance when texture reads are used for read-only data (A and B input matrices). The complete LU factorization applies matrix multiplications exceeding this limit by splitting them into a sequence of multiple DGEMM calls (two or three). Here the tuning is done for the largest case where texture mapping can be used without such splitting ($\sim$12K$\times$12K). Table 1 lists the performance of the autotuned kernels along with their most important tuning parameters (the blocking factors, i.e., the size of DGEMM performed by each multiprocessor in the outermost loop).

Table 1: Autotuned block outer product GPU DGEMM kernels.

TILE SIZE	32	64	96	128	160	192	224	256	288
BLOCKING	32×32×8	64×64×16	32×32×6	64×64×16	32×32×8	64×64×16	32×32×8	64×64×16	32×32×6
GFLOPS	208	250	255	272	265	278	269	277	274

3.4 Superscalar Scheduling

Manually multithreading the hybrid LU factorization would be tedious, given the three different levels of granularity involved: single tile, one column, a large block (submatrix). Here the scheduling infrastructure of the PLASMA library is used, namely the QUARK superscalar scheduler [9]. The LU factorization code is expressed with the canonical serial loop nest, where calls to CPU and GPU kernels are augmented with information about sizes of affected memory regions and directionality of arguments (IN, OUT, INOUT). QUARK schedules the work by resolving data hazards (RaW, WaR, WaW) at runtime. Two important extensions are critical to the implementation of the hybrid LU factorization: variable-length list of dependencies and support for nested parallelism.

CPU tasks, such as panel factorizations and row swaps, affect columns of the matrix of variable height. For such tasks, the list of dependencies is created incrementally, by looping over the tiles involved in the operation. It is a similar situation for the GPU tasks, which involve large blocks of the matrix (large arrays of tiles). The only difference is that here transitive (redundant) dependencies are manually removed, to decrease scheduling overheads, while preserving correctness.

The second crucial extension of QUARK is support for nested parallelism, i.e., superscalar scheduling of tasks, which are internally multithreaded. The hybrid LU factorization requires parallel panel factorization for the CPUs to be able to keep pace with the GPUs. At the same time, the ultra-fine granularity of the panel operations prevents the use of QUARK inside the panel. Instead, the panel is manually multithreaded using cache coherency for synchronization, and scheduled by QUARK as a single task, entered at the same time by multiple threads.

3.5 Communication

Each panel factorization is followed by a broadcast of the panel to all the GPUs. After each update, the GPU in possession of the leading leftmost column sends that column back to the CPUs (host memory). These communications are expressed as QUARK tasks with proper dependencies linking them to the computational tasks. Because of the use of lookahead, the panel factorizations can proceed ahead of the trailing submatrix updates and so can transfers, which allows for perfect overlapping of communication and computation, as further discussed in the following section.

4 Results

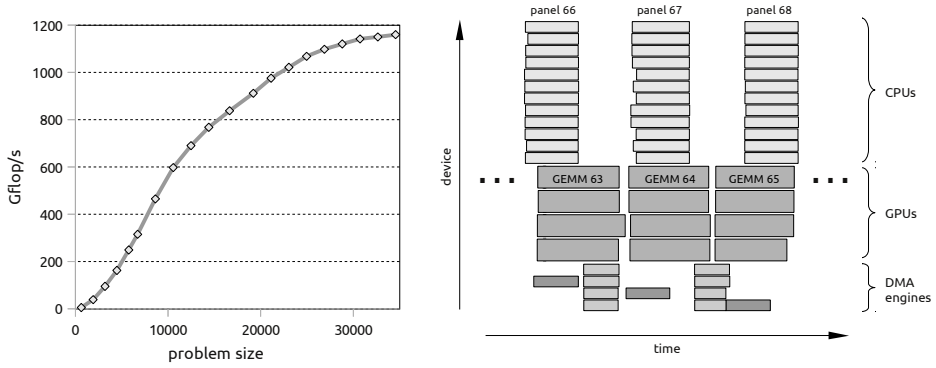
The system used for this work couples one CPU board with four sockets and one GPU board with four sockets. The CPU board is a H8QG6 Supermicro system with 4 AMD Magny Cours chips, 12 cores each, clocked at 2.1 GHz. The GPU board is an NVIDIA Tesla S2050 system with 4 Fermi chips, 14 multiprocessors each, clocked at 1.147 GHz.

The theoretical peak of a single CPU socket amounts to $2.1\ GHz \times 12\ cores \times 4\ ops\ per\ cycle \simeq 101\ Gflop/s$, making it ∼403 Gflop/s for all four CPU sockets. The theoretical peak of a single GPU amounts to $1.147\ GHz \times 14\ cores \times 32\ ops\ per\ cycle \simeq 514\ Gflop/s$, making it ∼2055 Gflop/s for all four GPUs. The combined CPU-GPU peak is ∼2459 Gflop/s.

The system runs Linux kernel version 2.6.35.7 (Red Hat distribution 4.1.2-48). The CPU part of the code is built using GCC 4.4.4. Intel MKL version 2011.2.137 is used for BLAS calls on the CPUs. The GPU part of the code is built using CUDA 4.0.

Figure 2a shows the overall performance of the hybrid LU factorization, and Table 2 lists the exact performance number for each point along with values of tuning parameters. Tuning is done by exhaustive search across all parameters. Matrix size goes up to 34,560. Beyond that point the the size of memory on all GPUs is exceeded. Each GPU can provide 2.6 GB of *Error Correcting Code* (ECC) protected memory.

Figure 2b shows a small fragment in the middle of a 23,040 run (the smallest size exceeding 1 Tflop/s performance). In the CPU part, only the panel factorizations are shown. The steps shown on the figure correspond to factoring submatrices of size ∼12,000. Due to the deep lookahead, panel factorizations on the CPUs run a few steps ahead of trailing submatrix updates on the GPUs. This allows for perfect overlapping of CPU work and GPU work. It also allows for perfect overlapping of communication between the CPUs and the GPUs, i.e., between the host memory



(a) Overall LU Performance

(b) 1 Tflop/s Trace Fragment

Figure 2: (a) Overall performance of the LU factorization. (b) Trace fragment of the run which exceeded execution rate of 1 Tflop/s.

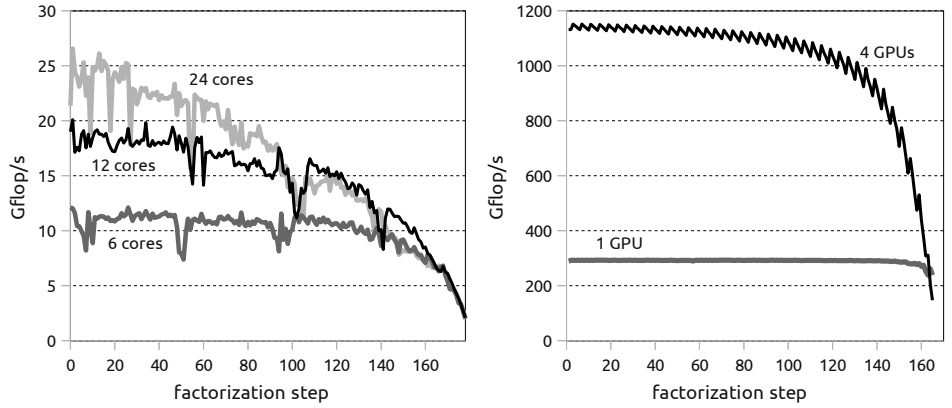
Table 2: LU performance and values of tuning parameters.

MATRIX SIZE [K]	0.6	1.9	3.2	4.5	5.8	6.7	8.6	10.6	12.5	14.4	16.6	19.2	21.1	23.0	25.0	26.9	28.8	30.7	32.6	34.6
TILE SIZE	64					96					128	192								
LOOKAHEAD	1					2			3	5	12						13	14		
PANEL CORES	12																			
GFLOPS	6	39	95	163	249	315	465	598	690	768	838	912	976	1022	1068	1098	1121	1142	1150	1160

and the device memories. Each panel factorization is followed by a broadcast of the panel to the GPUs (light gray DMA). Each trailing submatrix update is followed by returning one column to the CPUs (dark gray DMA).

Figure 3a shows the performance of the panel factorization throughout the largest run (34,560), using different numbers of cores, for panels of width 192. The jagged shape of the lines reflects the the fact that the panel cores have to compete for main memory with the other cores, applying updates at the same time. Generally, more cores provide higher performance, due to more computing power and larger capacity of their combined caches. However, 24 cores (two sockets) provide only a small performance improvement over 12 cores (single socket) due to the higher cost of inter-socket communication over communication within the same socket. In actual LU runs, the use of 12 cores turns out to always be optimal, even for large matrices. While 12-core panel factorizations are capable of keeping up with GPU updates, the remaining cores can be committed to CPU updates.

Figure 3b shows the performance of the GPU `DGEMM` kernel throughout the entire factorization. The gray line shows the `DGEMM` kernel performance on a sin-



(a) Panel on CPUs

(b) `DGEMM` on GPUs

Figure 3: (a) Performance of the panel factorization on CPUs at each step of the LU factorization. Panel width = tile size = 192. (b) Performance of the 4-GPU `DGEMM` task and performance of a single-GPU portion of that task.

gle GPU. The black line shows the performance of the 4-GPU `DGEMM` task. The jagged shape of the line is due to the load imbalance among the GPUs. The high peaks correspond to the calls where the load is perfectly balanced, i.e., the number of columns updated by the GPUs is divisible by 4. When this is not the case, the number of columns assigned to different GPUs can differ by one. The load imbalance can be completely eliminated by scheduling the GPUs independently. Although, potential performance benefits are on the order of a few percent.

5 Conclusions

The results reveal the challenges of programming a hybrid multicore system with accelerators. There is a disparity in the performance of the CPUs and the GPUs to start with. It turns into a massive disproportion when the CPUs are given the difficult (synchronization-rich and memory-bound) task of panel factorization, and the GPUs are given the easy (data-parallel and compute-bound) task of matrix multiplication. While the performance of panel factorization on the CPUs is roughly at the level of 20 Gflop/s, the performance of matrix multiplication on the GPUs is almost at the level of 1,200 Gflop/s (two orders of magnitude). The same disproportion applies to the computational power of the GPUs versus the communication bandwidth between the CPU memory and the GPU memory (host to device). The key to achieving good performance under such adverse conditions is overlapping of CPU

processing and GPU processing and overlapping of communication. The work also reveals that the PLASMA framework can easily adopt GPU acceleration, perhaps showing a path for the eventual merge of the PLASMA and MAGMA projects into a single cohesive multicore/manycore software package.

References

[1] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present and future. *Concurrency Computat.: Pract. Exper.*, 15(9):803–820, 2003. DOI: 10.1002/cpe.728.

[2] F. G. Gustavson, L. Karlsson, and B. Kågström. Parallel and cache-efficient in-place matrix storage format conversion. Technical Report UMINF 10.05, Department of Computer Science, Umeå University, 2010. `http://www8.cs.umu.se/research/uminf/reports/2010/005/part1.pdf` (accepted to ACM TOMS).

[3] PLASMA. `http://icl.eecs.utk.edu/plasma/`.

[4] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Dev.*, 41(6):737–756, 1997. DOI: 10.1147/rd.416.0737.

[5] A. M. Castaldo and R. C. Whaley. Scaling LAPACK panel operations using parallel cache assignment. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'10*, Bangalore, India, January 2010. ACM. DOI: 10.1145/1693453.1693484 (accepted to ACM TOMS).

[6] J. Kurzak, S. Tomov, and J. Dongarra. Autotuning GEMMs for Fermi. Technical Report UT-CS-11-671, Electrical Engineering and Computer Science Department, University of Tennessee, 2011. `www.netlib.org/lapack/lawnspdf/lawn245.pdf` (accepted to IEEE TPDS).

[7] R. C Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput. Syst. Appl.*, 27(1-2):3–35, 2001. DOI: 10.1016/S0167-8191(00)00087-9.

[8] MAGMA. `http://icl.eecs.utk.edu/magma/`.

[9] QUARK. `http://icl.eecs.utk.edu/quark/`.