Applied Mathematics, Modeling and Computer Simulation
C.-H. Chen et al. (Eds.)
© 2022 The authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/ATDE221053

High-Performance Batched LU Decomposition on GPU

Xinli LEI, Xuesong ZHANG[1], Lin MA, Tie BAO
College of Computer Science and Technology, Jilin University, Changchun, Jilin, China

Abstract. LU decomposition is an important computational step in many engineering and scientific computing problems. In most of critical applications, many small-scale problems need to be solved instead of a few large linear systems. However, when facing with small or medium sized matrices, existing batched LU decomposition algorithms suffer from the global memory access latency bottleneck, and the performance is poor. We implement a series of specialized optimized batched GPU-based LU decomposition algorithms for this situation, and two outperforming algorithms are selected after a systematic testing. They both achieved speedup ratio greater than 3 compared with cuBLAS, and even greater than 10 in some cases.

Keywords. LU Decomposition, Parallel Computing, Small and Medium-sized Matrix, Ordinary Differential Equations, Graphic Processing Unit

1. Introduction

LU decomposition is the most important computational step in many engineering and scientific computing problems which has a wide range of applications in machine learning, image processing, information encryption and high-speed information transmission. With the continuous development of computer science, the amount of data that needs to be processed increases exponentially, so how to use computers to process experimental data efficiently and quickly becomes a problem. Some early GPU-based parallel LU decomposition solvers include [1, 2, 3]. Peng et al. (2020) [4] proposed a GPU-based sparse LU decomposition solver in 2020, called GLU3.0, which is an improved version of GLU1.0/2.0 using a hybrid right-looking LU decomposition algorithm . Zhou (2016) [5] performed LU decomposition on huge and complex matrices based on the idea of parallel computing and transformed three optimal parallel computing processing methods using three parallel modes (OpenMP, MPI, PPL). Branko et al. (2017)[6] proposed using multiple GPUs in parallel to accelerate in-kernel LU decomposition and using blocks to overcome the memory limitations of GPUs. Volkov et al. (2009)[7] implemented right-looking algorithms for LU, Cholesky, and QR for GPUs, which are similar to those in the GPU-based advanced linear algebra library MAGMA (2014)[8], changing the data layout by transposing matrices. MAGMA also provides implementations of batch GEMMs on NVIDIA GPUs and CPUs for mid-range

1 Corresponding Author, Xuesong ZHANG, High-tech Development Zone, Changchun City, Province Jilin; E-mail: wetting@jlu.edu.cn.

and very small matrix sizes, optimized at the register operation level. Oreste et al. (2013)[9, 10] achieved a batch LU target of up to 128×128 on GPU. In their implementation, a single CUDA thread was used to solve a linear system of equations, but their solution was not faster than the NVIDIA batch LU implementation of cuBLAS (2017)[11]. In 2014, Dong et al. (2014)[12] proposed a GPU-based batch LU decomposition using a multi-level block right-looking algorithm that preserved the data layout but minimized the loss of partial rotations. In 2019, Mengyue (2019)[13] published a GPU-based batch LU decomposition algorithm . They executed logically complex and single-threaded tasks on the CPU and all logically simple and highly parallelized tasks on the GPU, thus reducing the data transfer time between the CPU and GPU and improving the running speed of the algorithm .

In this paper, we implement a series of GPU-based LU parallel decomposition algorithms for small and medium-sized matrices which are improved or refined in a step by step manner. The final performance result compared with the batched LU version of cuBLAS shows that our implementation has better advantages, and under some configurations, the best speedup ratio is greater than 10.

The remainder of the paper is organized as follows. Section 2 presents the background of the work. In section 3, the LU parallel decomposition algorithms are described in detail. Section 4 shows the performance results of each algorithm. And Section 5 comes with a conclusion.

2. Background

We design and implement a pipeline-based model solving task dispatching framework on CPU and GPU, which is mainly used to solve ordinary differential equations in parallel to improve computing efficiency. The core idea of this framework is to dynamically generate ODE solving tasks and schedule them to the corresponding hardware according to the solution settings. It consists of three main components: task generator, scheduler and synchronizer. At present, we have implemented GPU-based hybrid integration (Implicit-Explicit) and RIDC algorithms in this framework. However, for each step in the implicit method, the Newton-Raphson method has to be used to solve the linear equations, and the LU decomposition is the key step whose efficiency has great impact on the overall performance of the algorithm.

The existing Basic Linear Algebra Subroutines (BLAS) and Linear Algebra Package (LAPACK) libraries have been the foundation of the usual high-performance computing (HPC) software stack chain. But these architecture-dependent libraries (e.g. NVIDIA cuBLAS libraries for GPU) face major performance bottlenecks in some critical applications. Because in most of these ODE solving tasks, many small-scale ODEs need to be solved instead of a few large linear systems. Currently, high-performance implementations of batch linear algebra operations on GPUs targeting extremely small sizes are either poorly supported or not supported at all; meanwhile, for small matrices, the overhead of dealing with memory latency and data movement related to transposition are very expensive. Therefore, some special methods should be used to optimize the LU decomposition process for small and medium matrices.

The main contributions of this paper are as follows.

1. Design and implementation of different versions of the batched LU decomposition algorithms targeting at small-sized matrices, including their performance comparison.

2. Design and implementation of different versions of the batched LU decomposition algorithms targeting at medium-sized matrices, including their performance comparison.

3. According to these comparison results, the optimized LU decomposition configurations under different calculation parameters are obtained, so as to provide a reference support for the subsequent implicit GPU ODE solver.

3. LU Decomposition Algorithm

The proposed batched LU decomposition algorithm mainly focus on square dense matrices with dimension of power 2 ranging from 4 to 32. For matrix dimensions greater than 32 and less than 256, they are extended to the nearest multiple of 32 with identity diagonal elements and zero other elements in the padding area. The variable SysDim stands for the final extended sub-matrix dimension. Before the LU Decomposition process, all the batched small matrices are grouped into a large matrix in memory as is showed in figure 1. M and N are the dimension of the large matrix, and i, j represent the row and column index of the sub-matrix respectively. Sub-matrix is denoted by $A^{(g)}$, $g = i \times n + j$. For SysDim smaller than 32, the corresponding sub-matrix cannot be block partitioned further, thus the proposed decomposition methods are divide into two series:

Mini: $\mathrm{SysSim} \in \{4, 8, 16\}$.

Normal: $\mathrm{SysSim} = k \times 32, k \leq 8$.

For the Mini series, the entire sub-matrix can be cached into local registers, while for the Normal series, each sub-matrix cannot be cached as a whole, so it needs to be treated differently. The following part will describe these two series methods.

3.1. Mini-Naive Version

This version is straightforward, with each sub-matrix being assigned to one GPU thread. For each GPU thread, its processing logic is shown in Algorithm 1, where $p_k^{(g)}$ is the pivot index of the k'th row in sub-matrix $A^{(g)}$, and $a_{i,j}^{g}$ is the [i][j] element of $A^{(g)}$. When the algorithm finishes, the upper and lower triangle matrices in $A^{(g)}$ are used to store $U^{(g)}$ and $L^{(g)}$ matrices respectively.

Since the sub-matrix is too small, many optimization methods such as shared memory cannot be applied to this version. But the memory layout can be utilized to achieve global memory coalescing with better memory bandwidth, and this comes with the next version.

Algorithm 1: LU Decomposition (MINI_NAIVE)

$d \leftarrow SysDim$

$g \leftarrow ThreadIndex$

for k=0 to d-1

select $i \geq k$ *to maximize* $a_{i,k}^{g}$ *(find pivot index)*

$p_k^{(g)} = i$ *(save pivot index)*

$a_{k,1:d}^g \leftrightarrow a_{i,1:d}^g$ *(swap two rows)*

for j=k+1 to d-1

$a_{j,k}^g = a_{j,k}^g / a_{k,k}^g$ *(L update)*

$a_{j,k:d}^g = a_{j,k:d}^g - a_{j,k}^g a_{k,k:d}^g$ *(trailing update)*

end for

end for

end

3.2. Mini Global Coalescing Version

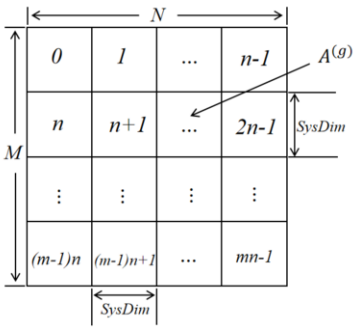

Figure 1. Basic Memory Layout

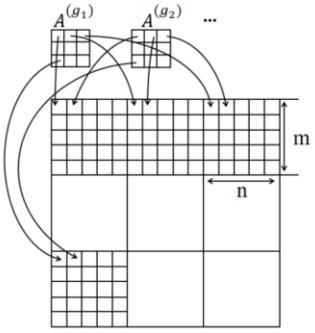

Figure 2. Global element combination layout

In this version, the memory layout of sub-matrix is shown in figure 2. The elements with the same index of all sub-matrices are rearranged together in the large matrix. For any two orderly adjacent sub-matrices $A^{(g)}, A^{(g+1)}$, their elements $a_{i,j}^g, a_{i,j}^{g+1}$ also form an adjacent relationship in the rearranged location. For any two elements of the same sub-matrix $A^{(g)}$, their memory location in the large matrix has the following relationship:

The column element stride between $a_{i,j}^g$ and $a_{i,j+1}^g$ is n;

The row element stride between $a_{i,j}^g$ and $a_{i,j+1}^g$ is $m \times N$.

The execution logic of this version is the same as that of Algorithm 1 except for the element stride of the sub-matrix. The address of $a_{i,j}^g = \text{addressof}(A^{(g)}) + i \times m \times N + j \times n$. In this version, because all threads of the same warp read or write adjacent data, the effect of memory coalescing access is obtained. Surely the performance of this version will outperform Algorithm 1, and the experimental results also confirm this.

3.3. Mini Register Version

For Kepler or later GPU architecture, the maximum number of 32bit registers allocated by each thread is 255. When the dimension of the sub-matrix is 4 or 8, the total number of elements is only 16 or 64. Therefore, each thread can load all the data of the sub-matrix into its thread specific register array before LU decomposition, and write it back to the main memory after LU decomposition. This comes with Algorithm 2.

Algorithm 2: LU Decomposition (MINI_ REGISTER)

$d \leftarrow SysDim$

$g \leftarrow ThreadIndex$

T regA[d][d] *(register cache)*

$regA \leftarrow A^{(g)}$ *(read into registers)*

LU decomposition on regA

$A^{(g)} \leftarrow regA$ *(write back to global memory)*

3.4. Mini Register Share Version

For double precision data, each element needs two 32-bit float registers to store. Limits by the maximum registers per thread can allocate, Algorithm 2 cannot deal with 16×16 case. When faced with sub-matrix of dimension 16, the number of cooperating threads for each one must be increased.

In Algorithm 3, each thread is only responsible for one column of data in a sub-matrix. All the threads targeting the same sub-matrix $A^{(g)}$ form a group of size SysDim. When performing row swap or trailing update, the pivot column is shared by all threads in the same group. Therefore, a shared memory sData should be allocated to store this information.

So far, we have implemented 4 different versions of the batched LU decomposition algorithm for small-sized matrices and their performance results are list in Section 4.

Algorithm 3: LU Decomposition (MINI_ REGISTER)

$d \leftarrow SysDim$

$col \leftarrow ThreadIndex\%d$ *(column index in current group)*

$g \leftarrow threadIndex/d$ *(group index)*

share T sData [d] *(share data to store pivot column)*

T regCol[d] *(register cache for one column)*

$regCol \leftarrow A^{(g)}_{1:d,col}$ *(cache column data into regCol)*

for k=1 to d-1

if k == col

pivot ← *findpivot(regCol,k)* *(find pivot in regCol)*

broadcast pivot in current group

sData ← *regCol* *(write pivot column to shared memory)*

end if

swap(regCol, k , pivot) *(swap two elements between the k'th col and the pivot col)*

TrailUpdate(sData, regCol) *(trailing update column data)*

$A^{(g)}_{1:d,col} \leftarrow regCol$ *(write back to global memory)*

end

3.5. Normal Naive Version

In this part, we focus on sub-matrix of size greater than 16. Due to the larger dimension size, it requires multiple rounds of blocked LU decomposition to accomplish the whole process. There are two major steps in each iteration:

1. Find pivot, row swap, pivot column update of current block (like L update in Algorithm 1).
2. Trailing update the other part of the sub-marix.

Algorithm 4 and 4.1 show the details. In this version, GROUPSIZE is fixed by 16, i.e. each sub-matrix is processed by 16 GPU threads, and "diag" is an input parameter, stands for the pivot row that the algorithm is currently processing.

Algorithm 4: LU Decomposition Pivot Finding, Swapping and L Updating (NORMAL_NAIVE)

$d \leftarrow SysDim$

$g \leftarrow GroupIndex$

$s_0 \leftarrow$ *subindex of current thread in group g*

$pivot \leftarrow \left\{ s_i \mid max(a^g_{s_i,diag}, i \in (0,1,\ldots,SysDim/GROUPSIZE)) \right\}$

pivot ← *reduce(pivot)* *(find pivot in final warp of this group)*

swap(pivot, diag) *(elements swap between the pivot row and the diag row with element stride=GROUPSIZE)*

$a^g_{s_i,diag} /= a^g_{diag,diag}$ *(L update by all threads in the group with element stride= GROUPSIZE)*

Algorithm 4.1: LU Decomposition R1_update (NORMAL_NAIVE)

r1_update(P1, diag) *(trailing update P1)*

r1_update(P2, diag) *(trailing update P2)*

r1_update(P3, diag) *(trailing update P3)*

In Algorithm 4.1, in order to achieve global memory coalescing as much as possible when executing trailing update, the update process is divided into three areas, as shown in figure 3. Inside each region, trailing updates are processed by multi threads with one column per thread.

3.6. Normal Opt Version

Algorithm 5 (NORMAL_OPT) is simplified on the basis of Algorithm 4: Within each sub-matrix, a single thread is used to process the pivot finding, row swap, and L updating. Then, carry out the trailing update in multi threads manner. As shown in figure 3, the trailing update is also partitioned into three areas, P1, P2, and P3. For region P2, cols are split into the sum of multiple powers of 2 (such as $cols = 2^0 + 2^1 + \ldots + 2^k$). The update of P2 is also split into k individual schedules. In each schedule of number i, 2^i columns are updated by 2^i threads. In P3, update is performed in bock manner with each blocksize = 16 or 32. This memory-aligned accessing mode will lead to a slightly improved performance. Due to the length limitation of this paper, Algorithm 5 is omitted.

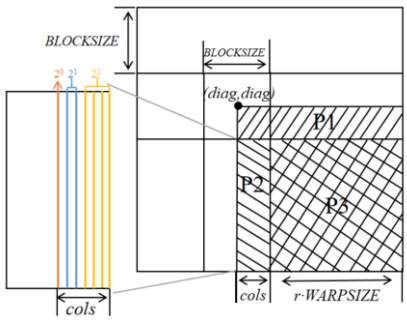

Figure 3. Global element combination layout

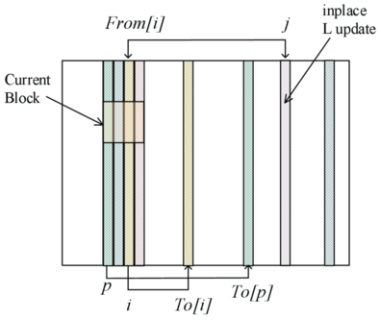

Figure 4. *From* and *To* in which *From[i]* ≠ *To[i]*.

3.7. Out of Order Version

After carrying some preliminary test on Algorithm 4 and Algorithm 5, we find the most time-consuming work lies in two parts, the row swap and trailing update. Each time a pivot is selected, the row swap is carried out immediately. If we record the position of the pivot, the row swap can be postponed until all pivot rows of current block have been selected. And if we reorder the sub-matrix to column-major mode, in which the pivot row becomes the pivot column, we will get the global memory coalescing effect when

swapping all the pivot columns of a block simultaneously. Therefore, in the following algorithm, we will assume that the sub-matrix is column-major.

Because the swap of all pivot columns is postponed until all pivots in the block have been decided, the L update(shown in Algorithm 1) is carried out in an out-of-order mode. That is, for the i' th pivot column of index j, the update L update is performed directly on column j before it is swapped to its final position i.

In order to record the position relationship of the pivot column index before the final swap, two kinds of movement information need to be maintained, namely From and To. From[i] stands for which column the i' th pivot column comes from. To[i] stands for position to which the column index i will eventually be moved. Since the same column may be moved multiple times during pivot swap, From[i] and To[i] may not be equal. This is showed in figure 4. Both information are essential for solving AX=B to restore the order of X. Compared with the sequential algorithm (Algorithm 4 or Algorithm 5), this out-of-order version needs an additional auxiliary space of size mn × SysDim.

Algorithm 6: LU Decomposition (Out_of_Order)

for j = 0 to SysDim / BLOCKSIZE

1) for i *= 0 to BLOCKSIZE - 1*

p = find_pivot(i*,* $BLOCK_{j,j}$*)* *(find the* i'*th pivot column in* $BLOCK_{j,j}$*)*

update From[i] , To[i]

L update(p) *(out of order L update)*

r1_update(i*, p)* *(out of order trailing update)*

end for

2) reorder_columns(From, To)

end

Algorithm 6.1: Reorder_columns

$d \leftarrow SysDim$

$g \leftarrow$ *GroupIndex*

$s_0 \leftarrow$ *subindex of current thread in group g*

for i *= 0 to d*

temp_to $\leftarrow A_{i,s}^{(g)}$ *(read coalescing)*

temp_from $\leftarrow A_{i,From[s]}^{(g)}$

$A_{i,s}^{(g)} \leftarrow$ *temp_from* *(write coalescing)*

$A_{i,To[s]}^{(g)} \leftarrow$ *temp_to*

end

3.8. Out of Order Block Version

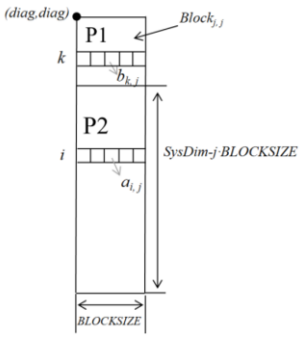

Figure 5. Trailing Update in P2

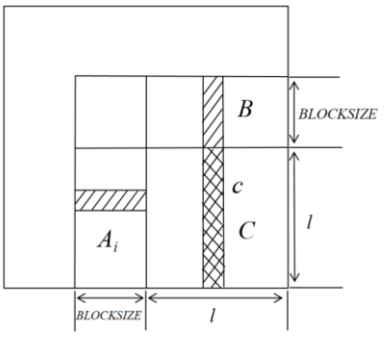

Figure 6. Trailing Update in P3

There is still another problem in Algorithm 6, i.e. the trailing update is inefficient. This is because it is carried out every time the pivot is selected, which can be optimized further. When the $i'th$ pivot column of index j has been selected, in order to determine the next adjacent $(i+1)'th$ pivot column correctly, the region P1 must be updated firstly. So the region P1 should be processed every time a pivot column is determined. For the other P2 and P3 regions, their trailing update can be postponed till after all pivot columns swap.

Algorithm 7: LU Decomposition (OUT_OF_ORDER_BLOCK)

for j = 0 to SysDim / BLOCKSIZE

1) for i = 0 to BLOCKSIZE - 1

p = find_pivot(i, $BLOCK_{j,j}$) (find the i'th pivot column in $BLOCK_{j,j}$)

update From[i] , To[i]

L update(p) (out of order update)

partial r1_update(i, p) (out of order update P1)

end

2) reorder columns(From, To)

3)r1_update_left_block(P2)

4)r1_update_block(P3) (gemm: $C = \gamma AB + \beta C$)

end

In the third step of Algorithm 7 (r1_update_left_block(P2)), for any element $a_{i,j}$ in P2 (figure 5), it is updated by:

$$a_{i,j} = a_{i,j} - \sum_{k=0}^{j-1} a_{i,k} b_{k,j}$$

Where $0 \le i \le SysDim - m \times BLOCKSIZE - 1, 0 \le j \le BLOCKSIZE - 1$.

Algorithm 7.1: R1_update_left_block(P2)

j ← subindex of current thread in group g

for i = 0 to l

for k = 0 to j – 1

wait $thread_k$ *to compute* $a_{k,j}$

$a_{i,j} = a_{i,j} - a_{i,k}b_{k,j}$

end

end

In the fourth step of Algorithm 7(r1_update_block(P3)), each thread is responsible for computing a column C, A and B are also vectors. For vector B, it is cached in the register, and for A and C, they are read directly the satisfying coalescing access condition (figure 6).

Algorithm 7.2: R1_update_block(P3)

T regData[BLOCKSIZE]

regData ← B (register cache)

for i = 0 to l

$C[i] = -A_i^{(g)}B + C[i]$

end

4. Experiments

According to the different dimensions of sub-matrix, we have implemented a total of 8 different algorithms. In this section, we will analyze the performance of each algorithm and do some comparative work. All the data precision in this section are double type. For comparison with other benchmarks, we also list the batched LU decomposition test results of cuBLAS cublasDgetrfBatched (figure 15). The test hardware configuration is: CPU:i9-10900k, GPU:RTX3080, Memory:64G.

In all figures, the abscissa corresponds to the large matrix dimension, the ordinate corresponds to the computing performance Gflops/s, and each curve corresponds to a different sub-matrix dimension.

For the MINI_NAIVE version, in order to make a cross comparison with the NORMAL version, results for dimension 16 or 32 are also added. In figure 7, most of the curves show an upward trend firstly, and then a downward trend. Going up means that global memory access latency gets partially hidden as the number of scheduled

thread blocks increases. As with the sub-matrix size increases, the horizontal data element stride among threads also increases, resulting to a non-coalesced and non-aligned memory access in each warp, and the overall performance degrade.

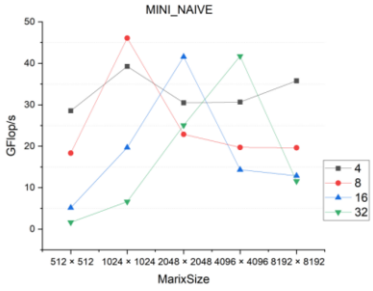

Figure 7. MINI_NAIVE Version

In the MINI_GLOBAL_COALESCING version(figure 8), data elements are rearranged so that both reads and writes are coalesced. Compared with MINI_NAIVE version, the performance is greatly improved. With M and N increase, each GPU thread processes more than one sub-matrices. This can reduce the cost of kernel function scheduling, thereby further improving the performance. But when M and N are small, this kind of loop mode will lead to fewer overall thread blocks, and the performance degrades as is shown for 512 or 1024. For sub-matrix of larger dimension like 32, there are more memory access during the LU decomposition for a single thread, its CGMA ratio(Compute to Global Memory Access Ratio) is low and the performance degrade a lot as is shown in the last sub-figure.

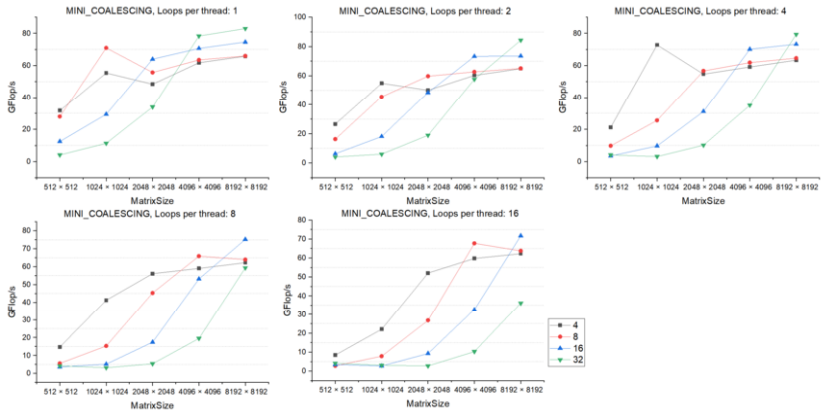

Figure 8. MINI_GLOBAL_COALESCING Version

The MINI_REGISTER version is specifically optimized for sub-matrix of size 4 or 8 and the LU decomposition is all done in the register cache(figure 9). Compared with MINI_GLOBAL_COALESCING version, the performance is improved significantly. When the sub-matrix dimension equals to 4, M × N with 512 × 512, the performance reaches about 55Gflops/s which is much higher than the cuBLAS version (figure 15).

For the largest M and N, the peaks performance for dimension 4 and 8 both exceed 80Gflops/s.

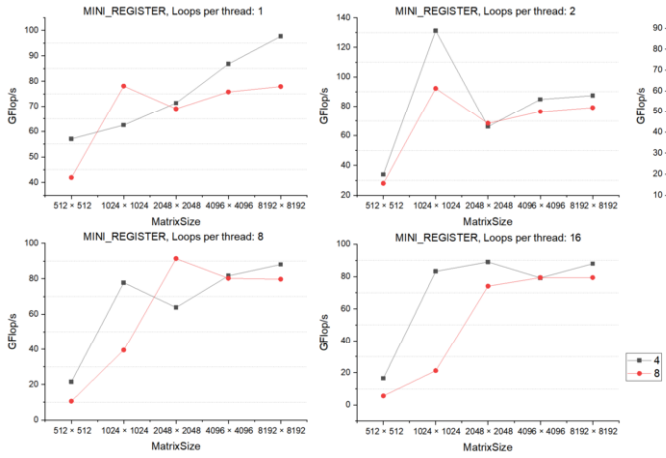

Figure 9. MINI_REGISTER Version

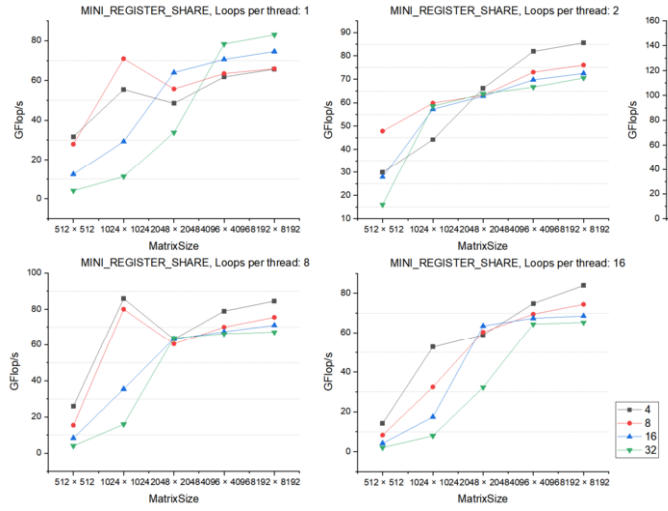

Figure 10. MINI_REGISTER_SHARE Version

In the MINI_REGISTER_SHARE version (figure 10), when the dimension of sub-matrix is 4 or 8, the overall performance is close to that of MINI_ REGISTER and the curve is basically linear in most cases. This is mainly due to the fact that each sub-matrix is processed by a group of threads, and the group size equals to the row dimension of the sub-matrix. In this version, we also add test results with dimension 16 and 32 which is mainly used for cross-comparison with the later normal algorithm. It shows that this algorithm supports dimension of size 16 very well and even achieves the best performance under the 512×512 configuration (table 3).

In the NORMAL_NAIVE version (figure 11), we adopt a fixed task allocation pattern of 16 threads group per sub-matrix. For sub-matrix dimension of 16 or 32, the algorithm achieves the best CGMA ratio under the data volume of 1024×1024 and remains close to the horizontal state with the increase of M and N. For sub-matrix with

dimension greater than 32, more global memory latency is hide by arithmetic calculations and the curve continues to rise.

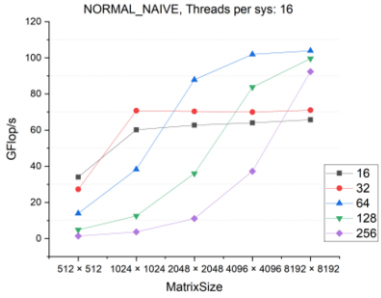

Figure 11. NORMAL_NAIVE Version

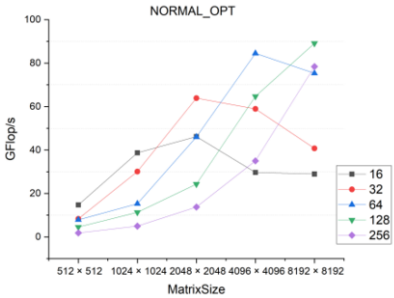

Figure 12. NORMAL_OPT Version

In the NORMAL_OPT version (figure 12), its overall performance is lower than NORMAL_NAIVE version. The reason is: in the process of LU decomposition of each sub-matrix, the find pivot, row exchange and L update steps are all conducted by one thread. The frequently access to global memory leads to a performance bottleneck. The main purpose of this version is used to compare with subsequent optimized versions to confirm the magnitude of the performance improvement.

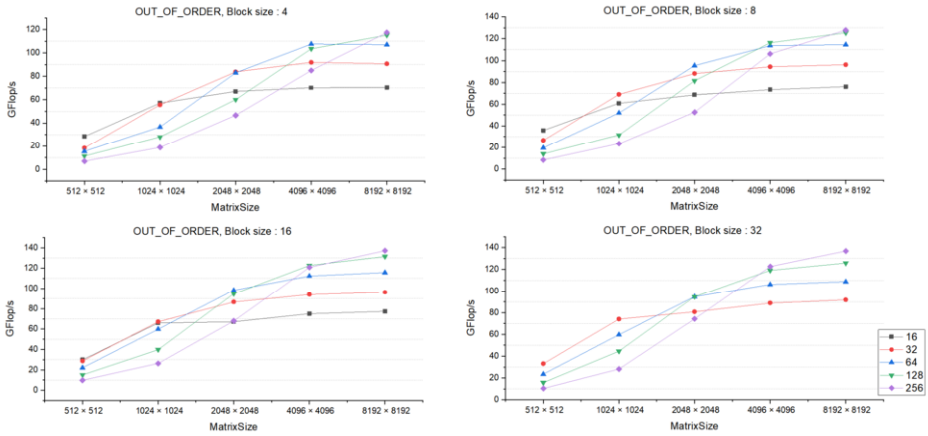

Figure 13. OUT_OF_ORDER Version

In the OUT_OF_ORDER version (figure 13), we postpone the swap of pivot columns until all the current block's pivot columns have been determined, ensuring the read and write coalescing during final pivot columns swap step. The larger the block, the more obvious the performance improved. The overall performance of this algorithm surpasses the NORMAL_OPT version and is close to the NORMAL_NAIVE version. All the curve in figure 13 is close to linear, which means this version has better stability.

In the OUT_OF_ORDER_BLOCK version (figure 14), when the sub-matrix dimension is 128 or 256 and M, N are small (512 or 1024), the performance is relatively low. This is due to the fact that the number of thread blocks is too small to overcome the scheduling cost. However, with the increase of M and N, the performance increases at a nearly linear trend and the highest peak value is close to 450Gflops/s. Compared with cuBLAS (figure 15), the speedup ratio is more than 15. Also, this version has

overwhelming superiority in terms of stability and performance when the size of the equations is larger than 16.

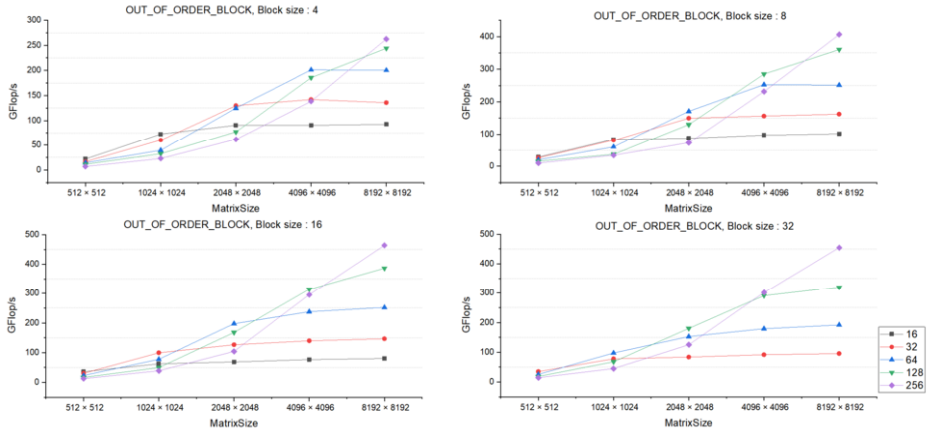

Figure 14. OUT_OF_ORDER_BLOCK Version

In figure 15, we list the test results of cuBLAS batched mode LU decomposition (cusolverDnDgetrf). As it can be seen that cuBLAS makes a special optimization for sub-matrix of size 32 which outperforms best in most cases. Even compared with this size, our OUT_OF_ORDER_BLOCK Algorithm still has advantages. When M and N are greater than 1024, the speedup ratio is nearly 3 (table 4).

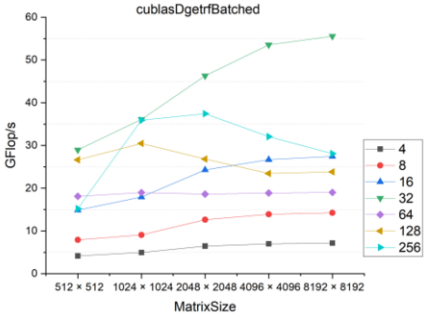

Figure 15. cublasDgetrfBatched

The following series of tables (tables 1-7) sort out the best performance of the LU algorithm under different configurations and the corresponding results of cusolverDnDgetrf under the same parameter configuration. It can be seen that except for few individual cases when M and N are small, most configurations achieve a speedup ratio greater than 3. In some cases, the speedup ratio is even greater than 10.

It also indicates from the series of tables that: for sub-matrix of dimension 4 or 8, the MINI_REGISTER version performs best, and its peak value is close to 100Gflops/s when M and N are larger than 1024; For sub-matrix dimension greater than 8, the OUT_OF_ORDER_BLOCK version shows its dominant advantages. It has the best performance in almost all configurations and the speedup ratio is very stable and grows linearly.

Table 1. Performance Comparison with cuBLAS of Size 4

Size 4	Config	Gflop/s	cuBLAS	Ratio
512*512	MINI_ REGISTER , Loops per thread: 1	57.03	4.16	13.71
1024*1024	MINI_ REGISTER_SHARE, Loops per thread: 4	143.88	4.96	29.01
2048*2048	MINI_ REGISTER, Loops per thread: 16	89.03	6.45	13.80
4096*4096	MINI_ REGISTER, Loops per thread: 1	86.8	6.98	12.44
8192*8192	MINI_ REGISTER, Loops per thread: 1	97.66	7.16	13.64

Table 2. Performance Comparison with cuBLAS of Size 8

Size 8	Config	Gflop/s	cuBLAS	Ratio
512*512	MINI_ REGISTER_SHARE, Loops per thread: 1	49.44	7.92	6.24
1024*1024	MINI_ REGISTER, Loops per thread: 2	92.24	9.07	10.17
2048*2048	MINI_ REGISTER, Loops per thread: 8	91.41	12.63	7.24
4096*4096	MINI_ REGISTER, Loops per thread: 8	80.24	13.91	5.77
8192*8192	MINI_ REGISTER, Loops per thread: 4	80.06	14.25	5.62

Table 3. Performance Comparison with cuBLAS of Size 16

Size 16	Config	Gflop/s	cuBLAS	Ratio
512*512	MINI_ REGISTER_SHARE, Loops per thread: 1	48.67	14.87	3.27
1024*1024	OUT_OF_ORDER_BLOCK, Threads per sys : 8	83.49	17.94	4.65
2048*2048	OUT_OF_ORDER_BLOCK, Threads per sys : 4	90.78	24.27	3.74
4096*4096	OUT_OF_ORDER_BLOCK, Threads per sys : 8	96.86	26.69	3.63
8192*8192	OUT_OF_ORDER_BLOCK, Threads per sys : 8	101.4	27.45	3.69

Table 4. Performance Comparison with cuBLAS of Size 32

Size 32	Config	Gflop/s	cuBLAS	Ratio
512*512	OUT_OF_ORDER_BLOCK, Threads per sys : 32	34.6	28.97	1.19
1024*1024	OUT_OF_ORDER_BLOCK, Threads per sys : 16	99.42	36.12	2.75
2048*2048	OUT_OF_ORDER_BLOCK, Threads per sys : 8	149.63	46.3	3.23
4096*4096	OUT_OF_ORDER_BLOCK, Threads per sys : 8	156.49	53.55	2.92
8192*8192	OUT_OF_ORDER_BLOCK, Threads per sys : 8	162.38	55.57	2.92

Table 5. Performance Comparison with cuBLAS of Size 64

Size 64	Config	Gflop/s	cuBLAS	Ratio
512*512	OUT_OF_ORDER_BLOCK, Threads per sys : 32	25.31	18.1	1.40
1024*1024	OUT_OF_ORDER_BLOCK, Threads per sys : 32	97.67	18.98	5.15
2048*2048	OUT_OF_ORDER_BLOCK, Threads per sys : 16	197.8	18.62	10.62
4096*4096	OUT_OF_ORDER_BLOCK, Threads per sys : 8	252.79	18.87	13.40
8192*8192	OUT_OF_ORDER_BLOCK, Threads per sys : 16	252.22	19.01	13.27

Table 6. Performance Comparison with cuBLAS of Size 128

Size 128	Config	Gflop/s	cuBLAS	Ratio
512*512	OUT_OF_ORDER_BLOCK, Threads per sys : 32	18.35	26.63	0.69
1024*1024	OUT_OF_ORDER_BLOCK, Threads per sys : 32	67.91	30.47	2.23
2048*2048	OUT_OF_ORDER_BLOCK, Threads per sys : 32	180.32	26.83	6.72
4096*4096	OUT_OF_ORDER_BLOCK, Threads per sys : 16	314.59	23.44	13.42
8192*8192	OUT_OF_ORDER_BLOCK, Threads per sys : 16	387	23.82	16.25

Table 7. Performance Comparison with cuBLAS of Size 256

Size 256	Config	Gflop/s	cuBLAS	Ratio
512*512	OUT_OF_ORDER_BLOCK, Threads per sys : 32	14.41	15.22	0.95
1024*1024	OUT_OF_ORDER_BLOCK, Threads per sys : 32	44.88	35.93	1.25
2048*2048	OUT_OF_ORDER_BLOCK, Threads per sys : 32	124.95	37.43	3.34
4096*4096	OUT_OF_ORDER_BLOCK, Threads per sys : 32	301.62	32.07	9.41
8192*8192	OUT_OF_ORDER_BLOCK, Threads per sys : 16	463.68	28.09	16.51

5. Conclusion

In this paper, we proposed *8* different batched LU decomposition algorithms for small and medium-sized matrices and made a systematic and comprehensive performance comparison among these algorithms. Finally, two outperforming algorithms are selected which have the best speedup ratio compared with cuBLAS. The mainly optimization techniques utilized in these algorithms include memory coalescing, register cache, out-of-order execution, and delayed trailing update. These techniques can also be applied to a batched version of Chomsky decomposition algorithm in approximately the same way, which will be an extension of our future work. Meanwhile, for the results in Table 3 and Table 4, it is reasonable to believe that the performance can be further improved. If the out-of-order execution and delayed trailing update technique are applied to the MINI_REGISTER_SHARE version, row swap can be avoided and the number of trailing update calls can be effectively reduced, then the better performance will be theoretically achieved. This work will also be carried out in the future.

Acknowledgments

This work is supported by the National Key RD Program of China under grant No.2018YFB1701600.

References

[1] N Galoppo, N.K Govindaraju, M Henson, and D Manocha. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In 2005 Proceedings of the ACM/IEEE Supercomputing (SC) Conference, page 3, 2005.

[2] D Yu Chenhan, Weichung Wang, and Danl Pierce. A cpu–gpu hybrid approach for the unsymmetric multifrontal method. Parallel Computing, 37(12):759–770, 2011.

[3] Thomas George, Vaibhav Saxena, Anshul Gupta, Amik Singh, and Anamitra R Choudhury. Multifrontal decomposition of sparse spd matrices on gpus. In 2011 IEEE International Parallel & Distributed Processing Symposium, pages 372–383. IEEE, 2011.
[4] Peng S, Tan S X D. GLU3. 0: Fast GPU-based parallel sparse LU decomposition for circuit simulation[J]. IEEE Design & Test, 2020, 37(3): 78-90.
[5] Zhou Tao, Man Di, Gao Wenpeng, et al. Parallel processing based on matrix LU decomposition [J]. Computer Knowledge and Technology: Academic Edition, 2016(7X): 21.
[6] Mrdakovic B L, Kostic M M, Olcan D I, et al. Acceleration of in-core LU-decomposition of dense MoM matrix by parallel usage of multiple GPUs[C]//2017 IEEE International Conference on Microwaves, Antennas, Communications and Electronic Systems (COMCAS). IEEE, 2017: 1-4.
[7] V. Volkov and J. W. Demmel,"LU, QR and Cholesky decompositions using vector capabilities of GPUs," Tech. Rep. LAPACK Working Note 2009.
[8] S. Tomov, R. Nath, and J. Dongarra,"Dense linear algebra solvers for multicore with gpu accelerators," in Proc. of the IEEE IPDPS'10, Atlanta, GA, April 19-23 2014.
[9] V. Oreste, M. Fatica, N. A. Gawande, and A. Tumeo, "Power/performance trade-offs of small batched LU based solvers on GPUs," in 19th International Conference on Parallel Processing, Euro-Par 2013, Aachen, Germany, August 26-30 2013.
[10] 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.
[11] NVIDIA. CUDA Basic Linear Algebra Subroutines (cuBLAS). (2017). http://docs.nvidia.com/cuda/cuBLAS/.
[12] Dong T, Haidar A, Luszczek P, et al. LU decomposition of small matrices: Accelerating batched DGETRF on the GPU[C]//2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS). IEEE, 2014: 157-160.
[13] Li Mengyue, Wang Ying, Ma Gang, et al. A batch LU decomposition algorithm based on graphics processor acceleration [J]. Electric Power Engineering Technology, 2019 (2): 57-63.