Programming the LU Factorization for a Multicore System with Accelerators

Jakub Kurzak[1], Piotr Luszczek[1] and Jack Dongarra[1,2,3]

[1] University of Tennessee, Knoxville TN 37919, USA
[2] Oak Ridge National Laboratory, Oak Ridge TN 37831, USA
[3] University of Manchester, Manchester M13 9PL, UK
{kurzak, luszczek, dongarra}@eecs.utk.edu

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

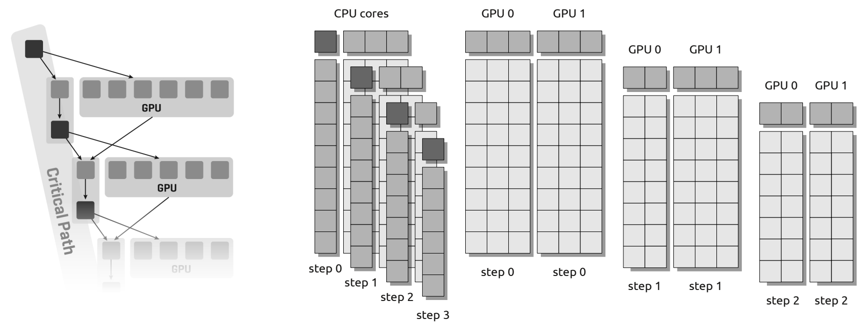
This paper presents an implementation of the canonical formulation of the LU factorization, which relies on partial (row) pivoting for numerical stability. It is equivalent to the `DGETRF` function in the LAPACK numerical library. Since the algorithm is coded in double precision, it can serve as the basis for an implementation of the *High Performance Linpack* benchmark (HPL) [2]. 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 factorizaton, 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 are the top nb rows of the trailing submatrix and C is the trailing submatrix withoug the top nb rows. Then the procedure is applied repeatedly, descending down the diagonal of the matrix.

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* and GPUs handling the coarse-grained data-parallel tasks outside of the critical path. Some number of columns (*lookahead*) is 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.



(a) Task Graph Scheduling (b) Work Partitioning

Fig. 1: Scheduling the task graph of 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.

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, where each tile occupies a continuous region of memory. Tiles are stored in column-major. Elements within tiles are store in row-major. Such layout is referred to as *Column-Row Rectangular Block* (CRRB) [4]. This layout is preserved on the CPU side (*host memory*) and the GPU side (*device memory*). The storage of elements by rows is critical to the performance of the row swap (`DLASWP`) operation on the GPUs. The storage of tiles by columns simplifies the communication of columns between the CPUs and the GPUs.

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 opetations: 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 row-major.

Panel factorization is a difficult workload. The LAPACK `DGETF2` function 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 a recursive-parallel panel factorization is used, providing an order of magnitude higher performance.

A recursive formulation of the panel factorization is the basis for the implementation [3]. The application of recursion allows to decrease memory intensity by introducing some degree of level 3 BLAS operations. Parallelization relies on splitting the panel vertically into a number of pieces. Each piece is handled by one core 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, what at some level resembles the technique of *Parallel Cache Assignement* (PCA) [1] 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 operatins requires very ligh-weight synchronization. Synchronization is implemented using 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`). `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 `DGEMM` on the GPUs. These straightforward implementations are sufficient to make the impact of the operations negligible in comparison to the `DGEMM`.

The `DGEMM` kernels are produced in the process of autotuning, similar to the one used in the MAGMA project. The system is called *Automatic Stencil TuneR for Accelerators* (ASTRA) [5] and follows the principles of *Automated Empirical Optimization of Software* (AEOS), popularized by the *Automatically Tuned Linear Algebra Software* (ATLAS) project [6].

The kernel is expressed through a parametrized *stencil*, creating a large search space of possible implementations. The search space is agressively 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 of 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 [4]. 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. Tuning is done for the largest C that can be mapped to a texture ($\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 non-trivial. It would be a challenge to track dependencies without automation, given the three different levels of granularity involved: single tile, one column, a large block (submatrix). Here the QUARK superscalar scheduler is used for automatic dependency tracking and work scheduling. 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 corectness.

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.

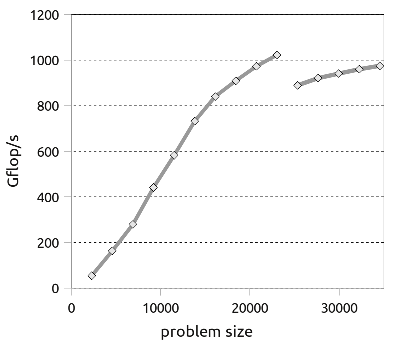
4 Results

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

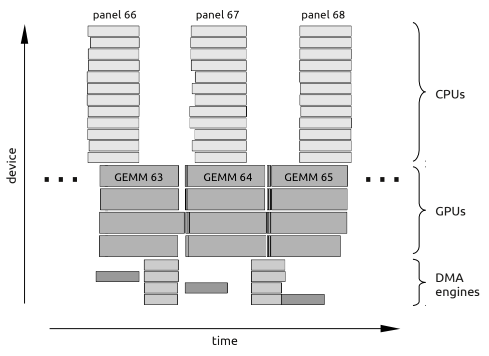
The theoretical peak of a single CPU socket amounts to $2.1\ GHz \times 12\ \textit{cores} \times 4\ \textit{ops per cycle} \simeq 101\ Gflop/s$, making it $\sim$403 Gflop/s for all four CPU sockets. The theoretical peak of a single GPU amounts to $1.147\ GHz \times 14\ \textit{cores} \times 32\ \textit{ops per cycle} \simeq 514\ Gflop/s$, making it $\sim$2055 Gflop/s for all four GPUs. The combined CPU-GPU peak is $\sim$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 manual orthogonal search, i.e., tuning tile size with all other parameters fixed, then tuning the lookahead depth, then tuning the number of cores used for the panel factorization and reiterating. The discontinuity between 23K and 25K is caused by abandoning the use of texture caches. At this point the matrix exceeds the mazimum size of a 1D texture of 2^{27} (1 GB). The chart continues until 35K. Beyond that point the size of the GPU memory, with ECC protection, is exceeded (2.6 GB).



(a) Overall LU Performance



(b) 1 Tflop/s Trace Fragment

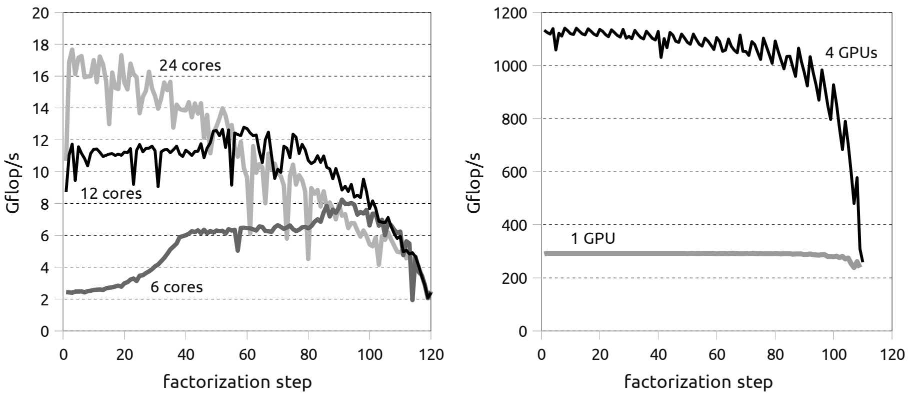
Fig. 2: (a) Overall performance of the LU factorization. (b) Trace fragment of the run which exceeded execution rate of 1 Tflop/s.

Figure 2b shows a small fragment in the middle of the execution trace of the 1 Tflop/s run. In the CPU part, only the panel factorizations are shown. The entire run factors a matrix of size 23K. The steps shown on the figure correspond to factoring submatrices of size ∼12K. 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. 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 of one column to the CPUs (dark gray DMA).

Table 2: LU performance and values of tuning parameters.

MATRIX SIZE [K]	2.3	4.6	6.9	9.2	11.5	13.9	16.2	18.6	20.7	23.0	25.3	27.6	30.0	32.3	34.6
TILE SIZE	96			128	160				192						
LOOKAHEAD	1	1	2	2	3	4	6	5	6	8	9	12			
PANEL CORES	6	12													
GFLOPS	54	163	279	441	582	732	840	909	973	1023	889	921	941	960	975

Figure 3a shows the performance of the panel factorization throughout the entire run, using different numbers of cores, for panels of width 192. The jagged shape of the lines reflects the unpredictable nature of a cache-based CPU memory system. The black line corresponds to the 1 Tflop/s run, for which 12 cores are used (one socket). Six cores deliver inferior performance due to smaller size of their combined caches, which cannot hold tall panels at the beginning of the factorization. 24 cores deliver superior performance for tall panels, and slightly lower performance for short panels, due to increased cost of inter-socket communication. It turns up that the use of 12 cores is more efficient, even for large matrices. 12-core panel factorizations are capable of keeping up with GPU `DGEMM`s, while the remaining cores are commited to CPU `DGEMM`s. As long as panel factorizations can execute in less time than GPU `DGEMM`s, it is better to free up more cores to do CPU `DGEMM`s. At the same time, decreasing the number of panel cores to six, would quadruple the time of the initial panel factorizations (Figure 3a), causing disruptions in the flow of the GPUs work (Figure 2b).



(a) Panel on CPUs (b) `DGEMM` on GPUs

Fig. 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 signle-GPU portion of that task.

Figure 3b shows the performance of the GPU `DGEMM` kernel throughout the entire factorization. The gray line shows the `DGEMM` kernel performance on a single 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. The two small dips on the left side of the line are due to random phenomena (jitter of unknown source).

5 Conclusions

The results reveal the challenges of programming a hybrid multicore system with accelerators. There is a disparity in 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 12 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.

References

1. Castaldo, A.M., Whaley, R.C.: Scaling LAPACK panel operations using parallel cache assignment. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'10. ACM, Bangalore, India (January 2010), DOI: 10.1145/1693453.1693484 (submitted to ACM TOMS)
2. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK benchmark: Past, present and future. Concurrency Computat.: Pract. Exper. 15(9), 803–820 (2003), DOI: 10.1002/cpe.728
3. Gustavson, F.G.: 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
4. Gustavson, F.G., Karlsson, L., Kågström, B.: Parallel and cache-efficient in-place matrix storage format conversion. Tech. Rep. UMINF 10.05, Department of Computer Science, Umeå University (2010), `http://www8.cs.umu.se/research/uminf/reports/2010/005/part1.pdf` (submitted to ACM TOMS)
5. Kurzak, J., Tomov, S., Dongarra, J.: Autotuning GEMMs for Fermi. Tech. Rep. UT-CS-11-671, Electrical Engineering and Computer Science Department, University of Tennessee (2011), `www.netlib.org/lapack/lawnspdf/lawn245.pdf` (submitted to IEEE TPDS)
6. Whaley, R.C., Petitet, A., Dongarra, J.: Automated empirical optimizations of software and the ATLAS project. Parellel Comput. Syst. Appl. 27(1-2), 3–35 (2001), DOI: 10.1016/S0167-8191(00)00087-9