Multi-GPU Implementation of LU Factorization

Yulu Jia\textsuperscript{a}, Piotr Luszczek\textsuperscript{a,1}, Jack Dongarra\textsuperscript{a,b,c}
\textsuperscript{a}University of Tennessee Knoxville, USA
\textsuperscript{b}Oak Ridge National Laboratory, USA
\textsuperscript{c}University of Manchester, UK

Abstract

LU factorization is the most computationally intensive step in solving systems of linear equations. By obtaining first the LU factorization of the coefficient matrix, we then may readily solve the system using backward substitution. The computational cost of LU factorization in terms floating point operations is cubic. There are various efforts to improve the performance of LU factorization. We propose a multi-core multi-GPU hybrid LU factorization algorithm that leverages the strengths of both multiple CPUs and multiple GPUs. Our algorithm uses some of the CPU cores for panel factorization, and the rest of the CPU cores together with all the available GPUs for trailing submatrix updates. Our algorithm employs both dynamic scheduling and static scheduling. Experiments show that our approach reaches 1134 Gflop/s with 4 Fermi GPU boards when combined with the total of 48 CPU cores from AMD. This is the first time such level of performance have been reported in a shared memory environment. Execution trace shows that our code also achieves good load balance and high system utilization.

Keywords: LU factorization, hardware accelerators, hybrid, multi-core multi-GPU

1. Introduction

General Purpose Graphic Processors (GPGPU) greatly advanced the arithmetic operation speed compared to conventional CPUs. GPUs’ superb performance renders them as the main choice for use in mathematical libraries. The LU factorization is a common method used in solving linear systems. Traditional LU factorization codes that used only CPU cores are being slowly superceded by hardware accelerated implementations that benefit from the emergence of GPGPUs. Such codes migrate portion of the work involved in LU factorization to the GPU with a significant performance gain. In this work, we extend current implementations of single CPU single GPU to multi-GPU multi-core platforms. Our algorithm is somewhat reminiscent of the block LU factorization used in LAPACK [1] with a drastic reimplementation that stresses parallelism at all hardware instances and alleviates inherent bottlenecks of bandwidth and latency between the computational components. The work is divided between the CPU cores and the GPUs in a non-uniform and asymmetric fashion. A number of CPU cores are used to factorize the panel in parallel, the rest of the CPU cores and all the GPUs are used for the trailing submatrix updates. Our algorithm also uses look-ahead to

\[\text{This work was supported by NSF through through grant 1038814.}\]
\[\text{Email addresses: yjia@utk.edu (Yulu Jia), luszczek@eecs.utk.edu (Piotr Luszczek), dongarra@eecs.utk.edu (Jack Dongarra)}\]
\[\text{1Corresponding author}\]
overlap panel factorization and trailing submatrix update. We employ both static work scheduling and dynamic work scheduling. We tuned the algorithm for a system with 48 AMD 6172 CPU cores and 4 Fermi GPUs.

The rest of the paper is organized as follows: Section 2 introduces the block LU algorithm. In Section 3, we describe the design of our algorithm and the techniques that we use such as data distribution, work scheduling, and look-ahead. Section 4 shows the method for validation of the numerical results. Section 5 presents the implementation details of the algorithm. We discuss the performance of the algorithm in Section 6 and Section 8 concludes.

2. Block LU Algorithm

The LU factorization of a matrix $A$ generates a lower triangular matrix $L$ and an upper triangular matrix $U$. The factorization also generates a permutation matrix $P$ that helps with numerical stability of the process. These matrices satisfy the following equation: $A = PLU$. This factorization is unique with the requirement that the diagonal entries of $L$ are all 1’s. LAPACK[1] and ScaLAPACK [2] implement a right-looking block LU algorithm with partial pivoting. The matrix is partitioned into blocks of size $NB$ (called a blocking factor). This algorithm iterates over diagonal entries of $A$. In every step, one block row and one block column are processed, and the square trailing submatrix is updated. When the factorization is in progress, the trailing submatrix is factorized as follows:

$$
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix} =
\begin{bmatrix}
P & 0 \\
L_{11} & L_{12}
\end{bmatrix}
\begin{bmatrix}
U_{11} & U_{12} \\
0 & U_{22}
\end{bmatrix} =
\begin{bmatrix}
P & 0 \\
L_{11}U_{11} & L_{11}U_{12}
\end{bmatrix}
\begin{bmatrix}
L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22}
\end{bmatrix}
$$

which gives: $U_{12} = L_{11}^{-1}A_{12}$ and $\tilde{A}_{22} = L_{21}U_{12} + L_{22}U_{22}$. Then the same process is again applied to $\tilde{A}_{22}$. This process continues until $\tilde{A}_{22}$ becomes a square $NB$ by $NB$ matrix which is factored with a unblocked code. The input and output of this algorithm can be shown as: $A \rightarrow P, L, U$

3. Multicore Multi-GPU LU Factorization

GPGPUs – commonly programmed using Single Instruction Multiple Threads (SIMT) paradigm – are the most suitable for algorithms that are rich in data parallelism. The theoretical peak of a single GPU exceeds by far the theoretical peak of a single CPU. The MAGMA project [3] is an effort to integrate GPUs with CPUs to carry out linear algebra operations and provide functional replacement of LAPACK on such hybrid hardware. Due to the huge performance difference between the CPU and GPU, coordinating the two different components poses a challenge, in the sense of scheduling tasks, synchronizing the components, data distribution, load balancing, and communication between different components. In our implementation, we adopted one-dimensional column cyclic data distribution between the CPU cores and GPUs. We also used a mix of static and dynamic scheduling. The computational tasks are statically scheduled between the CPU cores and GPUs and dynamically scheduled among the CPU cores.

3.1. Hybrid LU Factorization

CPU cores are general purpose execution units, they exhibit reasonable performance for varied instruction mixes that include arithmetic, logic, and branch operations. On the other hand, the GPUs, despite their “General Purpose” moniker, are especially good at floating point operations on large and regular data sets without branches. That being said, we formulate our code to combine the advantages of these two kinds of devices and try to minimize the negative impact of disadvantages by only giving each device the proper type of tasks. Panel factorization, a Gaussian elimination on tall and skinny matrices, is faster on CPU cores than on GPUs [4], while the GPUs are good in matrix-matrix multiplication – a Level 3 BLAS routine called DGEMM [5, 6]. Consequently, it is beneficial to let the CPU cores do the panel factorization and leave the DGEMM and DTRSM to the GPUs.

3.2. Data Layout

The original input matrix $A$ is stored in column major format conforming to the Fortran and LAPACK standards. $A$ is block partitioned into blocks of size $NB \times N$. Each block column is assigned to a GPU or a group of CPU cores. The block columns assigned to the CPU cores are left untouched in the column-major format. The block columns assigned to the GPUs are transposed into the row major format because the pivoting process is very slow in column major format on the GPUs.
3.3. Data Distribution

The theoretical peak of one single GPU card is nearly 514 Gflop/s in double precision. CUBLAS DGEMM can achieve around 300 Gflop/s on one GPU. The theoretical performance of one CPU core is around 10 Gflop/s. In order to make the CPU cores keep up with the speed of the GPUs, the CPU cores take lesser amount of work load. The data distribution used in this paper is column-wise 1D block cyclic. If we use GPU1, GPU2 etc. to denote the consecutive GPUs and gCPU to denote a group of CPU cores, then the block columns are assigned in this order: [(GPU 1, GPU2, GPU3, GPU4)1, ..., (GPU1, GPU2, GPU3, GPU4)k, gCPU]. In other words one block column is assigned to the GPUs in a round robin manner for $k$ rounds, then a column block is assigned to the group of CPUs. This assignment repeats until all the block columns are taken. Figure 1 illustrates this data distribution in a greater detail.

3.4. Overlapping Communication and Computation

In our implementation, the whole input matrix is resident in the CPU memory (the main memory), while a part of the data is duplicated in GPU memory. These two memory realms form separate address spaces, and are physically connected by the PCIe bus. As the LU factorization progresses, we need to frequently send data back and forth between the CPU memory and the GPU memory because we use the CPU cores to do the panel factorization. When the panel factorization is done, the GPUs will need the result in order to calculate $U_{11}$, $U_{12}$ and to update $\tilde{A}_{22}$, so the factorized panel need to be transferred from the main memory to the GPU memory. When the panel to be factorized is resident on the GPU, we need to transfer this panel back to the main memory. The PCIe data link is too slow for the data communication which could become a dominating factor in the total execution time of the algorithm. To alleviate this penalty we use asynchronous data transfers to overlap the communication with computation on the GPUs.

Algorithm 1 illustrates the overlap. Once we initiate the data transfer we can continue to launch computation kernels on the GPU. The purpose of this overlapping is to prevent the GPUs from waiting for the CPU cores to do the factorization. Using this asynchronous data transfer, the data will be transferred back to the main memory and factorized there in parallel with computations on the GPU. The GPU will be busy all the time.

3.5. Algorithmic Look-ahead

Algorithm 1 GPU algorithm

for every iteration do
    for every block column resident on the GPU and to the right of the current panel do
        call DTRSM to calculate part of $U_{12}$
        call DGEMM to calculate part of $\tilde{A}_{22}$
        if this column is the next panel then
            initiate the asynchronous data transfer to send this finished column back to the main memory (when this data are in the main memory, the CPU cores will start to factorize them, thus overlap with computations on the GPU)
        end if
    end for
end for
Look-ahead is a technique related to overlapping communication with computation [8]. Using look-ahead can eliminate the idle time of the GPUs while they are waiting for the CPU cores to provide the result of the current panel factorization. Since the GPUs are fast compared to the CPU cores and are the main computational power in the LU algorithm we are proposing, we cannot afford to let them become idle. Look-ahead is essential to keep the GPUs busy as much as possible, hence, it boosts the performance of the entire algorithm. The High Performance LINPACK benchmark [9, 10] which is the benchmark to rank the top 500 computers in the world uses look-ahead to minimize the idle time of the CPU cores. Look-ahead rearranges the sequential execution order while satisfying the data dependencies between tasks. Without look-ahead, after the $k$-th panel is factorized, the next thing to do is to update the trailing submatrix using the factorized $k$-th panel. Then the algorithm goes on to factorize the $k + 1$-th panel. Now the cores in charge of the trailing submatrix update have to wait for the result of the $k + 1$-th panel factorization, causing idle time. In the case of look-ahead of depth 1, after the $k$-th panel is factorized, the next job is to update just the first block column of the trailing submatrix. When the first block column of the trailing submatrix is updated, it is used to do the $k + 1$-th panel factorization. At the same time the update of the rest of the trailing submatrix from the $k$-th iteration continues. By the time this update in the $k$-th iteration is done, the $k + 1$-th panel factorization is also finished. There is no need to wait any more hence improving the utilization of the computational power in the system. Figure 2 shows a small example of look-ahead of depth 1 [7]. In theory, we can use look-ahead of any depth up to the number of block columns of the input matrix. But in our case, the trailing submatrix update takes much more time than the panel factorization, one level of look-ahead is good enough to hide the panel factorization and only look-ahead of depth 1 is used.

3.6. Parallel Panel Factorization

Computational workload required by panel factorizations increases with the problem size, thus, if performed sequentially, it becomes a bottleneck to parallelization of the entire LU factorization. Even though the Amdahl fraction [11, 12] associated with this work diminishes for larger problem sizes it remains far too high for the smaller ones [13]. More importantly, the key to high efficiency is the overlap that, in the case of LU, hides both the sequential execution for the panels and the communication between CPUs and GPUs. Unfortunately, the overlap loses its effectiveness because the sequential panel factorization takes too long on a multi-GPU platform. And this is due to the accelerators’ many-fold performance advantage over CPUs. Simply put, GPUs run out of work updating from the previous panel long before the sequential panel factorization finishes for the next panel.

An alternative solution is to implement the panel factorization on GPUs. However, due to the algorithm’s latency-bound nature, it is hard to implement on the current generation of GPUs. Such an implementation (even if it was available and fast) is a problematic proposition due to a more fundamental issue. Namely, the fact that the CPUs alone do not contribute enough performance to take on the tasks left by the GPUs that are now busy with the panel factorization. In other words, a slow task should be assigned to the device that contributes the least to the progress of the factorization.

In our implementation, panel factorization relies on the combined size of caches – insufficient amount of such combined cache results in poor performance for very tall panels as shown in Figure 3 for a run on 6 cores. Another aspect revealed in Figure 3 is that the performance of the factorization increases slowly with panel height because of
3.7 Work Scheduling

Our algorithm uses both static scheduling and dynamic scheduling. A number of CPU cores are dedicated to the panel factorization. The rest of the CPU cores are used for trailing submatrix update. All the GPUs are used solely for trailing submatrix update. In fact, the CPU cores that perform the trailing submatrix update are combined and treated as a single GPU for the purposes of work distribution.

Static scheduling strategy is used when scheduling work among these GPUs (both the real GPUs and the virtual GPUs made by combining the CPU cores). As shown in Figure 1, the block columns of the input matrix are assigned to the GPUs in a 1-D block cyclic manner. All the block columns assigned to a GPU are resident on the GPU through out the whole process of the execution. As the current panel swipes through the matrix, each GPU keeps updating the part of the matrix to the right of the current panel. All the operations associated with a piece of data are carried out by the hosting GPU. Each GPU has less and less work to do as the current panel progresses from the left side of the matrix to the right side. There is no data movement between the GPUs during the execution of the algorithm.
Dynamic scheduling strategy is used when distributing work among the CPU cores which are doing the trailing submatrix update. Each block column assigned to the CPU cores is divided into square blocks of size $NB$ by $NB$. Each operation, i.e. DTRSM or DGEMM, on a square block is a task unit for scheduling. In each iteration of the factorization we maintain a queue for each block column owned by the CPU cores. When a task is ready, we insert this task into the ready queue. Each thread checks this queue and takes one task from this queue if there is any when it idles. The row swapping operation, DLASWP, on the whole block column is statically assigned to a single core.

4. Validating the Numerical Result

In order to validate the result of the factorization, we first generate an input matrix $A$ and a right hand side $b$ – the entries of both are drawn from a uniform random distribution. We use our LU factorization routine to factorize the input matrix $A$ and solve the linear system $Ax = b$ using DGETRS from LAPACK which generates the solution vector $x$.

The correctness of the our factorization result is then verified using the following formula:

$$\frac{\|Ax - b\|_\infty}{\|A\|_\infty \|x\|_\infty n \epsilon} \leq O(1)$$

where $n$ is the matrix size and $\epsilon$ is the machine precision ($2^{-53}$ for the IEEE format double precision numbers). We consider the factorization correct if $x$ satisfies the inequality above.

5. Implementation

We implemented the algorithm using POSIX threads and CUDA 3.2. The input matrix is assigned to the GPUs and a group of CPU cores using the 1-D block column cyclic distribution. The block size $NB$ is chosen to be 256. All the block columns are sent to the GPU before the factorization starts. On each GPU the block columns are stored contiguously in order to save space and improve the read write performance. Also the matrix is stored in row-major format on the GPUs to make the pivot row swapping process fast.

5.1. Synchronization between the Panel Factorization and the Trailing Submatrix Update

A semaphore panel_ready is used to indicate the availability of the next panel for factorization. If the value of panel_ready is 1, it means the next panel is ready to be factorized. If its value is 0, it means the next panel is not ready yet. The value of panel_ready is set to 1 when the factorization begins. The CPU cores in charge of the panel factorization first perform a $P$ operation on panel_ready, then start to factorize the current panel. When the next panel to be factorized is updated, the thread who owns that panel performs a $V$ operation on panel_ready which sets its value back to 1, so the panel cores can continue with the next panel factorization.

An array of flags is used to indicate the availability of the factorized panel. For each block column there is a corresponding entry in this array. The flags are all set to 0’s initially. When the panel cores finish factorizing a panel, the corresponding flag is set to 1 meaning that it can be used to update the trailing submatrix. The GPUs and CPU cores doing the trailing submatrix update check the value of the flags with busy-waiting. When they observe the value of the flag change to 1, they continue to update the trailing matrix using the corresponding factorized panel.

5.2. Coordination between the CPU Cores Updating the Trailing Submatrix

A number of block columns are assigned to the CPU cores for update. Each of these block columns are assigned to a subset of the CPU cores. Each block column is logically partitioned into square blocks of size $NB$ by $NB$. An update operation (DTRSM or DGEMM) on one block is considered a task. Two task queues are maintained for each block column, namely the nonready queue and the ready queue. At the beginning of each iteration, one core constructs the nonready queue. After the row swap and the DTRSM at the top of the block column are finished, the tasks in the nonready queue are moved to the ready queue. The cores assigned to this block column check the ready queue using busy-waiting. Every core takes one task from the ready queue and performs the associated computation if the ready queue is not empty. Otherwise the busy-waiting continues.
Figure 4: Performance results of our algorithm: Performance of the algorithm using all the cores and GPUs (left) and performance of the algorithm with different number of GPUs (right).

Figure 5: Execution trace of the algorithm using all the cores and GPUs. The black line on top is the parallel panel factorization. The four lines below it are the 4 GPUs. The 42 blue lines at the bottom are the CPU cores doing the trailing submatrix update. Three portions of the trace are shown: the beginning of the execution trace (top), the middle part of the execution trace (middle), and the end of the execution trace (bottom).
6. Performance Evaluation

We implemented the proposed LU factorization algorithm in double precision. We give the scalability performance of the algorithm as well as the efficiency of the algorithm in this section. The efficiency is measured in terms of the theoretical peak performance and the performance of DGEMM on the machine. The experiments were carried out on a 48 CPU core shared memory system that featured 4 Fermi GPU boards connected to the system via PCIe. Table 1 details the specifications of the experimental environment and compares the performance of the CPU cores and GPUs. The input matrices in all the experiments are square. Their elements are generated using a random number generator with a uniform distribution between 0 and 1.

6.1. Scalability

We carried out experiments to investigate the performance of our algorithm factorizing the largest possible matrix allowed by the memory system. We also carried out experiments to measure the strong scalability and weak scalability of the algorithm. In strong scalability, we fix the problem size and vary the number of GPUs utilized in the algorithm. In weak scalability, we fix the workload size for every GPU while changing the number of GPUs used in the algorithm.

Figure 4 shows the performance of the algorithm in double precision. In this experiment we use all the available 4 GPUs and 48 CPU cores in the system while increasing the problem size until there is not enough memory to accommodate the matrix in the combined GPU memories. The main memory on the host side is not a limiting factor since as it is much bigger than that of the GPUs (124 GiB versus 3 GiB). The tested matrix sizes start at around 2000, and increase until they reach over 32000. Larger sizes exceed the capacity of the combined GPU memories. The performance increases along with the increase of the matrix size. Figure 5 shows the execution trace of the code. Since the entire trace is too long to be shown here, we excerpt three representative segments of the trace. Figure 5 shows the beginning of the trace, the middle part of the trace, and the end of the trace. From the figure, we observe that the GPUs are busy all the time in the beginning and in the middle part of the execution. There are some idling gaps in the GPU traces at the end of the execution due to the small amount of work left on the GPUs. The GPU works fast on large data inputs, when the data size drops, its speed also drops. In our case, this causes the GPUs which are updating the trailing submatrix to wait for the cores which are factorizing the current panel. There are gaps in the CPU core traces which is acceptable because a CPU core contributes much less computational power compared to a GPU. The algorithm achieves 1134.567 Gflop/s at matrix size around 32000. To our best knowledge, this is the first time such performance levels have been achieved in a shared memory system.

Figure 6 shows the weak scalability of the algorithm in double precision. In this experiment we vary the number of GPUs used in the algorithm while fixing the workload size of each GPU. The matrix sizes used here are 10752, 17920, 25088, 32256. We can see that the total performance scales up almost linearly when we increase the number of GPUs.

Figure 6 also shows the strong scalability of the algorithm in double precision. In this experiment we vary the number of GPUs used in the algorithm while fixing the total problem size. The matrix sizes used here are 21504,
21760, 21504, 23040. We can see that the total performance goes up almost linearly when we increase the number of GPUs.

6.2. Efficiency

The total theoretical peak performance of the system – 48 CPU cores and 4 GPUs – is 2463.2 Gflop/s. The total performance for DGEMM on the system is 1558.8 Gflop/s. Our algorithm achieves 1134.567 Gflop/s for a matrix size 32256, which is 46.06% of the theoretical peak, and 72.78% of the peak DGEMM performance.

7. Related Work

CUBLAS implemented the standard BLAS and some basic LAPACK routines on the GPUs [17]. The MAGMA [3] project implemented hybrid matrix factorization routines including the hybrid LU factorization. Originally MAGMA only had one-core and one-GPU implementations. The newest release (i.e. MAGMA 1.1) implemented multi-core and multi-GPU matrix factorization routines. CULA [18] implemented a hybrid processing model and has LU decomposition, QR decomposition, SVD, linear system solver, least square problem solver. Agullo, Augonnet, Dongarra et al. implemented CPU/GPU hybrid LU kernels using the tile LU algorithm [19]. Tomov et al. implemented hybrid LU, QR, and Cholesky factorization routines and dense linear algebra solvers based on these routines [20]. None of these efforts, however, broke the barrier of 1000 Gflop/s that we have managed to achieve in our implementation.

Panel factorization has been successfully parallelized and incorporated into a general LU factorization code [13] using an optimized implementation of mostly Level 1 BLAS. This was done in a flat parallelism model with Block Synchronous Processing (BSP) model [21] also referred to as fork-join execution. The authors refer to their approach as Parallel Cache Assignment (PCA). Our work on parallelizing the panel factorization [22, 23, 24] differs in a few key aspects. We employ recursive formulation of the factorization [14] and therefore are able to use Level 3 BLAS as opposed to just Level 1 BLAS. Another important difference is the nested parallelism with which we have the flexibility to allocate only a small set of cores for the panel work while other cores carry on with the remaining tasks such as the Schur complement updates. Finally, we use dynamic scheduling that executes fine grained tasks asynchronously, which is drastically different from a BSP or fork-join parallelism.

8. Conclusion and Future Work

LU factorization is an important step to solve dense systems of linear equations. When the system size is large, the speed of LU factorization becomes the main concern when solving such systems. A hybrid LU factorization algorithm that utilizes all the CPU cores and GPUs in a system is presented in this paper. This algorithm uses heterogeneous data layout across the CPU cores and GPUs, distributes data among different computing units according
to their computational power. It employs static work scheduling together with dynamic work scheduling, and overlaps computation with computation using look-ahead. Experiments show that this algorithm achieves 1134.567 Gflop/s on a system with 48 AMD 6172 cores and 4 NVIDIA Fermi GPUs. Currently the memory size on the GPU is a limiting factor of the problem size we are able to accommodate. Further research will use GPU non-resident memory technique to overcome this limit by moving data in and out of the GPU memory during execution. It’s also interesting to experiment with LU on GPU enabled distributed memory platforms.

References

17. CUBLAS Library (August 2010).
22. J. Dongarra, M. Faverge, H. Ltaief, P. Luszczek, Exploiting Fine-Grain Parallelism in Recursive LU Factorization, Accepted at the International Conference on Parallel Computing.