A CPU–GPU hybrid approach for the unsymmetric multifrontal method

Chenhan D. Yu\textsuperscript{a}, Weichung Wang\textsuperscript{a,}\textsuperscript{*}, Dan’l Pierce\textsuperscript{b}

\textsuperscript{a}Department of Mathematics, National Taiwan University, Taipei 10617, Taiwan
\textsuperscript{b}MSC. Software Corporation, Glendale, CA 91203, USA

**Abstract**

Multifrontal is an efficient direct method for solving large-scale sparse and unsymmetric linear systems. The method transforms a large sparse matrix factorization process into a sequence of factorizations involving smaller dense frontal matrices. Some of these dense operations can be accelerated by using a graphic processing unit (GPU). We analyze the unsymmetric multifrontal method from both an algorithmic and implementational perspective to see how a GPU, in particular the NVIDIA Tesla C2070, can be used to accelerate the computations. Our main accelerating strategies include (i) performing BLAS on both CPU and GPU, (ii) improving the communication efficiency between the CPU and GPU by using page-locked memory, zero-copy memory, and asynchronous memory copy, and (iii) a modified algorithm that reuses the memory between different GPU tasks and sets thresholds to determine whether certain tasks be performed on the GPU. The proposed acceleration strategies are implemented by modifying UMFPACK, which is an unsymmetric multifrontal linear system solver. Numerical results show that the CPU–GPU hybrid approach can accelerate the unsymmetric multifrontal solver, especially for computationally expensive problems.

**1. Introduction**

Solving large-scale unsymmetric and sparse linear systems $Ax = b$ is at the heart of various scientific and engineering computations. To solve these problems, direct methods relying on symbolic and numerical factorizations have been used for decades and remain popular choices. In this article, we intend to investigate how we can integrate the highly parallel computational capability of graphic processing units (GPU) with a central processing unit (CPU) system to accelerate the direct solving process for unsymmetric sparse linear systems.

Among various direct methods, we focus on the unsymmetric multifrontal method\cite{1–3} in particular. The choice of multifrontal method is mainly motivated by the following observation. A multifrontal method uses a factorization tree to transform a large sparse linear system problem into many operations involving smaller frontal dense matrices. The main computational tasks involving these dense frontal matrices are level 2 and level 3 Basic Linear Algebra Subprograms (BLAS) computations, we can accelerate the whole multifrontal process if we can accelerate the BLAS computations via a GPU.

Among various direct methods, we focus on the unsymmetric multifrontal method\cite{1–3} in particular. The choice of multifrontal method is mainly motivated by the following observation. A multifrontal method uses a factorization tree to transform a large sparse linear system problem into many operations involving smaller frontal dense matrices. The main computational tasks involving these dense frontal matrices are level 2 and level 3 Basic Linear Algebra Subprograms (BLAS) computations, we can accelerate the whole multifrontal process if we can accelerate the BLAS computations via a GPU.

A GPU contains a large number of computational cores on a single board. These cores can work simultaneously and thus justify the ideas to accelerate the BLAS and other operations in a multifrontal method via a GPU. However, it is important to realize that a GPU is not a standalone platform, but a device driven by a CPU. Consequently, the data must be transferred between the host CPU and the device GPU. The communication cost between the CPU and GPU thus plays a critical role in the overall performance.

Instead of writing a GPU-based unsymmetric multifrontal program from scratch, we modify UMFPACK\cite{3}, to implement the proposed acceleration strategies. While other unsymmetric multifrontal packages exist (like WSMP\cite{4} and MUMPS\cite{5}),
we chose UMFPACK mainly because it is efficient and well-designed, is written in C, has detailed documentation, and provides source code that is readily available.

Several attempts at parallelizing the multifrontal methods have been considered. Actually the multifrontal method can be parallelized naturally by factoring different frontal matrices on different CPU processors [6]. Furthermore, the frontal matrices can be mapped into the processors in several levels. For example, the MUMPS package employs three levels of parallelization. The first level maps each frontal matrix to a processor. The second and third levels map a frontal matrix to several processors which means each processor will do a block factorization of the corresponding frontal matrix [7]. In addition, a GPU has been used to accelerate multifrontal computations for symmetric linear systems in [8–10].

This paper is organized as follows. We briefly discuss the unsymmetric multifrontal method and the GPU architecture in Section 2. We propose the general strategies to accelerate the unsymmetric multifrontal method via a GPU in Section 3. Details of the acceleration algorithm and implementations are then discussed in Section 4. Numerical results are reported in Section 5. We provide conclusions in Section 6.

2. Background

In this section, we briefly introduce the framework of the unsymmetric multifrontal method and the NVIDIA Tesla C2070 GPU.

2.1. Multifrontal method

A typical multifrontal algorithm consists of the symbolic factorization and the numerical factorization. In the symbolic factorization stage, an elimination tree is formed. The numerical factorization includes pivoting, assembling, and updating. Then performed on each of the frontal matrices. For a frontal matrix, the following transformation is performed.

\[
\begin{bmatrix}
P & Q \\
R & C
\end{bmatrix} = \begin{bmatrix}
L_1 & 0 \\
L_2 & C - L_2 U_2
\end{bmatrix} \begin{bmatrix}
U_1 & U_2 \\
0 & I
\end{bmatrix},
\]

(1)

where block \(P\) contains all of the pivots in the current supernode, blocks \(Q\) and \(R\) contain the corresponding nonzero rows and columns of the pivots, \(P = L_1 U_1, L_2 = RU_1^{-1}\), and \(U_2 = L_1^{-1} Q\). Fig. 1(a) shows a schema of the current frontal matrix of the elimination tree structure. This frontal matrix will be transformed to the one shown in Fig. 1(b). Some variables (fnpiv, fncols, fnrows, nb, and fnc_curr) used in UMFPACK to record the dimensions are defined in Fig. 1(c). Algorithm 1 is a simplified version of the unsymmetric multifrontal method used by UMFPACK.

**Algorithm 1.** Outline of the unsymmetric multifrontal method

```
1: Initializations
2: Symbolic factorization and form the elimination tree
3: for all (frontal matrices) do
4: while (factorize the frontal) do
5: Assemble using update matrices and original matrix elements
6: Perform row pivoting
7: Factor the pivot by TRSV to get a part of \(L_1\) and \(U_1\) (BLAS2)
8: Update part of \(L_2\) by GEMV (BLAS2)
9: end while
10: Update \(U_2\) by TRSM and update \(C\) by GEMM (BLAS3)
11: end for
```

The main computational cost of the algorithm is the numerical factorization, which can be performed via BLAS functionality. Basically, two level 2 BLAS (BLAS2) and two level 3 BLAS (BLAS3) are necessary for each Frontal matrix. For the BLAS2, it includes (i) TRSV (triangular solve with single right hand side vector) in line 1 used to perform the factorization of a single
pivot to get a part of $L_1$ and $U_1$ and (ii) GEMV (general matrix-vector multiplication) in line 1 used to update $L_2$. For the BLAS3, TRSM (triangular solve with multiple right hand side vectors) and GEMM (general matrix-matrix multiplication) are used to block pending updates for $U_2$ and $C$, respectively, as shown in lines 10 of the algorithm. Note that, if $P$ has only one pivot, the computation $C = C - L_2U_2$ is actually an outer product and GER is used to update $C$.

2.2. Architecture of GPU

A GPU is designed to perform arithmetic intensive parallel computations by the multiple computational cores on-board. In this article, we consider how the NVIDIA Tesla C2070 GPU can be used to accelerate the multifrontal solver. Note that the C2070 is also codenamed Fermi and belongs to the GF100 series GPU cards. An abstract architecture schema of the C2070 GPU is illustrated in Fig. 2. The C2070 GPU contains 14 multiprocessors and each multiprocessor contains 32 thread processors (or scalar processors). The GPU card has 6 GB of global memory that is available to all of the multiprocessors. Each multiprocessor is equipped with low-latency shared memory that is available to the 32 thread processors in the same multiprocessor. In addition, each thread processor has its own registers. A read only texture memory can be used by binding part of the global memory. As the data is cached by spatial location on the texture memory, some memory can be reused, the performance of fetching global memory can be improved, and some hardware or software functionalities like interpolation are supported.

A kernel to be run on a GPU is invoked by the host CPU. A kernel consists of one-, two-, or three-dimensional (virtual) thread blocks that are distributed to the available multiprocessors. Each thread block can have no more than 1,024 threads for all dimensions in total and each thread will be assigned to a single thread processor at runtime. While the maximum available number of threads on a C2070 GPU is $1,024 \times 65,535 \times 65,535$, no more than 32 threads are executed on a multiprocessor simultaneously. Via the PCI-Express (PCI-E) interface, the data is transferred between the main memory on CPU (the host memory) and the global memory on GPU (also called the device memory). An NVIDIA development platform called Compute Unified Device Architecture (CUDA) [11] allows users to code programs by using high-level programming languages like C and FORTRAN. More hardware and programming details can be found in [12].

3. Acceleration strategies

We have shown that BLAS operations play a critical role in the multifrontal method in Section 2.1. To see how a GPU can be used to accelerate the unsymmetric multifrontal method, we first review two particular BLAS implementations based on CUDA and then consider how the communication between CPU and GPU will affect the overall performance, as the CPU–GPU hybrid environment introduces communication overhead. By using hardware features and modifying the algorithm suitably, we propose several ways to decrease the effects of the communication cost to accelerate the overall performance.

3.1. BLAS on a GPU

We focus on how the BLAS can be accelerated on a GPU by considering the following two BLAS implementations.
• CUBLAS [13] is provided by NVIDIA. It is an implementation of BLAS on top of CUDA. The CUBLAS routines can be called through the CUDA API directly, if the related matrices and vectors have been stored in the memory of the GPUs previously. CUBLAS includes the level 1, 2, and 3 BLAS routines. CUBLAS also continues to be improved upon. For example, in CUBLAS version 2.0, it takes advantage of the memory on GPU by using the concept of inverse memory hierarchy [14], which takes advantage of growing the aggregated size of on-chip memory and uses registers in an optimal way. In CUBLAS 3.2, some CUBLAS routines can be around 1.5× to 4.0× faster than CUBLAS 2.0 on a Fermi series GPU.
• The Matrix Algebra on GPU and Multicore Architectures project (MAGMA) [15] is under current development. The project focuses on the development of a dense linear algebra library in hybrid environments that include multicore CPU plus GPU systems. In version 0.2, MAGMA has provided some BLAS routines. These routines try to divide the computational tasks and then distribute the tasks to CPU and GPU suitably to accelerate the overall performance. Note that some of the MAGMA BLAS routines have been included in CUBLAS 3.2.

While the GPU can accelerate BLAS2 and BLAS3 operations, we will still need to keep some BLAS operations performed on the CPU. Such a choice actually involves another important consideration for using a GPU, namely the communication costs between CPU and GPU. We discuss this issue in the next section.

3.2. Communication between CPU and GPU

The performance for using CUBLAS or MAGMA BLAS actually depends on (i) the efficiency of BLAS of CPU and GPU and (ii) the trade-off between the acceleration due to GPU and the cost spent on data transfer between CPU and GPU. For example, \texttt{DGEMM} is used to compute the Schur complement \( C = C - L_2 U_2 \). The matrices \( L_2 \) and \( U_2 \) must be large enough, so the total time for a GPU acceleration of the BLAS, which includes kernel invoking, data communications, and arithmetic computations, will be higher than the time for using \texttt{DGEMM} on the CPU. In other words, replacing all the BLAS sub-routines with the counterparts within CUBLAS may actually degrade the overall performance in some cases.

Here, we suggest five strategies to decrease the effects of communication costs between CPU and GPU and to better use the computation capability provided by the GPU. The first three approaches, i.e. page-locked memory, zero-copy memory, and asynchronous memory copy are closely related to the target GPU hardware features. The last two approaches, i.e. memory reuse and threshold setting, are related to efficient algorithm design for any CPU–GPU hybrid environment.

**Page-locked memory.** We can accelerate data transfer between the host memory and the PCI-E bus by using page-locked memory. As the physical location of data stored in the pageable memory is unknown, the GPU cannot apply Direct Memory Access (DMA) and therefore the data transfer is slowed down. To circumvent this overhead, CUDA allows users to allocate page-locked memory on the host. In this case, the physical location of data is known and the hardware can apply DMA directly out of the physical random access memory to accelerate the data access. Using page-locked memory can get about 1.5× speedup on memory transfer in our experience. However, it is not recommended to allocate too much of page-locked memory, as it over-constrains the operating system virtual memory system and slows down the performance.

**Zero-copy memory.** Starting from CUDA version 3.0, the zero-copy memory mapping function is supported. The idea is to map the physical memory on the host into the virtual address space on the device, so the memory copy between the host and device can be omitted virtually. Performing a load or store instruction on one of these virtual address on the device actually initiates a PCI-E transaction that loads or stores the data directly in the host memory. While zero-copy memory can be efficiently accessed in some cases, it has its limit. It is not suitable to those cases where the data is being reused, as the data is still transferred every time when it is needed regardless if a change has occurred. Therefore, we implement zero-copy only if the task is large enough and has less memory reuse.

**Asynchronous memory copy.** We can hide the time for a memory transfer by an asynchronous memory copy. By using different streams, CUDA allows multiple processes to follow these streams in the following manner. All jobs that are stacked in the same pipeline stream will be executed in order. But work in different streams can be executed concurrently. The Tesla C2070 GPU supports such asynchronous memory copy by allowing concurrent data transfer and kernel launching, that is, the GPU can copy memory and execute a kernel at the same time.

**Memory reusing.** A way to reduce the amount of memory transfer is to reuse the common memory blocks in different tasks. To do so, we need to identify the tasks that use the same memory blocks and perform such tasks on the GPU. So the device memory can be reused in more than one task, rather than being transferred back and forth. However, we need to be careful in balancing the trade-off between saving on communication costs and the cost of computation performed on a GPU. Some tasks that are sequential, cause thread divergence, or have inefficient memory accesses and may be slow on a GPU.
Threshold setting. Another way to avoid inefficiency due to data communications is to set a threshold to a certain task to decide whether the task shall be performed on the CPU or the GPU. The thresholds are chosen so that the saving in computations is larger than the total time used to launch the kernel, to allocate memory, and to copy memory. For example, we can set a threshold on the size of matrices to be multiplied by GEMM, so that only the matrices that are large enough would be computed on the GPU. It is important that different thresholds must be set for different tasks. The values of the thresholds are determined by numerical experiments and are machine dependent.

4. Algorithms and implementation details

We have proposed general principles for accelerating the multifrontal method via the CPU and GPU in the previous section. Based on these principles, we propose Algorithm 2. The algorithm describes how the numerical factorization can be performed on the CPU and GPU cooperatively. It shows how the memory should be allocated in the CPU and GPU (lines 1–2). It indicates the tasks that should be done in CPU or GPU. Tasks on lines 8, 9, and 12 can be performed on CPU or GPU. We discuss this part in detail in Sections 4.1, with an assistance of the execution time-line corresponding to the algorithm that is shown in Table 1. In line 2, two criteria that determine whether the column pivoting set is large enough are (i) \( \text{fn piv} \) and \( \text{fn rows} \), as defined in Fig. 1(c), are large and (ii) \( \text{fn piv} = \text{nb} \). Note that the framework of Algorithm 2 is similar to Algorithm 1, however, column pivoting is considered in Algorithm 2.

Algorithm 2. CPU–GPU hybrid numerical factorization

1: Allocate page-locked memory on the host CPU: \( LU, L_2, U_2, C \)
2: Allocate device memory on the device GPU: \( LU_d, L_d, U_d, C_d \)
3: for (frontal matrices) do
4: while (factorize the frontal) do
5: Perform column pivoting on CPU
6: Assemble the frontal matrix due to the column pivoting on CPU
7: Perform row pivoting on CPU
8: Factor the pivot by TRSV to get a part of \( L_1 \) and \( U_1 \) (BLAS2)
9: Update part of \( L_2 \) by GEMV (BLAS2)
10: if (column pivoting set is large enough) then
11: Assemble the matrix for updating \( C \) on GPU
12: Update \( U_2 \) by TRSM and update \( C \) by GEMM (BLAS3)
13: end if
14: end while
15: end for
16: Free page-locked and device memory

In Sections 4.1 and 4.2, some implementation details on the modification of UMFPACK are presented as concrete examples of the acceleration strategies. Some of the implementation details are closely related to UMFPACK and may not be suitable to other multifrontal packages. However, we illustrate these details to elaborate on the proposed strategies and hopefully to inspire possible implementations for other unsymmetric multifrontal packages.

4.1. Acceleration implementation details

We discuss how we modify UMFPACK by following the acceleration strategies presented in Section 3 item by item.

Table 1

<table>
<thead>
<tr>
<th>(a) Time line for lines 2–2 in Algorithm 2</th>
<th>Works of lines 2–2</th>
<th>Copy C to GPU if needed (by CUDA stream) or TRSV and GEMV on GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Perform column pivoting</td>
<td></td>
</tr>
<tr>
<td>TH</td>
<td>TRSV and GEMV on CPU</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b) Time line for lines 2–2 in Algorithm 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>TH</td>
</tr>
<tr>
<td>Reuse ( U_2 ) with TRSM and perform GEMM on CPU</td>
</tr>
<tr>
<td>AC</td>
</tr>
<tr>
<td>Store-LU</td>
</tr>
</tbody>
</table>

Asynchronous memory copy sections are marked by AC. Some tasks use the threshold to determine whether they should be executed on CPU or GPU and these tasks are marked by TH.
Page-locked and zero-copy memory. We allocate the memory for $L$, $L^T$, $U$, and $C$ on the host as the page-locked memory. In addition, some temporary space that is used in the numerical factorization is also allocated as page-locked memory. All of these memory blocks are then accessed by the zero-copy technique.

Asynchronous memory copy. The tasks that perform asynchronous memory copies are marked as “AC” in Table 1. Here, we discuss two examples where asynchronous memory copies may occur. First, as shown in Table 1(a), after the column pivoting is done, we know the size of the column pivoting set. If the set is large enough so that $C$ will be updated in the GPU later (line 2 in Algorithm 2), we can copy $C$ to the GPU while other works (lines 2–2) are done on the CPU.

Second, as shown in Table 1(b), after the BLAS3 update is completed (in line 2 of Algorithm 2), we obtain part of the final factored matrices $L_1$, $U_1$, $L_2$, and $U_2$. These partial portions are then stored in arrays on the host memory with a sparse format. Such a process is named as “Store-LU” and it takes some time to finish this task due to the random memory accesses. Consequently, the process Store-LU offers a chance to hide the memory transfer time. We can let the tasks Store-LU and transferring block $C$ back to the CPU overlap each other and thus save time due to asynchronous memory copy.

Memory reusing. The matrix $U_2$ is reused in Algorithm 2 as indicated in Table 1 with the notation “Reuse $U_2$.” $U_2$ is reused because it is first updated by $\text{TRSM} \left( U_2 = L_1^{-1}Q \right)$ and then used in the computation $C = C - U_2U_2^T$. The matrix $C$ is then reused in Algorithm 2 as shown in Table 1 with the notation $\text{Reuse } C$. $C$ is reused because it is first updated by $\text{TRSM} \left( C = C - L_2U_2^T \right)$ and then used in the computation $C = C - L_2U_2^T$.

Threshold setting. We set a threshold for the BLAS2 tasks and another threshold for the BLAS3 updates to determine whether the BLAS tasks should be performed on the CPU or the GPU. These thresholds are labelled by “TH” in Table 1.

We conclude this section by making the following two remarks. First, while performing asynchronous memory copies in the numerical updating and numerical factoring stages, a computational routine must run long enough to hide the memory transfer time. Second, UMFPACK implements a relaxed amalgamation heuristic that attempts to balance the trade-off between forming larger BLAS3 update matrices by introducing extra zeros and additional arithmetic operations due to the extra zeros. Such amalgamation will affect whether the if-block in line 2 in Algorithm 2 is performed.

4.2. Assembling

Assembling integrates several children matrices into their common and unique parent matrix. Each of the child matrices keeps its own index vector to indicate where the elements should be added in the parent matrix. Assembling is a memory bound task and most of its work involves memory accesses, while some additions to matrix elements are necessary. The elements in the children matrices are taken once and therefore there is no data sharing nor reusing occurs on registers or shared memory. In addition, accesses of both children and parent matrices are likely uncoalescing, so the memory accesses are irregular.

While the aforementioned lack of reuse and irregular access patterns affect the performance of GPU (and other cache systems), several data transformation and memory usage techniques can be applied to accelerate the assembly on the GPU. First, we can bind the children matrices with the read-only texture memory to use the cache on the texture memory for faster accesses. Note that the parent matrix is stored in the global memory rather than in the texture memory, as we need to update the parent matrix and thus need to write the memory. Second, we can use the zero-copy memory. As the PCI-E Generation 2 interface can perform forward and backward memory transfers simultaneously, we may use this feature to copy children matrices from and to the host simultaneously. Third, Since only a part of the matrix $C$ involves the assembling and thus needs to be transferred, we can use the zero-copy technique to access this part of $C$ rather than transferring the whole $C$.

5. Numerical experiments

We modified UMFPACK to implement the proposed schemes aiming at accelerating the unsymmetric multifrontal solver by using both the CPU and GPU. The codes are used to measure the performance of the proposed algorithms. All the codes are written in C and FORTRAN. UMFPACK is compiled by Intel ICC and IFORT with optimization flag -O3. The modified codes are compiled by ICC, IFORT, and NVIDIA NVCC with flags -O3. The following packages are used: UMFPACK 5.5.0 (including UFConfig 3.5.0, AMD 2.2.1, CAMD 2.2.1, COLAMD 2.7.2, CCOLAMD 2.7.2, and CHOLMOD 1.7.1), MKL 10.3, and CUBLAS 3.2. The host is equipped with an Intel Xeon E5620 CPU at 2.4 GHz and 24 GB main memory. The device is a NVIDIA Tesla C2070 GPU with 6.0 GB device memory based on the CUDA architecture. A PCI-E Generation 2 (16X) card is used to connect the host and the device. A set of test problems is taken from [2]. The statistics of these problems are presented in Table 2.
We first present a breakdown analysis of one test problem to see how the acceleration schemes improve the performance times in each of the main tasks in Section 5.1. We demonstrate the acceleration results regarding assembling in Section 5.2. We discuss how we tune the parameters in Section 5.3. Finally, we make a comparison of all test problems to see how the GPU-accelerated solver performs generally in Section 5.4.

5.1. Breakdown analysis

The breakdown analysis of Problem OHNE2 is shown in Table 3 and we can see the main time savings is due to the BLAS3 updates. As shown in the table, the BLAS3 updates take most of time (76.58%) in UMFPACK, which also justifies why we focused on the acceleration of the tasks regarding BLAS3. CUBLAS routines can gain almost 23.5 times speedup with respect to MKL BLAS routines. However, the overall speedup is around 2.80X (no asynchronous memory copy) or 3.07X (with asynchronous memory copy), because there is extra work associated with memory that takes 25.27% of all computational cost (or 14.02 seconds) in the CPU–GPU codes. Here, the work associated with memory includes memory allocations and data transfers. This memory work is due to the pivoting schemes targeted for the unsymmetric linear systems. It is worth mentioning that such memory work has a good potential to be avoided for symmetric linear systems, as the pivoting orders can be determined in the symbolic factorization phase. The table also shows that the asynchronous memory copy does help

Note that all the performance times are reported in seconds. Double precision arithmetic is used in all numerical experiments. The results of accuracy are not reported as the solvers produce similar results up to relative machine precision. Results of FLOPS counting are also ignored as they are almost identical for both of the solvers.
by reducing the memory work from 14.42 to 9.40 seconds. The 5.02 seconds saving comes from the overlap of the AC tasks in Table 1.

Some tasks are not significantly accelerated in this problem. Little matrices are smaller than the threshold for GEMV and therefore most of the GEMV operations remain on CPU. Consequently, the improvement is invisible. Such a situation may be changed if the size of the matrices becomes larger or quicker BLAS routines on the GPU (or slower CPU BLAS) are used. The assembling is done on the CPU, as the children matrices do not exceed the threshold (around $1,200 \times 1,200$) in order for the assembly to be performed on the GPU.

Finally, the symbolic factorization, solve of $Ax = b$, store-LU, and some other miscellaneous works are done on the CPU and thus no further savings are gained.

5.2. Acceleration in assembling

We have proposed several acceleration schemes regarding assembling in Section 4.2, including the use of texture memory, zero-copy, and two-dimensional thread block. In addition, it seems we can reuse the page-locked memory matrix $C$ in assembling. To do so, we need to combine the memory work of assemble and BLAS3 update (lines 2 and 2 of Algorithm 2), so that we only need to copy the matrix $C$ forward and backward once. In this case, the threshold to perform assembling on the GPU becomes smaller. Actually, this threshold will be smaller than the one in zero-copy.

Numerical results regarding these schemes are shown in Table 4 for different matrix sizes. It is clear that the related matrices must be large enough so that assembling can be faster on the GPU. The table also shows that to reuse $C$ is faster than to use the zero-copy technique.

However, the reuse scheme is not recommended in the modification of UMFPACK, although it always leads to faster performance than zero-copy does in assembling. The reason is as follows. After assembling, another task named “column scaling” (not shown in Algorithm 2 for simplicity) is performed. The column scaling is relatively expensive on the GPU, as it updates $C$, performs a one dimensional add, and moves the memory of $L_2$, $U_2$, and $C$ around. If we port column scaling to the GPU, the time for the memory copy and kernel launch overhead will dominate the whole time cost and thus result in slower performance compared with doing it on the CPU. Consequently, we use the zero-copy method to do the assembling while modifying UMFPACK. Then the CPU does the column scaling and then copy $C$ to the device for the BLAS3 updating.

5.3. Parameter tuning

As discussed in Section 3.2, we can set thresholds to determine whether we want to perform BLAS3 updates on the GPU. In our implementation, the thresholds for performing these BLAS3 updates on the GPU (line 2, Algorithm 2) are determined by the parameter $fnrows$, $fn piv$, and $nb$.

As shown in Table 5, various combinations of $fnrows$ and $fn piv$ are used to solve Problem ARTIF-50. The total time for solving the problem by using CUBLAS (DTRSM and DGEMM) is presented in the table. Among the parameters $fnrows$ and $fn piv$ that are tested, the minimal timing (17.44 seconds) occurs when $fn piv = 16$ and $fnrows = 256$. We thus use $fn piv \geq 16$

<table>
<thead>
<tr>
<th>Table 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance of different acceleration techniques for assembling. In the column of “no zero-copy” timing, two acceleration schemes, texture memory and two-dimensional thread block, are used. In addition, the time used in forward copy, backward copy, and GPU kernel are listed within the parentheses.</td>
</tr>
<tr>
<td>Update mtx size</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>$63 \times 56$</td>
</tr>
<tr>
<td>$127 \times 127$</td>
</tr>
<tr>
<td>$220 \times 216$</td>
</tr>
<tr>
<td>$503 \times 638$</td>
</tr>
<tr>
<td>$835 \times 881$</td>
</tr>
<tr>
<td>$1,046 \times 1,103$</td>
</tr>
<tr>
<td>$1,684 \times 1,707$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threshold choosing for the BLAS3 updates.</td>
</tr>
<tr>
<td>$fnrows$</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>$128$</td>
</tr>
<tr>
<td>$1,024$</td>
</tr>
<tr>
<td>$512$</td>
</tr>
<tr>
<td>$256$</td>
</tr>
<tr>
<td>$128$</td>
</tr>
<tr>
<td>$64$</td>
</tr>
<tr>
<td>$32$</td>
</tr>
</tbody>
</table>
and \( \text{fnrows} \geq 256 \) as the thresholds to perform BLAS3 updates on the GPU. Note that the total timing results reported in the table have no big difference. It is because we have already done the memory hiding while transferring block \( C \), which occupies the largest part of the memory.

The most important user tunable parameter in UMFPACK related to our implementation is the dense BLAS block size \( \text{nb} \). The parameter decides the limit of the pivots that current workspace can afford. UMFPACK will keep accumulating candidate pivots until \( \text{nb} \) pivots are found and then perform the BLAS3 update (line 2 in Algorithm 2). The default value of \( \text{nb} \) in UMFPACK is 32. However, 32 is too small for the GPU to work more effectively than the CPU. We tune the upper bound of \( \text{nb} \) to 512 and do several tests of different \( \text{nb} \)'s. Table 6 shows that \( \text{nb} = 256 \) is the best value. If we choose a larger \( \text{nb} \), we need to allocate bigger memory for each of the single frontal matrices.

### 5.4. Overall comparisons

Table 7 shows overall timing comparisons of the test problems. These problems have been solved on the CPU by using UMFPACK with MKL BLAS and on the CPU and GPU by using the modified codes with MAGMA BLAS. Note that GotoBLAS2 [16] (results not shown here) and MKL BLAS have comparable speed. Speedups of total costs are also presented.
Roughly speaking, our GPU accelerated codes gain greater speedups for those problems that are more computationally demanding. While bigger and higher density matrices tend to result in longer execution times, it is actually the patterns of these coefficient matrices that affect the performance times significantly. For example, Problem AV41092 is smaller than TWOTONE; however, it takes more than six times of execution time to solve AV41092. A1To further demonstrate the effect of the coefficient matrix pattern, we solve a type of artificial problems. We use the standard finite-differences to discretize a three-dimensional Poisson problem with the Dirichlet boundary condition and random right hand side. The resulting symmetric coefficient matrix is then perturbed by adding a random nonzero off-diagonal vector to break the symmetry. Such kind of problems have higher nonzero density in the L and U factors of the LU decomposition. As shown in Table 7, we can gain higher speedups in these problems, especially for the larger ones. The dimensions of these test problems range from 125,000 (≈50³) to 314,432 (≈68³).

Among the total timing cost, the portion spent in the BLAS3 updates, which is closely related to the pattern of the matrices, plays a key role. Problems taking longer time in BLAS3 updates can benefit more from using a GPU. It is also true that a longer execution time in BLAS3 implies larger memory would be necessary to store the corresponding dense frontal matrices. The solvable problems are thus basically bounded by the device memory. Another way to run a bigger problem is to strip a frontal matrix and then to factor and update it in panels. The size of the frontal matrix that one can factor is limited by the length of the corresponding columns in a frontal matrix. Such scheme, however, may also introduce more data communication between the host and device. Consequently, the overall speedup depends on the tradeoff between the communication cost and time saving on GPU.

6. Discussions and conclusion

Aiming at solving unsymmetric linear systems by using multifrontal methods, we have studied how a CPU–GPU hybrid environment can accelerate the computations. The main motivation of the study is based on the fact that a multifrontal method transforms the factorization of a large and sparse linear system into a sequence of dense matrix operations, which has the potential to be performed on a GPU efficiently. We have analyzed the multifrontal method to find out how a GPU may assist on the acceleration by performing the BLAS computations. While the communication between CPU and GPU is a critical factor affecting the overall performance, we have suggested several strategies to increase the communication efficiency. These proposed methods include page-locked memory, zero-copy memory, asynchronous memory copy, memory reuse, and threshold setting. To find out how the proposed methods perform practically, we have modified the UMFPACK package to implement the ideas on a CPU–GPU hybrid system.

In our modification of UMFPACK, we have focused on the numerical factorization, which consumes most of the computational time in the whole process. We have left the symbolic factorization and the solve phase without change. In the numerical factorization, we have discussed how we accelerate the BLAS3 updates, how we deal with pivoting, and how we handle assembling the frontal matrices. In particular, BLAS3 updates are performed on the GPU if the sizes of the corresponding matrices are larger than the thresholds. Second, BLAS2 operations are necessary. Though the GPU can perform these BLAS2 operations in a highly parallel manner, the CPU needs less time overall. It is because the matrix sizes in the therein are relatively small and these BLAS2 routines are called frequently, that the savings in BLAS2 thus cannot cover the data transfers between the CPU and GPU and the overhead required to invoke the functions. Third, assembling may involve larger sizes of matrices to be added, but the accesses to memory are indirect and thus may be inefficient. Nevertheless, we port the assembling to the GPU to keep the frontal matrices on the GPU until they have been factored. The saving of assembling itself may not be significant, the advantage is that some of the memory copies can be avoided and the memory can be reused in other routines.

We have also reported numerical results including (i) a breakdown analysis that analyzes the acceleration performance of the main components in the multifrontal method, (ii) an efficiency analysis of various techniques to accelerate the assembling, (iii) parameter tuning studies for the BLAS3 update thresholds and a UMFPACK tuning parameter, and (iv) overall performance comparisons for a set of testing problems in timing. In short, the performance of multifrontal solvers is an interplay of size of the host memory, the size of device memory, and the data transfer efficiency between the CPU and GPU, the BLAS performances on CPU and GPU, the coefficient matrix size, and the matrix patterns.

6.1. Other potential improvement strategies

We have discussed how we accelerate the multifrontal method by modifying UMFPACK. It is possible to gain further performance improvement by redesigning the multifrontal algorithm so that a different pivoting strategy and block version of the BLAS are used. The main idea is to perform more tasks on the GPU so that we can have faster computations and further reduce data transfers. Such an idea leads to the blockwise Algorithm 3.

One way in which Algorithm 3 is different from Algorithm 2 is in the pivoting. UMFPACK uses a column elimination tree and decides the pivoting ordering of each frontal matrix at run-time. That is, UMFPACK finds pivots one-by-one to maintain numerical stability, to obtain minimum non-zero fill-ins, and to perform amalgamation. However, such run-time pivoting does not favor the GPU architecture due to excessive data transfers.
To eliminate the data transfer between the host and device, a better parallel factor algorithm for GPU would be a supernode left-looking factorization with no partial pivoting. Another possibility, as shown in Algorithm 3, is to factor \( LU \) blockwise, rather than pivoting one after another. That is, \( L_2 \) and \( U_2 \) are updated by using the BLAS3 routine \( \text{TRSM} \) rather than the BLAS2 routine \( \text{GEMV} \). Similarly, \( C \) is updated by using \( \text{GEMM} \). In addition, if each frontal matrix is large enough, the GPU can factorize it efficiently. We can perform a pre-ordering of the candidate pivots. For example picking a block of 32 or 64 pivots using roughly the same selection conditions and then factoring these pivot columns on the CPU. This would transform the update of \( L_2 \) to BLAS3 routines which is better for the GPU. However, this approach may add extra computations due to the increase in zero elements. More importantly, the approach may lower the numerical accuracy due to the blockwise pivoting. Few steps of iterative refinements, which is probably not easy to port on the GPU efficiently, might be required.

Note that techniques to reduce the memory copy time such as asynchronous memory copy or zero-copy can be applied in Algorithm 3. We also need an efficient way to transfer data to the device for the BLAS3 updates and in between the factorization and BLAS3 update, though it is better if we can reuse \( LU \) and \( L_2 \) blocks. For assembling, we suggest reusing \( C \) and pre-loading children matrices and the corresponding index arrays.

**Algorithm 3.** Blockwise numerical factorization

1. Allocate Page-locked Memory on CPU: \( LU, L, U, C \)
2. Allocate Device Memory on GPU: \( LUd, Ld, Ud, Cd \)
3. for each frontal do
   4. if the frontal size > threshold then
      5. Assemble pivot column and column row into frontal matrix on GPU
      6. Assemble update matrices into frontal matrix on GPU
      7. Factor \( LU \) block on GPU
      8. Update \( L_2 \) block by \( \text{TRSM} \) on GPU
      9. Update \( U_2 \) block by \( \text{TRSM} \) on GPU
     10. Update \( C \) block by \( \text{GEMM} \) on GPU
     11. Send memory back to CPU
     12. Store \( LU, L, U \) blocks, put update matrices on stack
   else
      14. Do all factorization steps on CPU
   end if
4. end for

Finally, it is possible to implement the multifrontal method on a multi-GPU system using POSIX threads, if the column elimination tree is used. We describe the idea by taking the factorization order on UMFPACK as an example. We can reorder the nodes on the chains by setting the last node of each chain as the first node of its parent’s first node. Consequently, we can use different threads to factor different chains of the frontal matrix as each chain is independent. We need to lock the parent chain until all of its children nodes have been factored.

### 6.2. Acceleration in symmetric linear systems

While we are targeting unsymmetric linear systems in this article, the proposed techniques can be applied to symmetric linear systems solved by multifrontal methods [17,18]. Even better, in symmetric problems, there is a good chance to gain further speedups as explained below. First, take the non-pivoting version BCSLIB-EXT [19] as an example, it is easier to implement memory reuse. Further, as it factors the pivot by a panel based \( LL^T \) factorization, we can use the routine \( \text{potrf} \) that is included in the MAGMA version of LAPACK. The CPU–GPU MAGMA version \( \text{potrf} \) can be 20 to 25 times faster than the CPU LAPACK version [15]. Second, as mentioned in Section 6.1, we can use a block version of pivoting to avoid some function calls and reuse more memory. Consequently the resulting memory works can be further reduced and the overall acceleration can be higher. Third, the TAUCS Sparse Linear Package is a symmetric positive definite multifrontal solver, which uses panel factorizations and panel updates. The structure of TAUCS symmetric multifrontal algorithm is close to the Algorithm 3, which uses \( \text{potrf} \) routines and has higher memory reuse. Consequently, TAUCS has good potential to be accelerated by GPU.

**Acknowledgments**

The authors are grateful to the referees for their valuable and helpful comments and suggestions that helped to improve the paper greatly. The authors are grateful to the Yukai Hung for many helpful discussions. This work is partially supported by the National Science Council, Taida Institute for Mathematical Sciences, and National Center for Theoretical Sciences (Taipei Office) in Taiwan.
References


