End-to-End LU Factorization of Large Matrices on GPUs

Yang Xia
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH, USA
xia.425@osu.edu

Peng Jiang
Department of Computer Science
University of Iowa
Iowa City , Iowa, USA
peng-jiang@uiowa.edu

Gagan Agrawal
School of Computer and Cyber Sciences
Augusta University
Augusta , GA, USA
gagrawal@augusta.edu

Rajiv Ramnath
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH, USA
ramnath@cse.ohio-state.edu

Abstract

LU factorization for sparse matrices is an important computing step for many engineering and scientific problems such as circuit simulation. There have been many efforts toward parallelizing and scaling this algorithm, which include the recent efforts targeting the GPUs. However, it is still challenging to deploy a complete sparse LU factorization workflow on a GPU due to high memory requirements and data dependencies. In this paper, we propose the first complete GPU solution for sparse LU factorization. To achieve this goal, we propose an out-of-core implementation of the symbolic execution phase, thus removing the bottleneck due to large intermediate data structures. Next, we propose a dynamic parallelism implementation of Kahn's algorithm for topological sort on the GPUs. Finally, for the numeric factorization phase, we increase the parallelism degree by removing the memory limits for large matrices as compared to the existing implementation approaches. Experimental results show that compared with an implementation modified from GLU 3.0, our out-of-core version achieves speedups of 1.13-32.65X. Further, our out-of-core implementation achieves a speedup of 1.2-2.2 over an optimized unified memory implementation on the GPU. Finally, we show that the optimizations we introduce for numeric factorization turn out to be effective.

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

ACM ISBN 979-8-4007-0015-6/23/02…$15.00
https://doi.org/10.1145/3572848.3577486

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**; **Linear algebra algorithms**.

Keywords: GPU acceleration, LU factorization, Memory limits

1 Introduction

A wide range of engineering and scientific computing applications require a solution of the large linear algebraic system of equations, i.e. $A \times x = b$. The direct method to solve this problem involves transforming the matrix A into two matrices: the *lower triangular* matrix L and *upper triangular* matrix U such that $A = L \times U$. This transformation is done with a method that is commonly referred to as LU factorization. After invoking this procedure, solution x can be easily obtained by solving equations involving these two triangular matrices, which are computationally much easier. For many applications, (for example, circuit simulation [16, 24, 30]), matrix A can be very large and sparse. As a result, LU factorization of large sparse matrices becomes a critical kernel for these applications.

Sparse LU factorization generally introduces new *fill-ins* (i.e. non-zero elements in L or U that were not non-zeroes in A). The positions of these new fill-ins are unknown before the runtime. As a result, LU factorization involves a *symbolic factorization* phase where the numbers (and likely positions) of these fill-ins are determined. This step is followed by a *numeric factorization* phase where the actual non-zero values are now computed.

Due to the importance of LU factorization, research on accelerating sparse LU factorization has been ongoing for decades [3–5, 8–11, 15, 19, 27, 32]. As the High Performance Computing (HPC) arena is getting dominated by heterogeneous and accelerator-based computing, several recent research efforts have specifically considered sparse LU factorization on GPUs [6, 13, 29]. However, each of these efforts has accelerated only a part of the computation using the GPUs – specifically either only numerical factorization [19, 28, 32, 36]

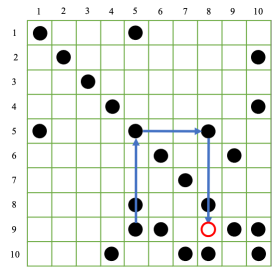
or only symbolic factorization (and further limited to counting the number of non-zeroes and not their locations) [11]. As a result, these works are not able to take full advantage of the parallelism from GPUs.

In this paper, we deliver the first efficient implementation that conducts all the steps of a sparse LU solver on a GPU, while also working with large matrices where intermediate memory requirements may exceed the available memory on a GPU. To achieve this goal, we address three major issues. The first issue is the large memory usage during the symbolic factorization step. To scale the implementation to large matrices, we propose an out-of-core GPU implementation that performs symbolic factorization iteratively.

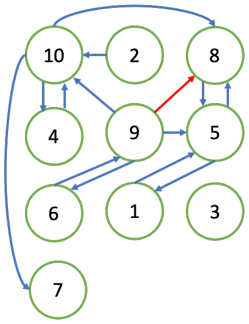
The second issue is the parallelization of the *scheduling* procedure. As a background, the implementation of numeric factorization is column by column and it is required to detect the dependencies among the columns and determine their order for numeric factorization. Previous works either use an *elimination tree*[10, 38] or a *levelization* algorithm [32, 33], but deploy the procedure on the CPU. In this work, we make the observation that this scheduling step is essentially a topological sort and propose an efficient GPU implementation with the *dynamic parallelism* feature of Kahn's algorithm [20].

Finally, previous GPU implementations [19, 32] use the dense data format for the matrix to enable fast data access during the numeric factorization step. In such a case, each column requires O(n) memory space and the number of columns that can be stored on the GPU (and thus can be processed in parallel) gets limited. This can become a significant challenge while dealing with large matrices (i.e. as n increases). To solve this issue, we propose to switch to a sparse data format when the matrix size increases. Although the data access (i.e. searching for a row id for a given column id) on sparse data formats would be less efficient, we show that we can still achieve better performance with a binary search-based access algorithm because the limit on the number of parallel columns is removed.

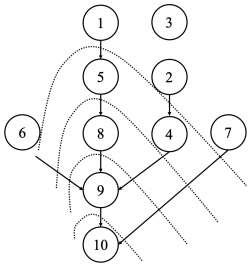
We evaluate our implementation using a set of matrices from SuiteSparse matrix collection [7]. We summarize our observations as follows. First, our out-of-core GPU symbolic factorization can support large matrices and delivers speedups in the range from 1.13 to 32.65 over an efficient CPU implementation from a recent publication (i.e. GLU 3.0[32]). Second, compared with an optimized *unified memory* solution for symbolic factorization, our out-of-core GPU solution achieves speedups in the range of 1.2-2.2 because it can access GPU device memory more efficiently. Then, we demonstrate that our optimization on numeric factorization further improves the performance by up to 3.33 times for large matrices, which is due to memory efficiency that leads to increased parallelism.



(a) An example matrix A



(b) Graph representation of matrix A



(c) Dependency graph of matrix A

level	Column Ids
0	1 2 3 6 7
1	4 5
2	8
3	9
4	10

(d) Column ids for different levels

Figure 1. Example matrix. (a) Symbolic factorization on sparse matrix A with row 9 under analysis, the red circle represents the new fill-in (b) The graph representation of matrix A, the read line presents the new fill-in (c) The dependency graph for matrix A, the dotted lines separate columns from different levels (d) A table to show column ids for different levels.

2 Background

The LU factorization of an $n \times n$ matrix, A, has the form $A = L \times U$, where L is a lower triangular matrix and U is an upper triangular matrix. LU factorization implementation generally involves major steps of pre-processing, symbolic factorization, and numerical factorization. During the pre-processing procedure, row and column permutations are performed in order to improve numerical stability and reduce the number of fill-ins in the L and U matrices [9, 10, 27]. Because our implementation primarily concerns numerical and symbolic factorization stages, we review them here. Specifically, we first introduce symbolic factorization in detail and then give an overview of a GPU implementation of numeric factorization: GLU3.0[19, 32]. Finally, we briefly explain the motivation for our work.

2.1 Symbolic Factorization and GPU Implementation

The LU factorization involves a sequence of *elementary row operations*, where the *pivoting row* is multiplied by a non-zero scalar and updated into another row below, which has non-zero in the *pivoting column*. As a result, for a sparse

matrix, LU factorization often introduces new non-zero entries, which are known as *fill-ins*. Figure 1 shows an example matrix A, which will be used throughout this paper. Considering row 9 in Figure 1(a), since row 9 has a non-zero at column 5, row 5 needs to perform row operations on row 9. As a result, it produces a new fill-in (9, 8).

As stated previously, a symbolic factorization step is needed to identify the non-zero structures of L and U matrices. The locations of new fill-ins are determined by the following theorem:

Theorem 1. *A fill-in at the index (i, j) is introduced if and only if there exists a directed path from i to j, with the intermediate vertices being smaller than both i and j[34].*

Based on Theorem 1, several algorithms have been proposed to implement symbolic factorization [14, 15, 34]. In this section, we briefly summarize the `fill2` algorithm, which exhibits a high degree of parallelism and thus is suitable for GPUs. The detailed procedure is shown in Algorithm 1. It uses an array $fill$ to indicate an already visited vertex by setting $fill(neighbor) = src$. At the beginning of the algorithm, it performs initialization, and then for each threshold that is treated as a frontier, it checks its neighbors, updates the status of the neighbors, and adds new fill-ins to either $L(src, :)$ or $U(src, :)$, as well as to the $newFrontierQueue(:)$. Subsequently, it will proceed to the next threshold vertex in line 11.

Recently, a GPU implementation of `fill2` algorithm was proposed to accelerate symbolic factorization [11]. As indicated in Algorithm 1, symbolic factorization from each source row is independent. Thus, we can perform parallel traversals from all columns. However, with such independent traversal, each node requires $O(n)$ memory, which leads to $O(n^2)$ overall memory requirements. Though using a distributed collection of resources can increase the aggregate available memory (previous study [11] deployed up to 44 nodes and 264 GPUs), better memory efficiency will be desirable. It should also be noted that their solution just counts the number of new fill-ins in each row, which is insufficient information for the subsequent numeric factorization step (which they do not implement on GPUs).

Algorithm 1 Fill 2 algorithm based on Theorem 1. This algorithm shows the procedure for the srcth row.

Input: src- the src-th row of the matrix, $A(:, :)$-original matrix
Output: Filled matrix L and U

```
1:  fill(:) = 0;
2:  fill(src) = src;
3:  for v in A(src, :) do
4:      fill(v) = src;
5:      if v < src then
6:          add v to L(src, :);
7:      else
8:          add v to U(src, :);
9:      end if
10: end for
11: for threshold = 0:src-1 s.t. fill(threshold) == src do
12:     add threshold to frontierQueue(:);
13:     for each frontier ∈ frontierQueue(:) : do
14:         for each neighbor ∈ A(frontier, :) do
15:             if fill(neighbor) < src then
16:                 fill(neighbor) = src
17:                 if neighbor > threshold then
18:                     add neighbor to L(src, :) or U(src, :);
19:                 else
20:                     add neighbor to newFrontierQueue(:);
21:                 end if
22:             end if
23:         end for
24:     end for
25:     swap (frontierQueue(:), newFrontierQueue(:))
26:     goto line 13;
27: end for
```

2.2 Numeric Factorization and GPU Implementation

The traditional numeric factorization method is right-looking LU factorization:

$$\begin{bmatrix} l11 & \\ l21 & L22 \end{bmatrix} \begin{bmatrix} u11 & u12 \\ & U22 \end{bmatrix} = \begin{bmatrix} a11 & a12 \\ a21 & A22 \end{bmatrix}$$

Here, $l11$, $u11$ are scalars and $l11 = 1$, $l21$ are column vector with size $(n-1) \times 1$, $u12$ is row vector with size $1 \times (n-1)$, and $L22$ and $U22$ are the $(n-1) \times (n-1)$ sub-matrices. To compute matrices L and U, we can first find that $u11 = a11$, $u12 = a12$, and $l21 = a21/u11$. Then, we can solve $L22 \times U22 = A22 - l21 \times u12$. As can be seen, the traditional right-looking method solves one row for the matrix U, followed by one column for the matrix L, and recursively solves the matrix with n iterations. However, this approach has sequential data dependence, which would limit the amount of parallelism.

Thus, to overcome this issue, previous efforts [19, 23, 32] proposed a hybrid column-based right-looking algorithm, which can utilize column-level parallelism. The procedure is shown in Algorithm 2. For each column j, the first step is to compute the L part of the current column, which is shown in lines 2-6. Then, it looks right to find all columns k ($k > j$), which satisfies $As(k, j) \neq 0$. Such columns are called *sub-columns* of the column j. Then, it can factorize all sub-columns of j in parallel as shown in lines 8-14.

To schedule the order of factorization for different columns, the algorithm needs to check the dependence relationship among different columns. For example, for any $U(i, j) \neq 0$, we can conclude that the column j depends on the column i. Specifically, this is because the column j is a sub-column of the column i and the algorithm will read $As(j, i)$ when it factorizes the column i, as shown in lines 8-14 in Algorithm 2. There are other dependencies, but for brevity, we refer the readers to the earlier publication[32]. GLU3.0 then derives

Algorithm 2 The hybrid column-based right-looking algorithm.

Input: *As*-the non-zero filled-in matrix of A after symbolic analysis

```
1: for j = 1; j <= n; j++ do
2:     for k = j+1; k <= n; k++ do
3:         if As(k, j) ≠ 0 then
4:             As(k, j) = As(k, j)/As(j, j)
5:         end if
6:     end for
7:     for k = j+1; k <= n; k++ do
8:         if As(j, k) ≠ 0 then
9:             for i=j+1; i <= n; i++ do
10:                if As(i, j) ≠ 0 then
11:                    As(i, k) = As(i, k) − As(i, j) × As(j, k)
12:                end if
13:            end for
14:        end if
15:    end for
16: end for
```

all dependence information of columns and constructs a *dependency graph*. Figure1(b) shows the dependency graph for matrix A. An edge (i, j) in the graph indicates that the column j depends on the column i. Based on the dependence graph, the algorithm groups columns into *levels* so that columns within a level are independent of each other and thus can be factorized in parallel. Figure 1(c) shows the level information for the matrix A. For example, columns 1, 2, 3, 6, and 7 are independent of each other and their processing can be in parallel. The process for determining such levels is essentially a topological sort but is also called *levelization*.

The GLU3.0 effort observed that potential parallelism keeps changing across the levels. In general, they classified the levels into three categories. In the beginning stage of factorization, the levels are "type A" levels. Such levels typically have a large number of parallelizable columns, while each column has very few associated sub-columns. Thus, they employ one thread block to factorize one column and one warp is assigned to a sub-column. In contrast, type C levels are at the end of the factorization process. In this stage, the levels have a limited number of columns, while each column generally has a large number of sub-columns. To exploit the parallelism of sub-columns, thread blocks are assigned to each sub-column, and kernel calls instead of blocks are assigned to each column. Type B levels, which are in the transitional stage, have great numbers of columns, and at the same time columns also have many sub-columns.

2.3 Limitations of Current Work and Challenges

Previous research efforts demonstrated GPU to use for either the numeric factorization phase [19, 32] or a partial symbolic factorization phase [11]. However, a complete GPU solution to solve the LU factorization has not been proposed. More specifically, most closely related previous work [11] provides a partial solution to GPU-based symbolic execution, i.e. the the scheduling phase is executed on the CPU only. In addition, this work does not integrate symbolic and numerical execution phases.

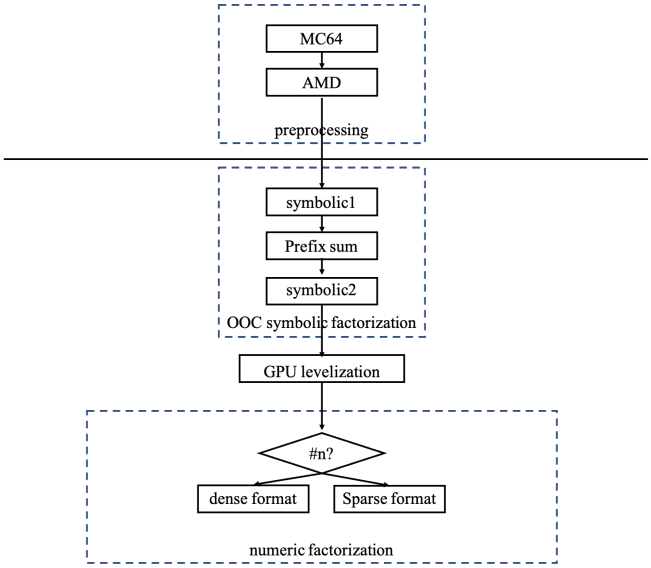


Figure 2. Overall framework of GPU LU factorization for large sparse matrices.

Based on the fact that the device memory of GPUs is getting larger and more applications are being ported on GPUs, we propose to develop the first 'end-to-end' sparse matrix LU factorization implementation. The main challenges in achieving this goal are: 1) the large memory requirements at intermediate stages, and 2) computations involving data dependencies.

3 End-to-end GPU LU factorization

In this section, we first introduce the framework of our end-to-end GPU solution. Then, we show our out-of-core GPU implementation to perform symbolic factorization and a GPU implementation of a topological sort for the purpose of scheduling. Finally, we present optimizations to increase the parallelism for large matrices during the numeric factorization step.

3.1 Overall Framework

The overall framework for our sparse LU factorization is shown in Figure 2. Following the convention, we first perform certain pre-processing steps, i.e. row and column permutations with the goals of reducing fill-ins and improving numeric stability. Then, we perform symbolic factorization in two phases. After that, a parallel version of *levelization* is conducted on the GPU – the output of this step is used to schedule parallel computations during the numeric factorization phase. Finally, there is the numerical factorization

implementation, where our novel contribution is switching to using a sparse data format when the number of rows becomes large – this reduces memory requirements and increases parallelism.

3.2 Out-of-Core Symbolic Factorization with Dynamic Parallelism Assignment

We first illustrate the memory limitation for symbolic factorization. As shown in Algorithm 1, associated with the processing of each source row is the requirement to allocate several arrays and each of which requires O(n) memory space (where n is the number of rows). As a result, O(n^2) memory space is required in total, and exceeds the memory limits even for a relatively small matrix size – it should be noted that the original matrix is a sparse $n \times n$ matrix whose memory requirements are much lower than n^2.

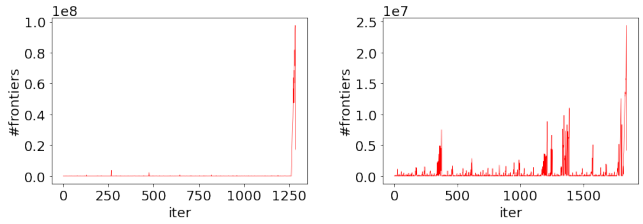
One solution to solve the memory limitation is to use a recently available feature (recent at least in the context of NVIDIA GPUs), called *unified memory* [17, 31]. This feature allows the applications to access the memory on the host side transparently, and data is loaded to the physical GPU memory while servicing page faults. Similar to the concept of virtual memory on a typical CPU operating system, this feature can significantly ease the programming, and indeed, this option has been used for several out-of-core GPU implementations lately [1, 2, 12, 25, 26, 40].

However, it turns out that an implementation based on this approach will have significant additional data movement costs, especially in view of irregular accesses with symbolic factorization – we demonstrate this experimentally later in the paper. As an alternative, we focus on a version where data movement is explicitly controlled. We first propose a naive out-of-core GPU implementation for symbolic factorization, which is shown as Algorithm 3. In the first step, it computes the number of iterations, which is based on the GPU's memory size. Assume that the GPU's device memory size is L. Each source row requires at most $c \times n$ storage for graph traversal to store values like the intermediate vertices – here, c, is a constant whose value turns out to be 6 for this problem. Then,

$$chunk\_size = L/(c \times n).$$

Accordingly, the number of iterations, denoted as num_iter, is $n/chunk\_size$.

We also note that there are two issues in the only other previous work on GPU-based symbolic factorization [11] (which only performed part of symbolic factorization) for our end-to-end out-of-core GPU implementation: First, it only counts the total number of new fill-ins produced during LU factorization. This is not sufficient information for the subsequent numeric factorization, i.e., as can be seen from Algorithm 2, the numeric factorization algorithm requires: 1) the number of new fill-ins of each row 2) the exact positions of each new fill-ins. Second, [11] used a fixed value for



(a) pre2 **(b)** audikw_1

Figure 3. Frontier size (y-axis) per iteration (x-axis)

chunk_size. To guarantee that the intermediate data structures reside on GPUs, it would use a conservative value for *chunk_size* and limit the parallelism degree correspondingly.

Algorithm 3 Naive out-of-core GPU implementation for symbolic factorization.

```
1:  num_iter = n / chunk_size
2:  ▷ Compute the number of the fill-ins in each row
3:  for iter = 0; iter < num_iter; iter++ do
4:      symbolic_1<<<>>> (chunk_size)
5:  end for
6:  ▷ fill_count record the number of non-zeros in each row
7:  prefix_sum(fill_count_d)
8:  Allocate memory for factorized matrix.
9:  ▷ Compute the positions of the fill-ins in kernel symbolic_2
10: for iter = 0; iter < num_iter; iter++ do
11:     symbolic_2<<<>>> (chunk_size)
12: end for
```

To solve the first issue in [11], our implementation contains two stages. For the first stage, we only compute the number of non-zeroes in each row of the factorized matrix. During each iteration, we launch kernel *symbolic_1* to compute the count of non-zeroes for *chunk_size* rows in parallel, which is shown in lines 3-5 in Algorithm 3. The implementation of *symbolic_1* is modified from a recent publication [11]. The counts of non-zeroes in each row are stored in the array $fill\_count$. Since we use the compressed sparse row (CSR) data format to store the factorized matrix, we apply a GPU implementation of prefix sum on the array $fill\_count$ to get the starting position of each row and the total number of non-zeroes (line 7). With this information, we are able to allocate device memory for the factorized matrix (line 8). Finally, for the second stage (lines 10-12), for each iteration, we launch kernel *symbolic_2* for *chunk_size* rows. The main difference in the implementation of *symbolic_2* (compared to *symbolic_1*) is that once we find a fill-in, we also store its location. This is also why $symbolic_1$ needs to be executed ahead of time to find the space requirements.

For the second issue, we observed that the memory requirements for each source row increase as the source row identifier value goes up. More specifically, this is the result

of Theorem 1, i.e., there will be a more significant number of intermediate vertices to consider as the source row identifier value increases, since an intermediate vertex needs to have a small value. We verify this with example matrices *pre2* and *audikw_1*, with results shown in Figure 3 – the intermediate vertices that have smaller identifiers than the source row identifier and need to be actively considered are denoted as the *frontiers*. As can be observed from the figure, the number of the frontiers is usually large for the last few iterations, and small otherwise. Thus, we propose a dynamic parallelism assignment implementation, which is summarized as Algorithm 4. We first partition the rows into two parts: the first part contains $n1$ rows and the second has the remaining ($n2$) rows – here, $n1$ is the number of rows before we see a 'large' number of frontiers, which we define as 50% of the highest number of frontiers we come across. The key difference is that we calculate and use distinct *chunk_size* for these two parts. Specifically, for the first part, the memory requirement corresponding to each row is smaller and thus we assign a large *chunk_size* to increase the parallelism. Then, we compute the number of iterations for each part, which is shown in lines 1-2 in the algorithm. Finally, for each part, we launch kernel *symbolic_1* iteratively to count non-zeroes, which is shown in lines 4-6 and 8-9, respectively. The procedure of the second stage is similar, which is omitted in the algorithm. Note that in carrying out this optimization, using more than 2 phases can be explored, but it will also imply more kernel launches.

Algorithm 4 Out-of-core GPU implementation for symbolic factorization with dynamic parallelism assignment.

```
1:  num_iter_1= n1 / chunk_size_1
2:  num_iter_2 = n2 / chunk_size_2
3:  ▷ Compute the number of the fill-ins in each row
4:  for iter = 0; iter < num_iter_1; iter++ do
5:      symbolic_1<<<>>> (chunk_size_1)
6:  end for
7:  ...
8:  for iter = 0; iter < num_iter_2; iter++ do
9:      symbolic_1<<<>>> (chunk_size_2)
10: end for
11: ▷ The second stage is similar ..
12: ...
```

3.3 Parallel Scheduling Procedure on GPUs

Based on the dependency graph, previous efforts calculate the level numbers for each column as follows:

$$level(k) = max(-1, level(c1), level(c2),) + 1$$

where $c1$, $c2$, ... are the children of the node k. The procedure is serial by nature since there are dependencies among different columns: the level number of column k depends on the level numbers of column $c1$, $c2$, and so on. As a result, previous efforts on LU factorization all performed *levelization* on CPUs, and thus did not achieve an end-to-end LU factorization on GPUs. To parallelize *levelization* on GPUs, which is essentially a topological sort, we first note the limitations of previous efforts on mapping this kernel to a GPU. There have been some general GPU graph processing systems [21, 41] that report they can support topological sort. However, none of them explicitly optimize for this algorithm. Some publications [37] explicitly report GPU topological sort implementation, but they use CPU to launch kernels and thus can not fully utilize the parallelism provided by GPUs. In this work, we propose a pure GPU implementation with *dynamic parallelism* feature of cuda. Compared with [37], our *dynamic parallelism* implementation has the following benefits: First, it avoids the synchronizations and data transfers between the CPU and the GPU. Second, with functions called within GPUs, the kernel launch overheads are greatly reduced. The detailed procedure is shown in Algorithm 5. The method assumes that a dependency graph G has been constructed, based on the method mentioned in Section 2 (line 14). Then, we count the in-degree for each node using the kernel *cnt_indegree* (line 15). The actual topological sort procedure starts as we launch the kernel *Topo_Sort* (line 16). Inside the kernel, we first create the *d_queue* data structure, which denotes the nodes to be processed. Initially, this set includes all nodes with no incoming nodes – this is done using a child kernel *cons_queue* (line 4). More specifically, this kernel checks the in-degree for each node, and if the in-degree of a node is zero, this method puts the node into *d_queue* and sets the level number to 0. During each iteration in the loop, a child kernel *update* is launched to update the in-degree values of the neighbors for the nodes in the *d_queue*. Then, we construct a new *d_queue* for such neighbors (*cons_queue* procedure, line 9). We increase the level

Algorithm 5 Parallel levelization implementation on GPUs.

```
1:  __global__ void Topo( ){
2:  level_num = 0
3:  ▷ d_queue is the queue of all nodes with no incoming edges
4:  cons_queue<<<>>>(...)
5:  level_num++
6:  while qsize > 0 do
7:      update<<<>>>(...);
8:      qsize = 0;
9:      cons_queue<<<>>>(...)
10:     level_num++;
11: end while
12: }
13: procedure Levelization
14:     cons_graph<<<>>>(...)
15:     cnt_indegree<<<>>>(...)
16:     Topo<<<>>>(...);
17: end procedure
```

number at the end of each iteration in line 10. We repeat this procedure until there are no nodes with no incoming nodes, i.e. q_size becomes zero.

Compared to the existing work on this problem we are aware of [37], our improvement lies in calling functions within the GPU, as opposed to using CPU to launch kernels. While a direct comparison is not possible as the baseline code is not available, we can expect significant improvement as kernel launch overheads are removed. The computation complexity of the sequential topological sort is $O(N+M)$ where N is the number of nodes and M is the number of edges. The span (longest execution steps) of parallel topological sort is the number of levels.

Algorithm 6 Binary search to access As(i,j).

Input: col_offset - column offset of CSC format.
row_id - row ids of CSC format.
val - values of CSC format.

```
1: ...
2: fs = col_offset[j]
3: fe = col_offset[j+1]
4: while fe >= fs do
5:     mid = (fs + fe) / 2
6:     if row_id[mid] == i) then
7:         As(i,k) = As(i, k)- val[mid] × As(j,k)
8:         break
9:     else if row_id[mid] > i then
10:        fe = mid - 1
11:    else
12:        fs = mid + 1
13:    end if
14: end while
```

3.4 Increasing Parallelism by Removing Memory Limits for Numeric Factorization

Previous numerical factorization implementations on GPUs - specifically the GLU implementations [19, 32, 33] – all used a dense format for matrix $As(i,j)$ (see Algorithm 2). To elaborate further, in Algorithm 2, we need to search a row id i, which is larger than column id j. Thus, when we use a dense format, we can access data efficiently because the position is direct i. However, we observe that this increases the total memory requirements, limits the number of rows that can be stored in a chunk, and thus reduces the amount of parallelism. Assume that the total available device memory is L, and the maximum parallelizable columns M can be calculated as:

$$M = \frac{L}{n \times sizeof(data\ type)}$$

, where n is the number of vertices. Since we use a thread block to perform the numeric factorization for one column, M denotes the maximal possible concurrent thread blocks. As can be seen from this expression, M would be smaller when n keeps getting larger. Eventually, as n gets very large, M could be smaller than the maximal number of concurrent threads (denoted as TB_max). This would cause the implementation is not able to utilize sufficient parallelism of GPUs.

Table 1. Specifications of Nvidia Tesla V100.

GPUs	Tesla V100
#SM	80
FP32 CUDA Cores/GPU	5120
Memory Interface	4096-bit HBM2
Register File Size / SM (KB)	65536
Max Registers / Thread	255
Shared Memory Size / SM (KB)	Configurable up to 96 KB
Max Thread Block Size	1024

To solve this issue, we propose to adopt the compressed sparse column (CSC) data format for $As(i,j)$, when we identify that n is larger than $\frac{L}{TB\_max \times sizeof(data\ type)}$. The challenge, however, with this change is that for a given column id j, we are not able to get the row id i that is larger than the column id j directly in this case. Thus, we utilize the *ascending property* in the CSC format and perform a binary search to find the position where the row id i is larger than column id j [1].

The detailed procedure is shown in Algorithm 6. In this algorithm, j denotes the column id and i denotes the row id, and the indexes between col_offset[j] and col_offset[j+1] are sorted. fs denotes the smallest possible indexes to search, which is initialized as col_offset[j] and fe denotes the largest position indexes to search, which is initialized as col_offset[j+1]. In each iteration, we compare the middle value of the indices, mid, with i. If the row id in mid is the same as i, then we have found the index, which is mid. Otherwise, if the row id in mid is larger than i, fe is updated as $mid-1$. Alternatively, if the row id in mid is smaller than i, fs is updated as $mid-1$, and we continue the search.

4 Evaluation and Performance Study

In this section, we present our experimental results on a set of large matrices to demonstrate the effectiveness of our end-to-end GPU implementation. We first show our experimental environment and the features of the selected input matrices. Then, to show the effectiveness of our out-of-core GPU implementation for symbolic factorization, we compare it with both a parallel implementation modified from GLU 3.0 [32] and an optimized *unified memory* implementation. Next, we evaluate the benefits obtained from the optimizations that have been introduced, including the use of sparse data representations during numeric factorization for the largest of the matrices.

[1] Note that our CSC format is sorted.

Table 2. Input matrices where the memory requirements of symbolic factorization exceed the size of GPU memory.

matrix	abbr	n	nnz	nnz/n
g7jac200sc	G7	59310	837936	14.1
rma10	RM	46835	2374001	50.7
pre2	PR	659033	5959282	9.0
inline_1	IN	503712	18660027	37.0
crankseg_2	CR2	63838	7106348	111.3
bmwcra_1	BMC	148770	5396386	36.3
crankseg_1	CR1	52804	5333507	101.0
bmw7st_1	BM7	141347	3740507	26.5
apache2	AP	715176	2766523	3.9
s3dkq4m2	S34	90449	2455670	27.1
s3dkt3m2	S33	90449	1921955	21.2
onetone2	OT2	36057	227628	6.3
rajat15	R15	37261	443573	11.9
bbmat	BB	38744	1771722	45.7
mixtank_new	MI	29957	1995041	66.6
Goodwin_054	GO	32510	1030878	31.7
onetone1	OT1	36057	341088	9.5
windtunnel_evap3d	WI	40816	2730600	66.9

4.1 Experimental Design

Environment: We conducted our experiments on an Nvidia Tesla V100. The specifications of the GPU are shown in Table 1. The GPUs are attached to an Intel(R) Xeon(R) CPU E5-2680 (2013 Ivy Bridge) running at 2.4 GHz – the CPU contains 14 physical cores and provides hyper-threading with 2 threads for each core, which is used for our baseline implementation. The size of the host memory is 128 GB in our experiments. The host operating system for our experiments is CentOS Linux release 7.4.1708 (Core). Our GPU implementations are based on CUDA 11.2 toolkit and NVCC V11.2.152 is used to compile our programs.

Input Matrices: We select 18 matrices from the SuiteSparse Matrix Collection [7] for detailed study and analysis. These matrices were selected because LU factorization was possible on these, and as we verified, the memory requirements for the intermediate data structures exceed the size of the device memory of the Nvidia Tesla V100. In other words, for each of these matrices, symbolic factorization, cannot be executed on a GPU without explicit data movement or the use of unified memory. The specifications of the matrices are shown in Table2. Our experiments use *float* as the data type.

Choice of Baselines: We primarily compare the performance of our out-of-core GPU implementations with a parallel implementation modified from GLU3.0[32], which is a recent efficient implementation. For the symbolic part, because our implementation directly starts with the code of Gaihre *et al.* [11] and improves it on functionality (i.e., calculating nonzero positions for numerical phase and preventing out-of-memory for larger data) instead of performance, a comparison of performance will not have been meaningful. Other efforts for numerical phase optimization have not always made code available, moreover, numerical phase is a small part of the execution time of the full code. Finally, we have extensively compared against another design option, which is to use unified memory.

4.2 Comparison with Modified GLU3.0 implementation

The results of this comparison are shown in Figure 4. The execution times are broken down by time spent on symbolic factorization and numeric factorization respectively. We note that the speedups (for the entire execution) are in the range 1.13-32.65. It can also be seen that the difference between our out-of-core GPU implementation and the GLU3.0 implementation is mainly from the symbolic factorization phase. While the relative performance between the multi-core CPU and GPU varies considerably, GPU speedups seem dependent on the number of non-zeroes per row, nnz/n. When the ratios are larger, the speedups tend to be larger. For example, matrices WI and MI have both the highest values of the ratio nnz/n and among the highest speedups, where AP and OT2 are on the opposite spectrum. This is consistent with the general observation that GPUs become more efficient as computations get (relatively) dense.

4.3 Comparison with Unified-Memory Solution(s)

We further compare our out-of-core GPU implementation with unified memory implementations. First, we note that even the unified memory solution gets limited by the size of the CPU main memory. Thus, for this experiment, we selected 7 out of 18 matrices, for which the intermediate data sizes can fit into CPU main memory but not for GPU device memory. Specifically, these are the matrices with the 7 smallest values of n in Table 2, all having fewer than 41,000 rows. For our experiments, we further tuned the unified memory implementation and tried different optimizations. We found that prefetching the intermediate data structures results in increased efficiency. The results comparing our implementation with the unified memory implementation (prefetching enabled) are shown in Figure 5. As in the previous figure, the execution times are broken down by time spent on symbolic and numeric factorization phases respectively. We see that our out-of-core GPU implementation is 1.06-2.22 times more efficient than optimized unified memory implementation. A closer analysis shows for matrices with a relatively higher density, i.e., WI and MI, unified memory implementation is quite competitive. On the other hand, for R15 and OT2 which have the lowest density, the unified memory overheads are larger. This matches our expectation that as there are fewer computations, the effect of page faults would be larger.

We further created a version of unified memory implementation without any prefetching – this version is limited to symbolic factorization. We compared the execution times of the symbolic factorization phase of our out-of-core GPU implementation with the unified memory-based implementations in Figure 6. As indicated in the figure, without prefetching, unified memory implementation performs worse. Their

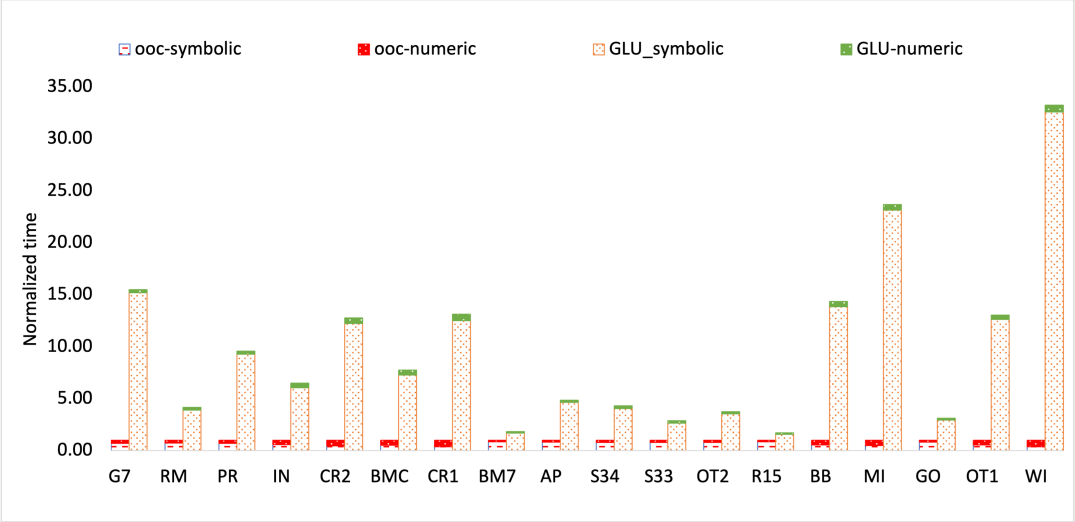


Figure 4. Normalized end-to-end execution times (times for symbolic and numerical phases separated) for out-of-core GPU implementation and the modified GLU3.0 baseline.

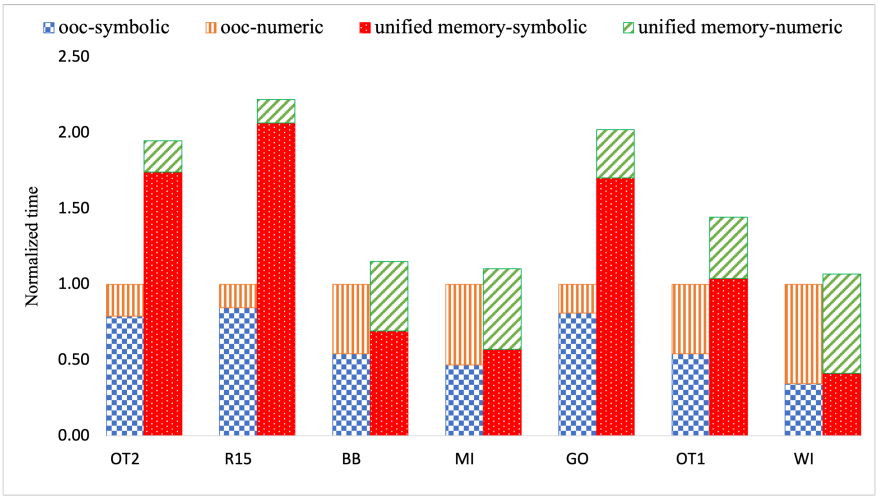


Figure 5. Normalized end-to-end execution times (times for symbolic and numerical phases separated) for out-of-core GPU implementation and a unified-memory GPU baseline

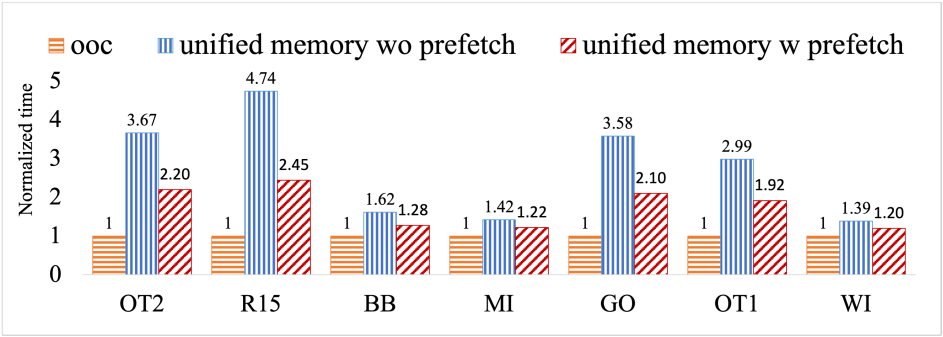


Figure 6. Normalized symbolic execution phase times for our out-of-core GPU implementation, unified memory implementations with and without prefetching.

relative performance gets worse for matrices with lower density, like R15 and OT2.

Further elaboration on the performance differences between the three versions is shown in Table 3. The main performance drawback of *on-demand paging* in the unified memory implementation is the overhead of GPU page faults. As shown in the table, a significant amount of time for unified memory versions is spent servicing page faults, whereas

Table 3. Comparison of the numbers of GPU page fault groups and the percentages of time to service GPU page faults without and with prefetching. *wp* denotes *with prefetching* and *wo p* denotes *without prefetching*.

matrix	# GPU faults wo p	faults wp	pc. wo p(%)	pc. wp(%)	pc. ooc(%)
OT2	16734	4638	78.37	56.60	0.06
R15	17322	4392	86.21	65.46	0.15
BB	19753	5798	46.98	26.94	0.09
MI	12803	4377	36.15	21.61	0.10
GO	13670	3848	78.19	57.39	0.33
OT1	16884	4717	69.58	45.18	0.06
WI	24977	8569	33.11	19.54	0.01

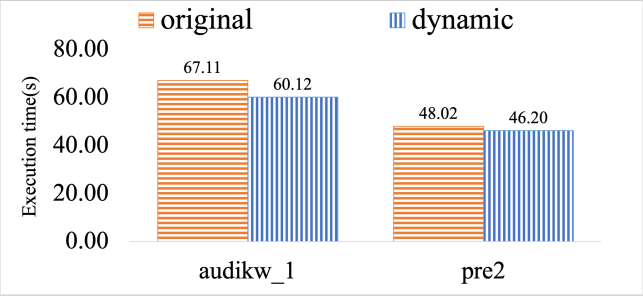


Figure 7. Execution times of our dynamic parallelism assignment implementation and original symbolic factorization implementation.

the out-of-core GPU implementation spends a very small amount of time on data movement. Referring back to Figure 6, we also observe that for matrices with significant computation overheads, such as MI and WI (which happen to be the more dense matrices), the percentages of the time spent on servicing GPU page faults are smaller, and correspondingly, the benefits of our implementation are also smaller.

4.4 Evaluation of Optimizations

Dynamic Parallelism Assignment: We further demonstrate the effectiveness and limitation of dynamic parallelism assignment. We compare our dynamic parallelism assignment implementation with the native out-of-core implementation for symbolic factorization on two large matrices. These matrices are chosen because they are large and the numbers of iterations are large. This comparison is shown in Figure 7. We observed that the *dynamic* implementation achieves up to 10% better performance than the naive implementation. We also noticed that performance improvement is limited by the implementation. During some steps in symbolic factorization, the parallelism degree is determined by the number of frontiers. Thus, when the numbers of frontiers are significantly large, the performance improvement would be limited for these steps.

Memory Optimization for Numeric Factorization: We next demonstrate the effectiveness of our optimizations to increase parallelism for numerical factorization. These benefits are noticed only for very large matrices, with sizes beyond

Table 4. Specifications of large matrices and the maximal number of parallel thread blocks for original version

matrix	Order	nnz	max #blocks
hugetrace-00020	16,002,413	47,997,626	124
delaunay_n24	16,777,216	100,663,202	119
hugebubbles-00000	18,318,143	54,940,162	109
hugebubbles-00010	19,458,087	58,359,528	102

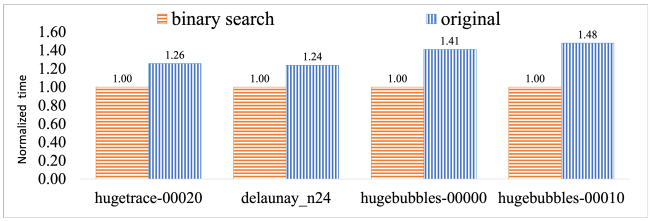


Figure 8. Normalized numeric factorization times of our binary search implementation and original implementation.

the ones we have used for other experiments so far. Specifically, these matrices are shown in Table 4 – for these, n is larger than $\frac{L}{TB\_max \times sizeof(data\ type)}$. Since these matrices happen not to be LU-factorizable (they are not full rank). For our experiments, we replaced their 0 diagonal elements with a non-zero number (1000) to make them factorizable.

Table 4 also shows the maximal number of parallel thread blocks for these matrices. Because the maximal number of thread blocks of our GPU is 160, the original numeric factorization implementation cannot exploit the full parallelism on our GPU [2]. We next compared the execution times of our binary search implementation with the original implementation in Figure 8. As can be seen from the figure, our binary search implementation achieves speedups of 2.88-3.33 since the number of parallel columns is increased.

5 Related Work

In this section, we discuss related efforts on accelerating LU factorization and certain relevant out-of-core implementations proposed in the context of graph processing and linear algebra.

Accelerating LU Factorization: Motivated by the success of supernodal method in accelerating Cholesky factorization for symmetric positive definite matrices (SPD), supernodal LU method was proposed for unsymmetric matrices [9, 10, 27]. Along these lines, Demmel *et al.* [9] introduced five types of unsymmetric supernodes, and more specifically, to exploit the cache more efficiently, they proposed supernode-panel updates and two-dimensional data partitioning for SuperLU. They also further proposed a multithreaded version, SuperLU_MT[10] and a distributed version, SuperLU_DIST [27], to exploit the intra-node and internode parallelism, respectively. Attracted by the enormous parallelism potentials of the GPUs, Gaihre *et al.* proposed GSOFA,

[2]In this experiment, the number of thread blocks for the binary search implementation is fixed to be 160.

which was the first work on accelerating the symbolic factorization on GPUs [11]. While their implementation can deal with distributed environments, they are limited to just determining the number of fill-ins on the GPUs. Davis *et al.*[8], noticed that, for many sparse matrices, such as those from circuit simulation, it is hard to form supernodes or dense parts. Thus, they adopted Block Triangular Form based on Gilbert Peierls (G/P) left-looking algorithm [15]. Chen *et al.* parallelized the KLU algorithm[8] on multi-core architecture by exploiting the column-level parallelism [4, 5]. Chen *et al.* further observed that not every matrix is suitable for a parallel algorithm and proposed a predictive method to decide whether a matrix should use a parallel or a sequential algorithm [3]. He *et al.* proposed GLU implementation to accelerate LU factorization for sparse matrices based on a hybrid right-looking LU factorization algorithm [19]. However, the hybrid right-looking introduces a new type of data dependency, which is called double-U dependency. Furthermore, GLU uses a fixed GPU thread allocation strategy, which limits its parallelism. To solve these issues, Peng *et al.* introduced a relaxed but much more efficient data dependency detection algorithm and developed three different modes of GPU kernel which adapt to different stages in LU factorization [32]. However, these works all deploy the symbolic execution phase on a CPU.

Out-of-Core GPU Implementations: Recently, many research works on out-of-core GPU implementation were proposed focusing on sparse linear algebra and graph computations. Among them, most research efforts focus on the case where the input graph is too large to fit in GPU device memory, which is unlike the challenge for LU where intermediate data size is the likely bottleneck. Generally, there are two major approaches to support GPU out-of-core implementations: partitioning-based and unified memory-based. In the partitioning-based approach, one first partition the input data to chunks such that each chunk can reside in GPU memory, and processes one chunk at a time [18, 22, 39]. To reduce the data transfer overhead which can dominate the time, Sengupta *et al.* [39] proposed to detect and skip partitions that are not needed or are inactive. Han *et al.* [18] further improve the approach, with the adoption of X-Stream style graph processing and renaming techniques to reduce the cost of explicit GPU memory management. Recently, Sabet *et al.* [35] proposed efficient GPU-accelerated subgraph generation techniques to further reduce the data transfer overhead. Besides, they adopt asynchronous execution to reduce the need for subgraph generations and reloading. Another general approach is to adopt the unified memory [17, 31] feature. This feature provides a managed memory space where CPUs and GPUs can observe a single address space with a coherent memory image. With this approach, over-subscription of GPU's memory and an on-demand data migration through page faults is supported. Lately, this has become a popular approach [1, 2, 12, 25, 26, 40], though our work has demonstrated that explicit data management can result in better performance.

6 Conclusions

LU factorization for large sparse matrices is an important scientific computing kernel, though none of the previous work provided a full GPU-based solution. To achieve this goal, we addressed a number of issues: We proposed an out-of-core implementation for the symbolic factorization phase that deals with the memory limits of the GPU, we have presented a dynamic parallelism-based scheduling procedure on the GPU, and to further improve the performance of the numeric factorization, we proposed to switch to sparse data formats when the matrices are very large. Our evaluation has shown that our solution can support large matrices on GPUs obtaining several fold speedups over an efficient solution modified from GLU3.0. Further, our out-of-core GPU implementation for symbolic factorization also outperforms unified memory implementations, especially leading to significantly better performance for very sparse matrices. We have also shown significant performance improvements from our optimizations on the numeric factorization phase.

Acknowledgements: This work was partially supported by NSF awards 2146852, 2131509, 2034850, and 2007793.

References

[1] Tyler Allen and Rong Ge. 2021. In-depth analyses of unified virtual memory system for GPU accelerated computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[2] Ammar Ahmad Awan, Ching-Hsiang Chu, Hari Subramoni, Xiaoyi Lu, and Dhabaleswar K Panda. 2018. OC-DNN: Exploiting advanced unified memory capabilities in CUDA 9 and volta GPUs for out-of-core DNN training. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 143–152.

[3] Xiaoming Chen, Yu Wang, and Huazhong Yang. 2012. An adaptive LU factorization algorithm for parallel circuit simulation. In *17th Asia and South Pacific Design Automation Conference*. IEEE, 359–364.

[4] Xiaoming Chen, Yu Wang, and Huazhong Yang. 2013. NICSLU: An adaptive sparse matrix solver for parallel circuit simulation. *IEEE transactions on computer-aided design of integrated circuits and systems* 32, 2 (2013), 261–274.

[5] Xiaoming Chen, Wei Wu, Yu Wang, Hao Yu, and Huazhong Yang. 2011. An escheduler-based data dependence analysis and task scheduling for parallel circuit simulation. *IEEE Transactions on Circuits and Systems II: Express Briefs* 58, 10 (2011), 702–706.

[6] D Yu Chenhan, Weichung Wang, et al. 2011. A CPU–GPU hybrid approach for the unsymmetric multifrontal method. *Parallel Comput.* 37, 12 (2011), 759–770.

[7] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[8] Timothy A Davis and Ekanathan Palamadai Natarajan. 2010. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Transactions on Mathematical Software (TOMS)* 37, 3 (2010), 1–17.

[9] James W Demmel, Stanley C Eisenstat, John R Gilbert, Xiaoye S Li, and Joseph WH Liu. 1999. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl.* 20, 3 (1999), 720–755.

[10] James W Demmel, John R Gilbert, and Xiaoye S Li. 1999. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Anal. Appl.* 20, 4 (1999), 915–952.

[11] Anil Gaihre, Xiaoye Sherry Li, and Hang Liu. 2021. GSOFA: Scalable Sparse Symbolic LU Factorization on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 1015–1026.

[12] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture*. 224–235.

[13] Thomas George, Vaibhav Saxena, Anshul Gupta, Amik Singh, and Anamitra R Choudhury. 2011. Multifrontal factorization of sparse SPD matrices on GPUs. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 372–383.

[14] John R Gilbert and Joseph WH Liu. 1993. Elimination structures for unsymmetric sparse LU factors. *SIAM J. Matrix Anal. Appl.* 14, 2 (1993), 334–352.

[15] John R Gilbert and Tim Peierls. 1988. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.* 9, 5 (1988), 862–874.

[16] Laura Grigori, James W Demmel, and Xiaoye S Li. 2007. Parallel symbolic factorization for sparse LU with static pivoting. *SIAM Journal on Scientific Computing* 29, 3 (2007), 1289–1314.

[17] Design Guide. 2013. Cuda c programming guide. *NVIDIA, July* 29 (2013), 31.

[18] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 233–245.

[19] Kai He, Sheldon X-D Tan, Hai Wang, and Guoyong Shi. 2015. GPU-accelerated parallel sparse LU factorization method for fast circuit analysis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 3 (2015), 1140–1150.

[20] Arthur B Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (1962), 558–562.

[21] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 239–252.

[22] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. 2016. Gts: A fast and scalable graph processing method based on streaming topology to gpus. In *Proceedings of the 2016 SIGMOD*. 447–461.

[23] Wai-Kong Lee, Ramachandra Achar, and Michel S Nakhla. 2018. Dynamic GPU parallel sparse LU factorization for fast circuit simulation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 11 (2018), 2518–2529.

[24] E Lezar and DB Davidson. 2010. GPU-based LU decomposition for large method of moments problems. *Electronics letters* 46, 17 (2010), 1194–1196.

[25] Chen Li, Rachata Ausavarungnirun, Christopher J Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 49–63.

[26] Lingda Li and Barbara Chapman. 2019. Compiler assisted hybrid implicit and explicit GPU memory management under unified address space. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[27] Xiaoye S Li and James W Demmel. 2003. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software (TOMS)* 29, 2 (2003), 110–140.

[28] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S Duff, and Brian Vinter. 2016. A synchronization-free algorithm for parallel sparse triangular solves. In *European Conference on Parallel Processing*. Springer, 617–630.

[29] Robert F Lucas, Gene Wagenbreth, Dan M Davis, and Roger Grimes. 2010. Multifrontal computations on GPUs and their multi-core hosts. In *International Conference on High Performance Computing for Computational Science*. Springer, 71–82.

[30] Laurence W Nagel. 1975. SPICE2: A computer program to simulate semiconductor circuits. *Memorandom* (1975), ERL–M520.

[31] Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. 2014. Unified memory in cuda 6.0. a brief overview of related data access and transfer issues. *SBEL, Madison, WI, USA, Tech. Rep. TR-2014-09* (2014).

[32] Shaoyi Peng and Sheldon X-D Tan. 2020. GLU3. 0: Fast GPU-based parallel sparse LU factorization for circuit simulation. *IEEE Design & Test* 37, 3 (2020), 78–90.

[33] Ling Ren, Xiaoming Chen, Yu Wang, Chenxi Zhang, and Huazhong Yang. 2012. Sparse LU factorization for parallel circuit simulation on GPU. In *Proceedings of the 49th Annual Design Automation Conference*. 1125–1130.

[34] Donald J Rose and Robert Endre Tarjan. 1978. Algorithmic aspects of vertex elimination on directed graphs. *SIAM J. Appl. Math.* 34, 1 (1978), 176–197.

[35] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: minimizing data transfer during out-of-GPU-memory graph processing. In *European Conference on Computer Systems*. 1–16.

[36] Piyush Sao, Richard Vuduc, and Xiaoye Sherry Li. 2014. A distributed CPU-GPU sparse direct solver. In *European Conference on Parallel Processing*. Springer, 487–498.

[37] Rahul Saxena, Monika Jain, and DP Sharma. 2018. GPU-based parallelization of topological sorting. In *Proceedings of First International Conference on Smart System, Innovations and Computing*. Springer,

411–421.

[38] Olaf Schenk, Klaus Gärtner, and Wolfgang Fichtner. 2000. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. *BIT Numerical Mathematics* 40, 1 (2000), 158–176.

[39] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. Graphreduce: processing large-scale graphs on accelerator-based systems. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[40] Pengyu Wang, Jing Wang, Chao Li, Jianzong Wang, Haojin Zhu, and Minyi Guo. 2021. Grus: Toward unified-memory-efficient high-performance graph processing on gpu. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 2 (2021), 1–25.

[41] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.