GSOFA: Scalable Sparse Symbolic LU Factorization on GPUs

Anil Gaihre, Xiaoye Sherry Li and Hang Liu

Abstract—Decomposing a matrix $\mathbf{A}$ into a lower matrix $\mathbf{L}$ and an upper matrix $\mathbf{U}$, which is also known as LU decomposition, is an essential operation in numerical linear algebra. For a sparse matrix, LU decomposition often introduces more nonzero entries in the $\mathbf{L}$ and $\mathbf{U}$ factors than in the original matrix. A *symbolic factorization* step is needed to identify the nonzero structures of $\mathbf{L}$ and $\mathbf{U}$ matrices. Attracted by the enormous potentials of the Graphics Processing Units (GPUs), an array of efforts have surged to deploy various LU factorization steps except for the symbolic factorization, to the best of our knowledge, on GPUs. This paper introduces GSOFA, the first GPU-based symbolic factorization design with the following three optimizations to enable scalable LU symbolic factorization for *nonsymmetric pattern* sparse matrices on GPUs. First, we introduce a novel fine-grained parallel symbolic factorization algorithm that is well suited for the *Single Instruction Multiple Thread* (SIMT) architecture of GPUs. Second, we tailor supernode detection into a SIMT friendly process and strive to balance the workload, minimize the communication and saturate the GPU computing resources during supernode detection. Third, we introduce a three-pronged optimization to reduce the excessive space consumption problem faced by multi-source concurrent symbolic factorization. Taken together, GSOFA achieves up to 31× speedup from 1 to 44 Summit nodes (6 to 264 GPUs) and outperforms the state-of-the-art CPU project, on average, by 5×. Notably, GSOFA also achieves up to 47% of the peak memory throughput of a V100 GPU in the Summit Supercomputer.

Index Terms—Sparse linear algebra, sparse linear solvers, LU decomposition, static symbolic factorization on GPU

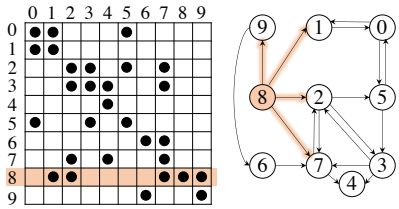
1 INTRODUCTION

Many scientific and engineering problems req
large-scale linear systems, i.e., $\mathbf{Ax} = \mathbf{b}$. Solvin
lem with direct methods [1] often involves L
tion [2], [3], that is, decomposing the original m
lower and upper triangular matrices $\mathbf{L}$ and $\mathbf{U}$,
where $\mathbf{A} = \mathbf{LU}$. Since LU decomposition of a sp
typically introduces new nonzeros, also know
in the $\mathbf{L}$ and $\mathbf{U}$ factored matrices, *symbolic fac*
is used to compute the locations of the fill-ins
and $\mathbf{U}$ matrices. This information is needed to
compressed sparse data structures for $\mathbf{L}$ and $\mathbf{U}$,
subsequent numerical factorization.

Symbolic factorization is a graph algorithm that acts on the adjacency graph of the corresponding sparse matrix. Figure 1 shows a sparse matrix $\mathbf{A}$ and its adjacency graph $G(\mathbf{A})$. In the graph, each row of a matrix corresponds to a vertex in the directed graph with a nonzero in that row corresponding to an out edge of that vertex. For instance, row 8 of $\mathbf{A}$ in Figure 1(a) has nonzeros in columns $\{1, 2, 7, 8, 9\}$. We thus have those corresponding out edges for vertex 8 in $G(\mathbf{A})$ of Figure 1(b). For brevity, we do not include the self edges corresponding to the diagonal elements. We will use this example throughout this paper.

- *A. Gaihre and H. Liu are with Department of Electrical and Computer Engineering, Stevens Institute of Technology, NJ. E-mail: {agaihre, hliu77}@stevens.edu*
- *X.S. Li is with Lawrence Berkeley National Laboratory, Berkeley, CA. E-mail: xsli@lbl.gov*

Manuscript received February 24, 2021; revised May 7, 2021. (Corresponding author: Hang Liu.)



(a) The sparse matrix $\mathbf{A}$. (b) G($\mathbf{A}$): Graph of $\mathbf{A}$.

Figure 1: (a) A sparse matrix $\mathbf{A}$ used throughout this paper where each ● represents a nonzero, (b) G($\mathbf{A}$): the graph representation of $\mathbf{A}$.

The motivation to parallelize the sparse symbolic factorization on GPUs is threefold. Firstly, the newer generation GPUs have a relatively large amount of device memory, which allows the entire sparse matrix to reside on the GPUs. Secondly, more and more exascale application codes are moving to GPUs, which demands the underlying linear solvers to reside on GPUs to eliminate CPU-GPU communication. Ultimately, we will move all the workflows of the sparse LU solver to GPUs. Third, many GPU-based efforts have gone to deploying numerical factorization and triangular solution on GPUs [4], [5] due to (i) numerical factorization and triangular solution take more time, and (ii) symbolic factorization consumes excessive memory space. We argue that since sparse symbolic factorization detects the fill-ins and supernodes for the numerical steps, it is imperative to move sparse symbolic factorization on GPUs.

Existing symbolic factorization algorithms are difficult to deploy on GPUs, as well as scale-up in distributed settings, due to stringent data dependency, lack of fine-grained parallelism, and excessive memory consumption. There mainly exist three types of algorithms to perform symbolic factorization, that is, fill1 and fill2 algorithms by Rose and Tarjan [6] and Gilbert-Peierls algorithm [7], [8]. They are based on traversing either the graphs $G(\mathbf{L})$ and $G(\mathbf{U})$ or graph $G(\mathbf{A})$. In particular, fill1 works on $G(\mathbf{U})$

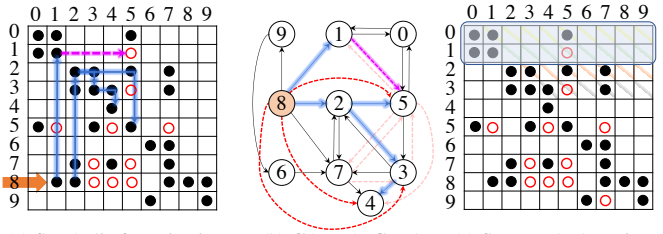
for fill-ins detection while Gilbert-Peierls algo
that on $G(\mathbf{L})$. Fill2 algorithm is analogous t
algorithm [9]. Consequently, fill1 and Gilbert-P
rithms are sequential in nature as the fill-in c
larger rows or columns has to wait until the
of the smaller rows or columns. Fill2 algorithm
grained parallelism (i.e., in each source) due t
ority requirement, although coarse-grained para
at source level) exists. Furthermore, symbolic f
needs to identify supernode [10], which is us
dite the numerical steps. But supernode detecti
cross-source data dependencies. Third, we need
parallel symbolic factorization for a collection
saturate the computing resources on GPUs. This introduces overwhelming space consumption problem, which is similar to multi-source graph traversal [11].

In this paper, we choose the fill2 algorithm as a starting point stemming from the fact that fill2 directly traverses the original instead of the filled graphs, i.e., allowing independent traversals from multiple sources. However, we have to revamp this algorithm to expose a higher degree of *fine-grained* parallelism in order to match GPUs' SIMT nature and reduce the excessive memory requirements during the multi-source symbolic factorization to, again, fit GPUs, which often equip limited memory space. To summarize, we make the following three major contributions.

First, to the best of our knowledge, we introduce the first fine-grained parallel symbolic factorization that is well suited for the SIMT architecture of GPUs. Particularly, instead of processing one neighbor of each source at a time, we allow all the neighboring vertices to be processed in parallel, such that the computations align well with the SIMT nature. *Since this relaxation enables traversal to some neighbors before their dependencies are satisfied, we further allow revisitation to these vertices to ensure correctness*. We also introduce optimizations to reduce the reinitialization overhead and random memory access to the algorithmic data. Finally, we use multi-source concurrent symbolic factorization to saturate each GPU and achieve intra-GPU workload balance.

Second, we not only tailor supernode detection into a SIMT friendly process but also strive to *balance the workload*, *minimize the communication* and *saturate GPU computing resources* during supernode detection. Particularly, we break supernode detection into two phases to expose massive parallelism for GPUs. Further, we assign a chunk of continuous sources to one computing node to avoid inter-node communication and interleave the sources of each chunk across GPUs in each node to balance the workload. Eventually, we investigate the configuration space of unified memory and propose a design that can significantly reduce page faults during supernode detection.

Third, we introduce a three-pronged optimization to combat the excessive space consumption problem faced by multi-source concurrent symbolic factorization: (i) We propose the external GPU frontier management because the space requirement of frontier-related data structures is very dynamic, and their access pattern is predictable. (ii) We identify and remove the unused allocations in vertex status-related data structures; and (iii) since various data structures present dynamic space requirements with respect to different sources, we propose allocating single memory space for all these data structures and dynamically adjust their capacities to reduce the frequency of copying frontier-related data structures between CPU and GPU memories.



(a) Symbolic factorization on **A**. Row 8 under analysis. (b) **G(L+U)**: Graph representation of **L+U**. (c) Supernode detection on **L+U**.

Figure 2: (a) Symbolic factorization on sparse matrix with row 8 under analysis **A**, (b) the Graph representation of filled matrix **(L+U)** with row 8 under analysis and (c) Supernode detection on filled matrix. The ○ is the new nonzeros (fill-ins) in the **L** and **U** factors. And ⇢ / ⇢ indicates the new edges in the $G(\mathbf{L}+\mathbf{U})$ graph. The dark edges with blue or pink glows in (a) and (b) show the traversal paths that lead to new fill-ins.

The rest of this paper is organized as follows: Section 2 introduces the background of this work. Section 3 presents the design challenges. Sections 4, 5 and 6 present the fine-grained symbolic factorization algorithm design, parallel efficient supernode detection, and space consumption optimization techniques. We evaluate GSOFA in Section 7, study the related work in Section 8, and conclude in Section 9.

2 Background

2.1 Sparse LU Factorization

Factorizing a sparse matrix **A** into the **L** and **U** matrices often involves several major steps, such as preprocessing, symbolic factorization, and numerical factorization.

Matrix preprocessing performs either row (partial pivoting) or both row and column permutations (complete pivoting) in order to *improve numerical stability* and *reduce the number of fill-ins* in the **L** and **U** matrices. For numerical stability, existing methods aim to find a permutation so that the pivots on the diagonal are big, for instance, by maximizing the product of diagonal entries and make the matrix diagonal dominant [12]. Supplemental file illustrates how matrix preprocessing works with matrix **A**. Regarding fill-in reduction, the popular strategies are minimum degree algorithm [13], Reverse Cuthill-McKee [14] and nested dissection [15], or a combination of them, such as METIS [16].

Symbolic factorization. Symbolic factorization can be viewed as performing Gaussian elimination on a sparse matrix with either 0 or 1 entries. At each elimination step, the pivoting row is multiplied by a nonzero scalar and updated into another row below, which has nonzero in the pivoting column; this is called an elementary row operation. After a sequence of row operations, the original matrix is transformed into two triangular matrices **L** and **U**. Considering row 8 in Figure 2(a), since row 8 has nonzeros at columns 1 and 2, both rows 1 and 2 will perform row operations on row 8. Using row operation from row 2 as an example, it produces a fill-in (8, 3), which further triggers row operations from rows 3 and 5 that produce fill-ins (8, 4) and (8, 5). We will detail this process shortly. The following fill-path theorem succinctly characterizes the fill-in locations reflecting the above elimination process.

Theorem 1. A fill-i
only if there exis
intermediate vert

Theorem 1 can b
graph $G(\mathbf{U})$ by the
$G(\mathbf{A})$ by the fill2 alg

Figure 3(a) shov
in three steps: (i) l
tierQueue(:). While
newFrontierQueue(:
array that tracks wh
If the neighbor verte
we mark this neighb
src at line 13, and ac
based upon the loca
is smaller than *src*,
next iteration travers
under fill1 algorithm
$2 \rightarrow 3 \rightarrow 4$ leads to
that the third path g

Figure 3(b) prese
it uses fill(:) array t
setting fill(neighbor) = src. The algorithm differs from fill1 mainly on two points. (i) It uses the original graph G($\mathbf{A}$) for traversal (line 12). Second, during traversal, this algorithm only permits traversing one vertex (i.e., threshold) at a time, starting from the smallest one (line 9 - 10). For each threshold that is treated as a frontier, this algorithm checks its neighbors, updates the statuses of the neighbors in fill(:), and adds new fills to either $\mathbf{L}$(src,:) or $\mathbf{U}$(src,:), as well as to the newFrontierQueue(:) if this neighbor obeys Theorem 1. This process continues until the vertices that are smaller and connected to the threshold vertex are exhausted, i.e., frontierQueue(:) is empty. Subsequently, fill2 will proceed to the next threshold vertex in line 9. Considering row 8 in Figure 2(b) again, the three fill-ins are due to the three paths going through only existing edges: $8 \rightarrow 2 \rightarrow 3$ leads to (8, 3), $8 \rightarrow 2 \rightarrow 3 \rightarrow 4$ leads to (8, 4), $8 \rightarrow 2 \rightarrow 5$ leads to (8, 5).

A decade later, [8] introduces the Gilbert-Peierls algorithm to find the fill-in structures, which is a simpler way of interpreting fill1 algorithm. Specifically, this approach determines the nonzero structures column by column. For clarity, we define that $\mathbf{L}(i:j, m:n)$ and $\mathbf{U}(i:j, m:n)$ denote the block from rows i to j, and columns m to n in $\mathbf{L}$ and $\mathbf{U}$ matrices, respectively. For column k, it traverses the graph $\mathbf{L}(:, 0:k-1)^T$ in a Depth-First Search (DFS) manner. The vertex that is reachable by the vertices in column k results in a fill-in at column k. The graph used by [8] is similar to that of fill1 except for that [8] applies a transpose to the $\mathbf{L}$ matrix. Further, the vertices that can be reached from the source vertices will automatically satisfy Theorem 1 because only the graph of $\mathbf{L}(:, 0:k-1)^T$ is used.

It is worth noting that the nonsymmetric-pattern sparse LU symbolic factorization is much harder than the symmetric-pattern counterpart: (i) In the symmetric case, the transitive reduction of the filled graph $G(\mathbf{L})$ is a tree, called elimination tree; symbolic factorization using the elimination tree can be done in time $O(nnzeros(\mathbf{L}))$, linear to the output size. (ii) However, for the nonsymmetric

```
1: Input: src, A(:, :)
2: Output: src-th row of L(src, : ) and U(src, : )
3: Initialization: fill(:) = 0; //Only init once
4: fill(src) = src;
5: forall v in A(src, :) do
6:     fill(v) = src;
7:     if  v < src
8:         add v to frontierQueue(:) and L(src, :);
9:     else  (add v to U(src, :));
10: forall frontier ∈ frontierQueue(:) do
11:     forall neighbor ∈ U(frontier, :) do
12:         if fill (neighbor) < src
13:             fill (neighbor) = src;
14:             add neighbor to L(src, :) or U(src, :);
15:             if (neighbor < src)
16:                 add neighbor to newFrontierQueue(:);
17: swap (frontierQueue(:), newFrontierQueue(:))
    and goto line 10;
```

(a) Fill1 algorithm

```
1: Input: src, A(:, :)
2: Output: src-th row of L(src, : ) and U(src, : )
3: Initialization: fill(:) = 0; //Only init once
4: fill(src) = src;
5: forall v in A(src, :) do
6:         fill(v) = src;
7:         if  v < src  add v to L(src, :);
8:         else  add v to U(src, :);
9: for threshold = 0:src-1 s.t. fill(threshold) == src do
10:    add threshold to frontierQueue(:);
11:    forall frontier ∈ frontierQueue(:) do
12:        forall neighbor ∈ A(frontier, : ) do
13:            if fill (neighbor) < src
14:                fill (neighbor) = src;
15:                if neighbor > threshold
16:                    add neighbor to L(src, :) or U(src, :);
17:                else
18:                    add neighbor to newFrontierQueue(:);
19:    swap (frontierQueue(:), newFrontierQueue(:))
       and goto line 11;
```

(b) Fill2 algorithm

Figure 3: Fill1 and fill2 algorithms. We use MATLAB format to denote both arrays and matrices. The snippet with the same background colors in the algorithms represent equivalent sections of these two algorithms.

cases, the transitive reduction of the filled graphs $G(\mathbf{L})$ and $G(\mathbf{U})$ are Directed Acyclic Graphs (DAGs), called elimination DAGs. Computing these DAGs is expensive, and all variants of this method on the nonsymmetric symbolic factorization algorithms take asymptotically longer than linear time [6], [8].

Numerical factorization. After the structure of the fill-ins is determined, the solvers perform numerical factorization to calculate the values of the $\mathbf{L}$ and $\mathbf{U}$ matrices. Popular numerical factorization methods are *left-looking* [8] and *right-looking* [2]. We explain how numerical factorization works on matrix $\mathbf{A}$ in supplemental file.

Triangular solution. Once LU factorization arrives at $\mathbf{A} = \mathbf{LU}$, we can solve $\mathbf{Ax} = \mathbf{b}$ in two steps with triangular solver, given $\mathbf{Ax} = \mathbf{b}$ becomes $\mathbf{LUx} = \mathbf{b}$. First, triangular solver can solve $\mathbf{Ly} = \mathbf{b}$ to derive $\mathbf{y}$. Second, triangular solver derives $\mathbf{x}$ by solving $\mathbf{Ux} = \mathbf{y}$.

2.2 Supernode

Symbolic factorization also needs to identify the supernodes in $G(\mathbf{L} + \mathbf{U})$, which is important to improve the performance for both numerical factorization and triangular solver. Particularly, *a supernode is a range of rows or columns with the same nonzero structures*, such as rows 1 and 2 in Figure 2(c), so that these rows or columns can be treated as a dense matrix [10]. During numerical factorization, a dense matrix format allows us to use Level 3 Basic Linear Algebra Subprograms (BLAS) operations such as matrix-matrix multiplication, which is often faster than the lower level BLAS operations like Level 2 BLAS operations. In this paper, GSOFA supports T3, one of the most popular types of supernodes among five types of supernodes [10]. Definition 1 defines a T3 supernode.

Definition 1. Supposing a T3 supernode begins at row r and extends through row $s-1$, row s belongs to T3 if and only if $nnz(\mathbf{U}(s, :)) = nnz(\mathbf{U}(s-1, :)) - 1$ and $\mathbf{L}(s, r) \neq 0$ according to [10].

Definition 1 states that for a supernode that already contains rows r through $s-1$, two requirements are needed for the next row s to be included in this supernode: (i) The number of nonzeros in row s of $\mathbf{U}$ shall be one fewer than that of row $s-1$. (ii) There shall be a nonzero at (s, r) in the

filled matrix $\mathbf{L}$. If either of the requirements is not met, row s will not belong to the supernode. Intuitively, (i) a nonzero at $\mathbf{L}(s, r)$ ensures that all the nonzero patterns of row r in $\mathbf{U}$ are mapped into row s during symbolic factorization. (ii) Further, because the nonzero count of row s is one fewer than that of row $s-1$ in $\mathbf{U}$, we can conclude rows $s-1$ and s follow the same nonzero pattern in $\mathbf{U}$. Thus, they belong to the same supernode.

Now, we use the example in Figure 2(c) to illustrate how to detect a supernode in $G(\mathbf{L}+\mathbf{U})$. Starting from row 0, we check whether (i) the number of nonzeros in row 1 of $\mathbf{U}$ matrix (i.e. 2) is one fewer than row 0 (i.e., 3); and (ii) there is a nonzero at (1, 0). Since both requirements are met, row 1 is included in the supernode starting at row 0. Applying a similar process to row 2, we find that row 2 has neither one less nonzero than the previous row (i.e., row 1) nor a nonzero at (2, 0). Hence, row 2 starts a new supernode.

2.3 Graphics Processing Units

This section discusses general-purpose GPUs with recent NVIDIA V100 GPU [17] as an example.

Streaming processors and threads. The V100 GPU is designed with NVIDIA Volta architecture. V100 is powered by 80 Streaming Multiprocessors (SMX). Each SMX features 64 CUDA cores, resulting in a total of 5,120 CUDA cores. During execution, a GPU thread runs on one CUDA core. A SMX schedules a group of 32 consecutive threads known as a warp in a SIMT manner. A collection of consecutive warps further formulate a Cooperative Thread Array (CTA), or a block. All the CTAs together in one kernel are called a grid.

Memory architecture. V100 comes with two memory capacities, that is, 16 GB and 32 GB, with peak bandwidth up to 900 GB/s. Each SMX has 96 KB on-chip fast memory that is shared by the configurable shared memory and L1 cache. All the SMXs share a L2 cache at a size of 6,144 KB. Each thread block can use up to 65,536 32-bit registers.

Fine-grained parallelism. GPUs favor fine-grained parallelism. Particularly, GPUs can only achieve the aforementioned ideal computing and memory throughput when a warp of threads is working on the same instruction and fetching data from consecutive memory addresses. Otherwise, GPUs might suffer from either warp divergence or uncoalesced memory access issues, resulting in order of magnitude performance degradation [17].

Unified memory [18] can unite the memory space of all GPUs and CPUs into a single virtual address space. During execution, any process or thread can access the data from this virtual address space. In the background, the required data is either transferred from where the data resides to where the data is needed implicitly or directly accessed remotely. This is different from the traditional explicit method, which requires programmers to explicitly call *cudaMemcpy()* (or similar functions) in order to transfer the data. Consequently, unified memory bests explicit transfer when the requested data is either small (in terms of size) or randomly accessed, or both. Further, unified memory can be used without terminating the kernel, which is not possible from the explicit method.

Matrix (A)	Abbr.	Order (A)	nnz (A)	Struct. symm.	$\frac{\text{nnz(A)}}{\text{Order(A)}}$	$\frac{\text{\#Fill-in}}{\text{nnz(A)}}$
BBMAT	BB	38,744	1,771,722	0.53	45.7	18.29
BCSSTK18	BC	11,948	149,090	1	12.47	6.52
EPB2	EP	25,228	175,027	0.67	6.93	9.28
G7JAC200SC	G7	59,310	717,620	0.03	12.1	24.51
LHR71C	LH	70,304	1,528,092	0	21.7	3.10
MARK3JAC	MK	64,089	376,395	0.07	5.9	28.59
RMA10	RM	46,835	2,329,092	1	49.729	3.14
AUDIKW_1	AU	943,695	77,651,847	1	82.28	31.43
DIELFILTER	DI	1,157,456	48,538,952	1	41.93	22.39
HAMRLE3	HM	1,447,360	5,514,242	0	3.8	32.63
PRE2	PR	659,033	5,834,044	0.33	8.8	20.70
STOMACH	ST	213,360	3,021,648	0.85	14.2	25.77
TWOTONE	TT	120,750	1,206,265	0.24	10	6.07

Table 1: Dataset specifications. Note, in graph terminology, order ($\mathbf{A}$) and nnz($\mathbf{A}$) represent $|V|$ and $|E|$ of the graph $G(\mathbf{A})$, respectively, where $\mathbf{A}$ is the matrix of interest.

2.4 Dataset

Table 1 presents the datasets that are used to evaluate GSoFA. They are available from Suite Sparse Matrix Collection [19]. This dataset collection includes a variety of applications, such as circuit simulation (HM, PR, and TT), structural problems (BC, AU), computational fluid dynamics (BB and RM), thermal problems (EP), economic modeling (G7 and MK), chemical engineering (LH), electromagnetics (DI) and bioengineering problems (ST). Besides, this dataset collection covers a wide range of variations in both structural symmetry and sparsity (the fifth and sixth columns in Table 1). This table arranges the datasets into smaller (upper) and larger (lower) collections with respect to order ($\mathbf{A}$). The larger matrices are used in space complexity analysis in Sections 6 and 7. Following SuperLU_DIST [20], we use ParMETIS [21] library to preprocess the matrix, and adopt Compressed Sparse Row (CSR) format to represent the matrices [22], [23].

3 Design Challenges

Challenge #1. Existing symbolic factorization algorithms present limited fine-grained parallelism.

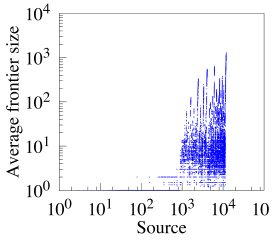
Comparing the fill1 and fill2 algorithms, we can see that fill1 is severely limited in parallelism due to data dependency, because, to find the nonzero structure of the current row, we need to wait for the completion of detecting all the nonzero structures of the previous rows in $\mathbf{U}$. In comparison, the fill2 algorithm exhibits a high degree of parallelism: because graph $G(\mathbf{A})$ is static, we can perform parallel independent traversals from all vertices in $G(\mathbf{A})$. Therefore, an algorithm variant based on fill2 is more favorable for a massively parallel device like GPU.

Despite that fill2 presents more parallelism, fill2 also faces the limited *fine-grained parallelism issue* because line 9 - 10 of fill2 in Figure 3(b) only allows processing one threshold vertex at a time. It is important to note that *simply allowing fill2 to process multiple threshold vertices in parallel will not warrant the correctness* because the threshold value has to be strictly incremented sequentially. This is due to the constraint that each vertex can only be visited once in fill2. An example of this argument is discussed in Section 4.

To mitigate this bottleneck, GSoFA allows all frontiers to be processed in parallel, such that a warp of GPU threads

can work on consecutive t
frontiers might be worked on
we introduce new data struc
revisitations in order to ensure

Challenge #2. Parallel s
good trade-off between l
tion reduction.



(a) BC: small dataset

Figure 4: Average frontier siz
we only present two datasets

Theorem 1 indicates tha
source generally increases w
ID, because more intermedi
the source ID when the sc
tal results in Figure 4 corr
trend is that the workload s
ID. For instance, the workload ratios between the smallest and largest sources are 1,265× and 6,230× for BC and TT datasets, respectively.

We note that the optimization to balance the workload contradicts the optimization for supernode detection. On the one hand, as suggested by the workload dynamic of different sources in Figure 4, a proper workload balancing strategy would require interleaving the sources across GPUs. On the other hand, supernode detection has to check a continuous range of sources, implying that assigning a continuous range of sources to one GPU would minimize the communication cost. These contradictory goals require novel source scheduling and system optimizations.

In this paper, we first transform supernode detection into a massively parallel process that fits GPUs. Subsequently, we introduce a trade-off to both balance the workload and significantly reduce the communication cost during supernode detection with the help of a judicious source scheduling mechanism and an interesting unified memory-based data sharing design.

Challenge #3. The auxiliary data structures may consume overwhelming memory space.

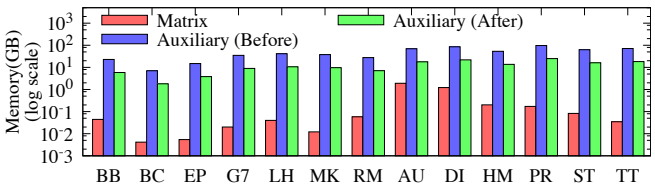


Figure 5: GSOFA memory consumption of the original matrix vs the auxiliary data structures on a Summit compute node.

While the fast runtime is important for symbolic factorization, so is the small space consumption. As shown in Figure 3(b), the original fill2 algorithm requires three auxiliary

```
1: Input: maxId(:), src, A(:,:)
2: Output: src-th row of L(src,:) and U(src,:)
3: Initialization: maxId(:)= maxVal; //Initialize every maxVal/|V| number of src
4: maxId (src) = 0;
5: forall v in A(src,:) fill(v) = 0, maxId (v) = 0, add v to either L(src,:) or U(src,:);
6: forall v in A(src,:) such that v < src, add src to frontierQueue(:);
7: forall frontier ∈ frontierQueue(:) in parallel do //Fine-grained parallelism
8:     newMaxId = Maximum of frontier and maxId(frontier);
9:     forall neighbor ∈ A(frontier,:) in parallel do //Fine-grained parallelism
10:        if ((atomicMin of maxId(neighbor) and newMaxId) > newMaxId)
11:            if neighbor > newMaxId
12:                if ((atomicMax of fill(neighbor) and src) < src)
                       //if not detected as fill-in before
13:                    add neighbor to either L(src,:) or U(src,:);
14:            else continue; //Avoid re-insertion to frontierQueue(:)
15:        if (neighbor < src) atomicAdd neighbor to newFrontierQueue(:)
16: swap (frontierQueue(:), newFrontierQueue(:));

Line 9.5
if fill(neighbor) == src continue;
//Enable line 9.5 when there exist many fill-ins in the graph.
```

Figure 6: GSOFA parallel algorithm. The data structures are in MATLAB format. $\mathbf{L}(\mathbf{src},:)$ and $\mathbf{U}(\mathbf{src},:)$ represent the $\mathbf{L}$ and $\mathbf{U}$ structures of the row **src**.

arrays, fill(:), frontierQueue(:) and newFrontierQueue(:). When we improve the fill2 algorithm with combined frontier queue traversal, we need to add two new arrays: tracker(:) and newTracker(:). In addition, we need maxId(:) to allow the revisitation mechanism to avoid false negatives. Altogether, each source needs six nontrivial auxiliary data structures, each of which consumes memory space of size $\mathcal{O}(|V|)$. Further, since we often need tens to thousands of concurrent sources to saturate a GPU (discussed in Section 4), the memory requirement of the initial version of GSOFA for certain matrices becomes significantly large in comparison to the available memory on GPUs. Figure 5 presents the GPU memory consumption for the sparse matrix and the auxiliary data structures (before and after space optimization) mentioned in Table 2 on one Summit node. One can observe that the memory requirement for the auxiliary data structures is orders of magnitude larger than that for the matrices. In particular, maximum and minimum memory consumption ratios for auxiliary data structure and matrices are 3121:1 in MK and 36:1 in AU, respectively.

Section 6 presents a series of optimizations to reduce the auxiliary data structures memory consumption while maintaining similar performance. As shown in Figure 5, these optimizations reduce the memory consumption of the auxiliary data structure by an average factor of 4×. The maximum and minimum ratios of memory consumption for auxiliary data structure and matrices are reduced to 801:1 in MK and 9:1 in AU.

4 A New Fine-Grained Parallel Symbolic Factorization Algorithm

GSOFA addresses the first challenge, i.e., limited fine-grained parallelism within traversal for a single row, by *i) allowing parallel processing of the frontiers, (ii) adding a new data structure maxId(:) array of size $|V|$, and (iii) allowing revisitation of vertices.* Below we explain the data structure and algorithm design separately. Formally, from the source vertex src to a specific vertex v, there often exist multiple paths, and each path has a maximum numbered vertex. GSOFA uses maxId(v) to store the minimum of all the maximums of the paths from src to v.

GSOFA allows repeated updates to maxId(:) along the traversal. And the traversal terminates once the maxId values of all the vertices converge to their minimal value.

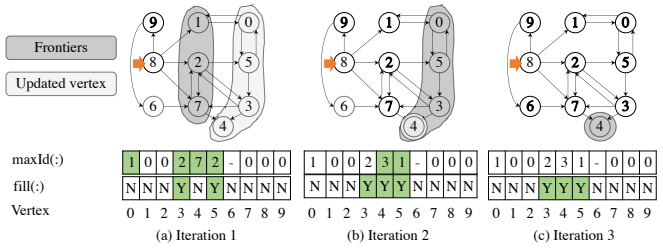


Figure 7: GSOFA traversing the graph from Figure 1(b). The dark, and light gray regions, respectively, represent the current frontier and vertices with maxId(:) updated. Vertex 8 is the source. The maxId(:) and fill(:) are at size of $|V|$. Note, Y and N in fill(:) are respectively used to show if the corresponding vertex is detected as fill-in or not.

As shown in Figure 6, in the beginning, the neighbors of *src* that are smaller than src are eligible for continuing traversal along with these neighbors. Hence they are inserted into frontierQueue(:) at line 6. During traversal, the algorithm allows all the frontiers to explore the graph in parallel. With initial setting of 0 for every vertex, fill(v) of a vertex v in the algorithm is set to *src* if there is a nonzero (src, v) in the filled graph. The value of fill(:) helps avoid the re-detection of fill-ins at line 12. For a vertex v, fill(v) $<$ *src* means that vertex v is not yet in the filled structures of row *src*.

GSOFA fulfills two tasks in lines 9 - 15. The first task is to check whether a neighbor introduces a fill. The second task is to decide whether a vertex can become a frontier. For both tasks, we need the new path from the frontier to the current neighbor to change the maxId of this neighbor, that is, maxId(neighbor) should be updated by newMaxId at line 10. The function atomicMin updates the maxId(neighbor) by the minimum of maxId(neighbor) and newMaxId atomically at line 10. For the first task, this neighbor further needs to satisfy Theorem 1. And this neighbor should not have introduced a fill before, i.e., line 12. Otherwise, we face the issue of re-insertion of this neighbor into either $\mathbf{L}$ or $\mathbf{U}$. For the second task, a neighbor further needs to meet another criterion in order to become a frontier, that is, this neighbor is smaller than the source (line 15). Otherwise, this neighbor either cannot be a frontier.

Example. To aid the understanding of GSOFA, Figure 7 demonstrates how GSOFA traverses from $src = 8$ on the graph of Figure 1(b). After line 5, the values in maxId(:) corresponding to the vertices {1, 2, 7, 8, 9} become 0, while the rest remain as $|V|$ (i.e., '-' in this case).

At iteration 1, the frontiers {1, 2, 7}, in parallel, traverse their neighbors {0}, {3, 5, 7} and {2, 4}, arriving at updated maxId(:) and fill(:) array at the bottom of Figure 7(a). Subsequently, we obtain 1, 2, 7 and 2 as the maxId along the paths $8 \rightarrow 1 \rightarrow 0$, $8 \rightarrow 2 \rightarrow 3$, $8 \rightarrow 7 \rightarrow 4$ and $8 \rightarrow 2 \rightarrow 5$. Clearly, these paths introduce fills at 3 and 5 as shown in the fill(:). Proceeding to iteration 2, frontiers {0, 3, 4, 5} generate new paths as $8 \rightarrow 1 \rightarrow 0 \rightarrow 5$, $8 \rightarrow 2 \rightarrow 3 \rightarrow 4$ and $8 \rightarrow 2 \rightarrow 5 \rightarrow 3$. Even path $8 \rightarrow 1 \rightarrow 0 \rightarrow 5$ updates maxId(5) at line 10, it will not introduce a fill due to line 12, thus 5 will not be enqueued into newFrontierQueue(:). Path $8 \rightarrow 2 \rightarrow 5 \rightarrow 3$ stops at line 10 because the stored maxId(3) = 2 is smaller than the newMaxId = 5. Finally, path $8 \rightarrow 2 \rightarrow 3 \rightarrow 4$ qualifies line 10 since the stored maxId(4) = 7 is greater than newMaxId = 3, as well as introduces a new fill due to line 12 is true. Afterwards, only 4 is in the newFrontierQueue(:) at iteration 3 which will not introduce any new frontiers.

It is important to note that the parallel traversal to all the neighbors of a frontier is not possible by simply allowing to process multiple threshold vertices in parallel, as mentioned in Challenge #1 (lines 9 - 10 of fill2 algorithm). Now, we discuss how the detection of the fill-in (8,4) from Figure 7 may not occur when we allow multiple threads to traverse the neighbors {1, 2, 7} in parallel in fill2. At line 6 in fill2 algorithm, the fill(:) entries of all the neighbors {1, 2, 7} of source 8 are marked as 8. Assuming multiple threads can work in parallel at line 9, implying the thresholds will be 1, 2 and 7 from three different threads, respectively. In that case, all the neighbors 1, 2 and 7 will be inserted into the frontierQueue(:). When accessing the neighbor of the frontier at line 12, if a thread working on frontier 7 explores vertex 4 before any other threads, fill2 algorithm will update fill(4) = 8 without detecting a fill-in at (8, 4) because the path $8 \rightarrow 7 \rightarrow 4$ does not satisfy Theorem 1. Later, the thread that could detect a fill at (8, 4) because of path $8 \rightarrow 2 \rightarrow 3 \rightarrow 4$ fails to do so because this thread cannot enter the branch at line 13 since fill(4) is already updated as 8. Hence, the fill-in (8, 4) remains undetected if we seek fine-grained parallelism in straightforward fill2 algorithm at Figure 3(b).

Optimizing the initialization of maxId(:). Maintaining a maxId(:) array for every traversal would require an excessive amount of space that becomes impractical for large datasets considering limited memory space on modern GPUs. Hence reusing maxId(:) array for different sources becomes essential. But this reuse also comes with reinitialization overhead, that is, making maxId(:) to maxVal before traversal. And this overhead is nontrivial, e.g., it takes 22% of the total time for PR dataset to re-initialize the maxId(:).

To reduce the reinitialization overhead, we propose to divide the value range $[0, maxVal]$ of maxId(:) into smaller ranges, i.e., $[0, |V|]$ range for each source. Note that maxVal is $2^{32}-1$ for a 32-bit unsigned integer type. During traversal, different sources can work on their respective value range of maxId(:) without reinitialization. For instance, for the first source, the traversal updates the maxId of the vertices in the range of [maxVal - $|V|$, maxVal]. Moving to the next source, we regard the range of [maxVal - $2 \cdot |V|$, maxVal - $|V|$] as valid for maxId. In this context, any maxId value greater than the upper bound, which is maxVal - $|V|$ in this case, is treated as initialized. Note, the value of maxId will not be smaller than maxVal - $|V|$ before execution. This optimization helps skip the maxId(:) initialization for a total of $\frac{maxVal}{|V|}$ sources. For instance, it helps reduce the ratio of reinitialization over the total time consumption from 22% to 0.082% for PR dataset.

Optimizing the access to maxId(:) and fill(:). It is important to mention that the accesses to both maxId(:) and fill(:) array are random thus time-consuming. One can either access maxId(:) first to reduce the follow-up access to fill(:) or vice versa (i.e., adding line 9.5 to the pseudocode). In both cases, we avoid repeated frontier enqueuing for vertices that are already detected as fill-ins at line 14.

Putting maxId(:) access before fill(:), which is the pseudocode in Figure 6, will avoid the access to fill(:) array when the new path fails to update the maxId of existing paths, i.e., line 10 is evaluated as false. Consequently, this path avoids

the access to follow-u
Figure 7(a) falls in thi

Adding line 9.5 to
unnecessarily loweri
in. Note, we do not
always propose its c
paths that come acro
example, this logic a
when the path $8 \rightarrow 1$
5) is already a fill du
of the paths from 8 th

Including line 9.5
Particularly, adding li
a relatively larger nu
relatively smaller nur
will become false for
overhead than benefi

Multi-source syn
frontierQueue(:) exec
in Figure 6 concurren
the computing resources. To balance the workload across threads, we combine the frontiers of various concurrent traversals into a single frontierQueue(:), which is analogous to recent multi-source graph traversal projects [11]. We further make three revisions to this design. First, we need to use maxId(:) instead of a single bit to track the status of each vertex. Furthermore, GSOFA relies upon tracker(:) and newtracker(:) to identify the source of each frontier. Last but not least, we directly use atomic operations to enqueue frontiers into the combined frontier queue, which is faster than prefix-sum based approach, according to [24].

5 Parallel Efficient Supernode Detection

This section further introduces a *massively parallel supernode detection algorithm* which can *balance the workload*, *minimize the communication* and *saturate the GPU resources* with judicious source scheduling and unified memory assistance.

GPU-friendly supernode detection with shared memory optimization. We introduce a two-phase supernode detection design to match the SIMT nature of GPUs. First, we use one GPU thread to examine whether the nonzero count of a row in $\mathbf{U}$ is one fewer than the previous row (i.e., requirement i), and use a bitmap in shared memory to keep track of this information. If requirement (i) is not met, it means the current row starts a new supernode. We will use a queue in shared memory to keep track of this leading row. Second, each GPU thread dequeues a leading row of a supernode, examines the bitmap, also assesses the requirement (ii) of a supernode in Definition 1. Only when both requirements are met, the current row is considered to be part of this supernode. *The novelty of this two-phase design is that the first phase helps break the chunkSize of rows into independent subranges so that our second phase can work on these subranges in parallel, which fits the SIMT nature of GPUs.* In addition, the first phase is also massively parallel.

Figure 8 demonstrates how to, in parallel, detect supernodes for row ranges 0 - 3 of matrix $\mathbf{U}$. In phase I, all threads in parallel check whether the nnz of current row is 1 fewer than the prior row. Since only row 1 satisfies the condition, the bitmap is set to "0100" (❶). Further, the

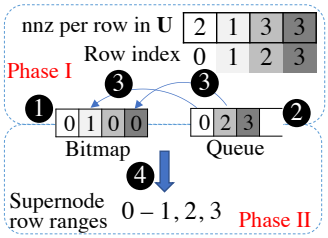


Figure 8: Parallel supernode detection for matrix $\mathbf{U}$ in Figure 2(c), where Phase I confirms $nnz(\mathbf{U}(s,:)) = nnz(\mathbf{U}(s-1,:)) - 1$ while Phase II checks $\mathbf{L}(s,r) \neq 0$.

leading rows of supernode are {0, 2, 3} in queue (❷). In phase II, each thread is assigned to one leading row entry and checks whether the bit following the leading row is 1 (❸), as well as the corresponding entry in $\mathbf{L}(s,r) \neq 0$ (❹). If both conditions are met, the supernode grows. We will continue this process until no supernode grows. Since row 1 satisfies both conditions while row 2 does not, supernode of row 0 grows to include only row 1.

Communication- and saturation- aware inter-node source scheduling. Since inter-node communication is significantly more expensive than intra-node counterpart, we restrict the supernode detection inside of each computing node. We use the following equation to perform inter-node source scheduling:

$$\begin{aligned} &chunkSize \times numConcurrChunksPerNode \\ &= \#C \times numGPUsPerNode, \end{aligned} \tag{1}$$

where $chunkSize$ is identical to the size of the user defined maximum supernode. Note, we want the size of a chunk to be neither a fraction nor multiple of the maximum supernode size. On the one hand, since we interleave consecutive chunks to different nodes, making $chunkSize$ a fraction of the maximum supernode size will introduce inter-node communication during supernode detection. On the other hand, making $chunkSize$ a multiple of supernode size will make chunk size too large, potentially lead to worse workload imbalance. Further, $\#C$ is the preferred number of sources to saturate one GPU. In short, all the three items in Equation (1), i.e., $chunkSize$, $\#C$, and $numGPUsPerNode$ are known, one can derive the value for $numConcurrChunksPerNode$. For example, the PR dataset on one Summit node would need 6,144 sources to saturate the GPUs. If the defined supernode size is 128, *numConcurrChunksPerNode* would be 48.

Unified memory assisted intra-node source scheduling. Once the number of concurrent chunks for a computing node is decided, we assign the sources of these chunks to various GPUs of this node in an interleaved fashion in order to balance the workload. However, since supernode detection requires the nonzero count of consecutive rows, communicating the fill information between GPUs is needed. Specifically, for a supernode spanning from row r to $s-1$, one needs to communicate the *nonzero count* of row s in $\mathbf{U}$, and whether there is a *nonzero at* $\mathbf{L}(s,r)$. Since both data sizes to communicate are small and we cannot predict which nonzero count and fill location are required beforehand, traditional *cudaMemcpy* based data transfer is not suitable for such a kind of data sharing.

To this end, we choose the unified memory option over the explicit data transfer. In this context, GSOFA uses a single CPU process to manage all the GPUs in each node, where the kernel launches are performed asynchronously. It is also

important to note that unified memory introduces several configurations which concern the performance.

First, we choose remote access over page migration when accessing the unified memory. Since we interleave the sources assignment in a fine-grained manner, various GPUs might compete to migrate the same page that stores the fill(:) or nonzero count information from the source GPU. This will hurt the performance. During implementation, our key guideline is *keeping this data in the GPU that modifies this data*. Particularly, only one GPU will detect the fill information for a specific row (i.e., modify that fill(:)), so does that for the supernode detection. This implies that only one GPU needs to keep the fill(:) information of that row locally while the remaining GPUs will access that information remotely. In implementation, we use the *cudaMemAdviseSetPreferredLocation* flag to advise the unified memory driver to keep that data in the same GPU which modifies this data [25]. For the remaining GPUs, we use *cudaMemAdviseSetAccessedBy* flag to instruct them to access that data remotely.

Second, GSOFA allocates separate fill(:) memory for different GPUs instead of using one virtual space spanning across all GPUs in one node. This design is inspired by a key observation that there exist a significant number of page faults when all the GPUs share one fill(:) address. This is caused by the fact that this single memory space is not perfectly aligned from one GPU to another. That is, one page could span across two GPUs. In this case, once both GPUs need to modify that page, these GPUs will compete for that shared page for writing. This will lead to frequent page migration.

6 SPACE OPTIMIZATION

Continuing our discussion in Challenge #3 of Section 3, this section will rigorously quantify the *space crisis faced by multi-source concurrent symbolic factorization*. Table 2 presents the space complexity of the six major data structures used by GSOFA, namely, *frontierQueue(:)* & *newFrontierQ* *tracker(:)* & *newtracker(:)*, *maxId(:)*, and *fill(:)*. When th ber of concurrent sources is #C, the total space consu would be around $6 \cdot |V| \cdot \#C$ entries. Apparently consumption immediately becomes a key problem f graphs with relatively big number of concurrent s However, when it comes to performance, GSOFA a larger #C which will provide more workload t saturate the GPU computing resources.

Data structure	Space complexit
frontierQueue(:) & newFrontierQueue(:)	$2 \cdot \|V\| \cdot \#C$
tracker(:) & newTracker(:)	$2 \cdot \|V\| \cdot \#C$
maxId(:)	$\|V\| \cdot \#C$
fill(:)	$\|V\| \cdot \#C$

Table 2: The data structures used by multi-source co GSOFA, where $|V|$ and #C are the number of vertices in the graph, and concurrent sources, respectively.

Dataset	Average usage (%)	Peak usage (%)
AU	0.12	8.2
DI	0.04	4.55
HM	0.01	1.7
PR	0.11	25.0
ST	0.1	1.0
TT	0.1	11.0

Table 3: Percentage of usage of allocated frontier queues.

In this section, we propose three interesting optimizations based upon the access pattern and usage of various data structures to combat the high space complexity. Particularly, we introduce external GPU frontier management, bubble removal in maxId(:), dynamic space allocation to dynamically assign memory space to various data structures. Note, this dynamic space allocation also allows GSOFA to support configurable space consumption.

External GPU frontier management is motivated by the observation in Table 3 where the average usage of frontier-related data structures remains low for the large datasets, i.e., AU, DI, HM, ST and TT. However, the peak usage can rise to as high as 25% for PR. This observation implies that we can allocate relatively smaller memory space to hold frontier-related data structures because the usage remains low for vast majority of the iterations. Once the usage goes beyond the allocated space, we resort to our external-GPU option, which is presented below.

We propose to only allocate a fraction of the required space on GPU for the four frontier-related data structures, i.e., frontierQueue(:), newFrontierQueue(:), tracker(:) and newtracker(:) and write the extra frontiers out of GPU. Then, we load these external GPU data structures in GPU for computation. Figure 9 demonstrates this design. For brevity, Figure 9 only uses a single thread to traverse the graph for source 8 without loss of generality. The size of the allocated newFrontierQueue(:) is only three. At iteration 1, frontiers 1 and 2 exhaust the newFrontierQueue(:) by their neighbors 0, 5, and 3. Subsequently, we copy these three frontiers to CPU memory to still have available space for the incoming frontiers, 7 in this case. Proceeding to the next iteration that comes after swapping the queues, we will finish frontier 7 in the frontierQueue(:) first and load the frontiers from CPU in GPU for further computation. It is worthy of mentioning that, instead of directly storing the source ID in tracker(:) and newtracker(:) in multi-source concurrent traversal, we propose to store the index of the source for each frontier to

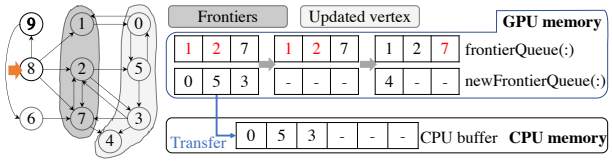


Figure 9: External GPU frontier management. Note, the traversal iteration is same as in Figure 7(a).

Bubble removal in maxId(:) is supported by the key observation that a source vertex is not allowed to traverse vertices that are larger than the source. Consequently, we can remove the "bubbles" in the maxId(:) that are larger than the source. Here, "bubble" means the allocated space that is not used in maxId(:). Assuming the source vertex is v, according to Theorem 1, we will never update the maxId of the vertices that are larger than v.

Dynamic space allocation across data structures is motivated by two facts. First, maxId(:) and fill(:) accesses are more random than a frontier-related data structure. Particularly, the accesses to maxId(:) and fill(:) are determined by the frontier's neighbors which often have random vertex IDs. Therefore, GSOFA needs to put the entire maxId(:) and

fill(:) arrays in GPU memory in order to achieve desirable performance. Second, the space requirement maxId(:) is dynamic, that is, smaller sources need smaller maxId(:) and vice versa. Therefore, we can dynamically adjust the maxId(:) space so that the frontier-related data structures can have more GPU resources when possible.

Towards this end, we first allocate a big chunk of memory, in contrast to allocate separate memory spaces for various data structures discussed in Table 2. Subsequently, this memory chunk is dynamically divided among the data structures with priority given to maxId(:) and fill(:). For a given number of concurrent sources in a traversal, after the space reduction strategy of maxId(:), the amount of memory required for maxId(:) remains low for smaller sources. In this scenario, we can allocate more space for other data structures. Once we start working on larger sources, the maxId(:) space requirements begin to climb. In this context, we prioritize the space requirement for maxId(:) along with fill(:) so that a large number of concurrent sources can execute together with better, at least sustained, performance.

GSOFA with configurable space budget. After putting all the major data structures of GSOFA into one continuous memory space, we enable a new feature, i.e., configurable memory budget for GSOFA. This optimization will first conduct bubble removal, dynamic space allocation and external GPU frontier-related data structure management. If GSOFA still suffers from space shortage, GSOFA will judiciously reduce the number of concurrent sources to restrict the memory space consumption in the given budget.

7 EVALUATION

We implement GSOFA with ∼2,500 lines of C++/CUDA code and compile the source code with NVIDIA CUDA 10.1 Toolkit with the optimization flag set to be O3. We use IBM Spectrum MPI 10.3.0.0 for inter-node communication. The evaluation platform is the Summit supercomputer at Oak Ridge National Laboratory [26], where each computing node is equipped with dual-socket IBM POWER9 CPU processors (i.e., 42 cores), and six NVIDIA V100 GPUs. All the GPUs on one Summit node are connected with NVIDIA's high-speed NVLink. We use Traversed Edges Per Second (TEPS) to report the graph traversal performance and take the average of three runs. We use NVIDIA nvprof profiling tool for Figures 13 and 18.

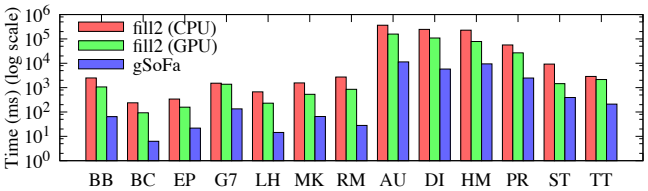


Figure 10: Performance of the parallel CPU and GPU versions of fill2 and GSOFA on a Summit compute node.

Figure 10 compares the original fill2 algorithms on CPU and GPU, and GSOFA on one Summit node. Both CPU and GPU versions of fill2 that we are using are the straightforward parallel implementation of the fill2 algorithm in Figure 3(b). The parallel CPU version employs all the 42 cores in a Summit node to perform symbolic factorization for 42 sources in parallel. The GPU version allows every allocated thread to work in a source as long as the GPU memory is sufficient. On average, GSOFA is 13.01× faster than the GPU-based parallel fill2 algorithm, with the maximum and minimum speedups 30.41× (RM) and 3.70× (ST), respectively. When GSOFA is compared to CPU version, GSOFA enjoys even higher speedups, that is, 33.06×, on average, with 97.11× (RM) and 11.32× (G7) as the maximum and minimum speedups. This suggests that the original fill2 algorithm is not suitable for massively parallel GPUs.

Comparison with the state-of-the-art CPU algorithms. We compare GSOFA with the state-of-the-art sequential symbolic factorization in GLU3.0 [27] and the parallel symbolic factorization from SuperLU_DIST [2]. Note, both the GLU3.0 and SuperLU_DIST may use GPUs only during numerical factorization or later phases. However, both libraries use CPUs for symbolic factorization.

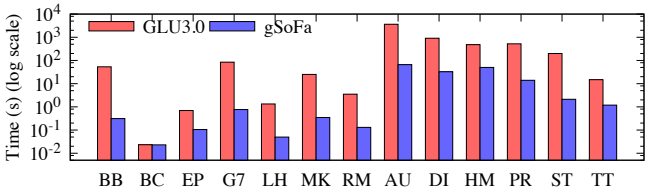


Figure 11: Performance comparison of GSOFA with the state-of-the-art CPU symbolic factorization in GLU3.0. For fair comparison, GSOFA uses one V100 GPU since GLU3.0 performs symbolic factorization sequentially.

Figure 11 demonstrates the time comparison of GSOFA with the symbolic factorization on GLU3.0 [27]. We limit the GSOFA to a single GPU because GLU3.0 can only perform single-threaded symbolic factorization on CPU. Note, GLU3.0 is based upon Gilbert-Peierls algorithm [8] which suffers from stringent data dependency problems if intended to implement in parallel (discussed in Section 2). In the figure, we can observe a maximum and minimum speedup of 171.1× (BB) and 1.01× (BC). The workload in BC dataset is too small to saturate the GPUs, i.e., see Figure 4(a). On average, the GSOFA is 50.1× faster than symbolic factorization in GLU3.0.

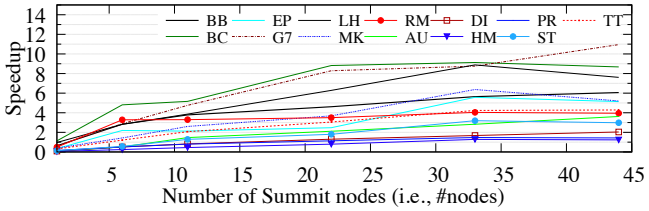


Figure 12: Speedup of GSOFA over the state-of-the-art CPU parallel symbolic algorithm in SuperLU_DIST.

Figure 12 compares the performance between GSOFA and the CPU parallel symbolic factorization in SuperLU_DIST. GSOFA starts worse than the CPU parallel algorithm for majority of the datasets. However, with more and more Summit nodes, GSOFA begins to outperform the CPU algorithm. Particularly, initially on one node, GSOFA is 1.2× faster on BC and 1.6×, 2.8×, 1.6×, 1.03×, 2.6×, 2.0×, 13.8×, 11.3×, 19.3×, 6.7×, 7.4× and 3.2× slower on BB, EP, G7, LH, MK, RM, AU, DI, HM, PR, ST and TT, respectively on one node. When it goes to six Summit nodes, GSOFA bests CPU parallel symbolic factorization on

majority of the datasets, i.e., BC, EP, BB, RM, G7, MK, LH and TT. GSOFA finally outperforms CPU parallel symbolic factorization across all the datasets by 5×, on average, with 44 Summit nodes, at which the maximum speedup of 10.9× is achieved for G7 matrix and minimum of 1.3× is achieved for HM matrix. For G7, the speedup of GSOFA is high because G7 is highly non-symmetric which limits the benefits of symmetric pruning in SuperLU_DIST and its larger workload (i.e., $\frac{\text{nnz(A)}}{\text{Order(A)}}$ in Table 1). HM though has relatively low symmetry, but it presents the lowest workload, which causes the speedup to be poor.

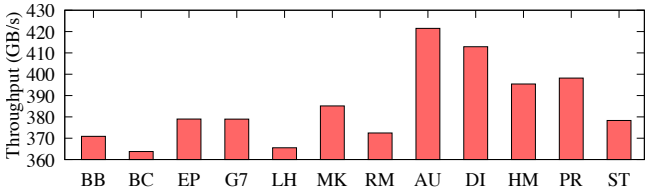


Figure 13: Throughput achieved by GSOFA on a V100 GPU.

Throughput. Figure 13 demonstrates the throughput achieved by the GSOFA on V100 GPU. Particularly, GSOFA achieves a maximum of 421.5 GB/s (47% of peak memory throughput) for AU dataset and a minimum of 363.8 GB/s (40% of peak memory throughput). On average, GSOFA achieves a throughput of 385.2 GB/s (43% of peak memory throughput) over all the dataset, which is rarely observed for graph traversal related applications [28].

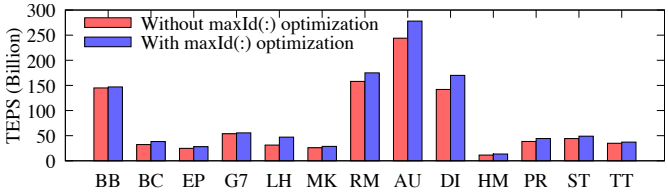


Figure 14: Impacts of maxId(:) initialization optimization.

Performance impact of maxId(:) optimization. This reduces the initialization overhead for maxId(:). As shown in Figure 14, we observe noticeable performance gains across all datasets. On average, we achieve 14% speedup with this optimization, where the maximum impact comes from LH of 50% and a minimum of 2% increment in BB dataset.

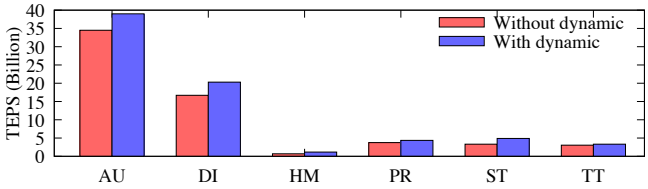


Figure 15: Performance impact of dynamic space allocation on one GPU with 1 GB space allocation.

Performance impact of dynamic space allocation. Figure 15 presents the effect of dynamic memory allocation. We allocate 1 GB of memory and study the performance of *with* versus *without* the dynamic memory optimization. As expected, this optimization yields performance gains for all the large datasets in Table 1. Particularly, we observe improvements of 1.2×, 1.2×, 1.7×, 1.2×, 1.5× and 1.1× on AU, DI, HM, PR, ST and TT respectively.

GSOFA space optimization with limited memory budget. This includes all the space optimizations, further along

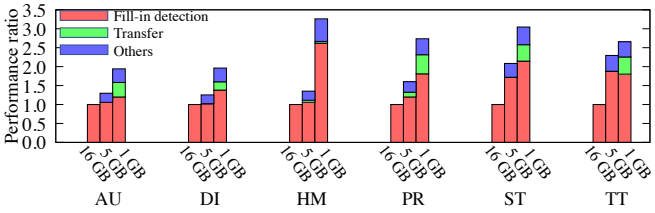


Figure 16: Performance study of GSOFA with different memory capacities on 44 Summit nodes. The allowed GPU memory for all the data structures in GSOFA varies from 1 to 16 GB.

with an explicit restriction on the available memory space. Particularly, we enable external GPU frontierQueue(:), newFrontierQueue(:), tracker(:) and newtracker(:) management, bubble removal and dynamic allocation. For the single large array that is shared by all the data structures, we limit its size to be 16, 5, and 1 GBs to demonstrate the performance robustness of GSOFA. This optimization will involve transferring data between CPU and GPU memories and other overheads, such as checking the condition of memory overflow.

Figure 16 shows the trade-off between the runtime and the space consumption. The general trend is that the performance drops with the decrease of allocated space. Particularly, with merely 1 GB memory budget, the GSOFA performance decreases by 1.9×, 1.9×, 3.3×, 2.7×, 3.0× and 2.6×, respectively, on AU, DI, HM, PR, ST and TT datasets. The performance drops in fill-in detection are caused by the fact that a limited memory budget leads to the reduction of #C. Note, enabling external GPU GSOFA also results in the overhead of checking whether the frontier queue overflows, which is denoted as "GSOFA: Others" in Figure 16.

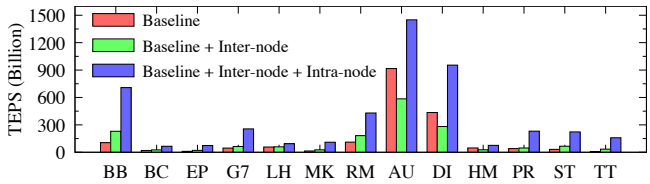


Figure 17: Performance impacts of supernode detection optimizations on six nodes (i.e., 36 GPUs). Here, the inter-node optimization interleaves the chunk of rows among the compute nodes and the intra-node optimization interleaves rows scheduling in fine-grained manner during fill-in detection to obtain optimum intra-node workload balancing.

Performance impact of supernode detection optimizations. In Figure 17 we use six nodes (i.e., 36 GPUs) in order to showcase the impacts of our inter-node workload balancing strategies. Specifically, the "baseline" version assigns a block of continuous chunks of sources to a node. The "inter-node" interleaves the chunks of sources across different nodes in a round-robin approach. And "intra-node" performs unified memory-assisted fine-grained source scheduling across GPUs in a node. On average, the inter-node scheduling optimization yields 1.6× speedup over the baseline. The intra-node optimization further adds another 3.3× speedup. The maximum gains of inter- and intra- node optimizations are 4.3× (TT) and 4.9× (PR), respectively. We also observe performance drops for the inter-node optimization on AU, DI, and HM graphs because the new combination of chunks of sources might have relatively more non-uniform workload distribution.

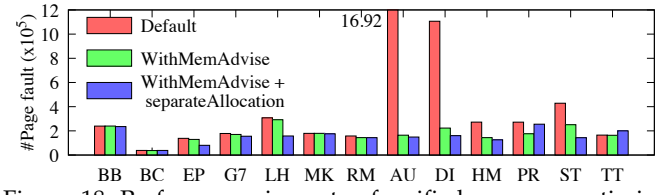


Figure 18: Performance impacts of unified memory optimization on one Summit node.

Performance impacts of unified memory optimizations. Figure 18 demonstrates the effects of cudaMemAdvise and data structure separation optimizations. We can observe a general trend of reduction in the number of page faults with each optimization's addition. Particularly, cudaMemAdvise reduces the page fault by 2.2×. In particular, we see the maximum drop of page faults in AU by a factor of 10.3×. The data structure separation optimization, when added to the cudaMemAdvise version, further reduces the page faults by an average of 20%, with the maximum drop to be 85% in LH. Note, PR and TT experience more page faults due to the data structure separation optimization. This is potentially caused by the fact that the change of memory alignment could result in more page faults for the memory that is not at the boundary.

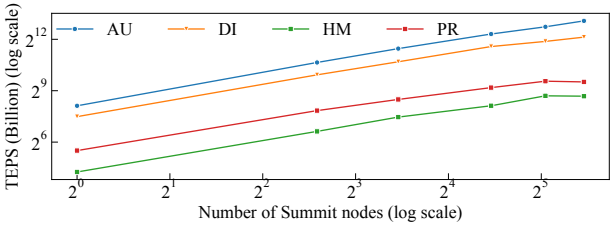


Figure 19: Scaling GSOFA to 44 Summit nodes (264 GPUs).

Strong scalability. Figure 19 demonstrates strong scaling of GSOFA up to 44 Summit nodes (264 GPUs) for relatively large datasets, i.e., AU, DI, PR, and HM. Particularly, GSOFA achieves speedups of 31.0×, 24.9×, 21.5×, and 16.1×, respectively, for AU, DI, PR, and HM. It is worth noting that GSOFA can effectively use a number of GPUs that is not necessarily a power-of-two, which provides great flexibility to the application code. We also notice that the HM and PR enjoy close to linear scalability from 1 to 33 nodes but flat out from 33 to 44 nodes because these two datasets do not have the adequate workload to saturate 44 nodes.

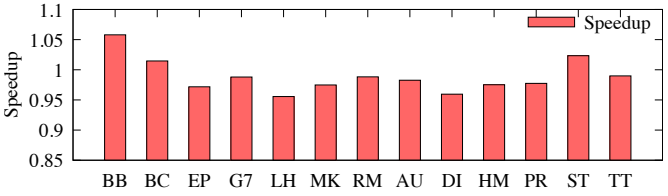


Figure 20: Impacts of using line 9.5 in algorithm in Figure 6.

Performance impacts of optimizing access to maxId(:) and fill(:). Putting fill(:) ahead of maxId(:) introduces observable performance differences. Particularly, as shown in Figure 20 in a GPU, enabling line 9.5 in Figure 6 leads to performance gain in BB, BC and ST datasets with the maximum gain of 5.5% on BB dataset. This performance gain indicates that noticeable fill-ins are repeatedly detected. Likewise, among the remaining datasets, LH dataset experiences a maximum performance drop of 4.6% because the number of maxId(:) update after fill detection of a vertex is small. Given the majority of the datasets experience performance drop with the enabling of line 9.5, we disable line 9.5 for all the datasets in our evaluations.

GSOFA performance variation with *chunkSize*. Figure 21 presents the performance impact of *chunkSize* from Equation 1 to GSOFA. In general, one can observe that the increase of the *chunkSize* leads to longer time. On average, the performance degradation is 1.04× and 1.13× when we increase *chunkSize* from 64 to 128 and 256, respectively, with the maximum slowdown as 1.6× of MK for *chunkSize* = 64 to 256. The reason behind this trend is that the increase of the *chunkSize* typically leads to more imbalanced workload. In this work, we follow SuperLU_DIST to set *chunkSize* = 128 for GSOFA as the default configuration even though chunkSize = 64 is slightly faster.

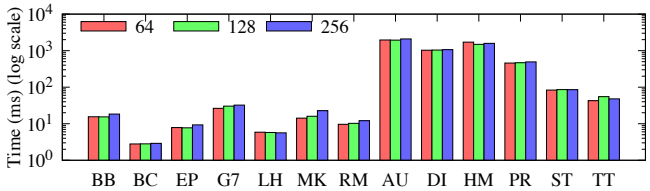


Figure 21: GSOFA performance variation with respect to *chunkSize* = 64, 128 and 256 in six Summit compute nodes.

8 Related Work

Most of the major state-of-the-art sparse LU factorization codes, such as GLU3.0 [27], [29], [30] and SuperLU_DIST [20] adopt the Gilbert-Peierls algorithm [8] for symbolic factorization. GLU3.0, [29] and [30] directly use the sequential version of the Gilbert-Peierls algorithm [8]. SuperLU_DIST implements a parallel CPU-based symbolic factorization algorithm [2] based upon Gilbert-Peierls algorithm with symmetric pruning [31] and supernodal traversal [10]. To improve parallelism for symbolic factorization, SuperLU_DIST resorts to nested dissection to partition **A** into independent rows at a level of the separator tree [2]. The filled structures computed at a level of the separator tree are communicated among the required processes that perform symbolic factorization at the higher level of the separator tree. This leads to inter-process communications while the computation changes the level along the separator tree. To the best of our knowledge, GSOFA is the first GPU-based parallel symbolic factorization algorithm.

It is also worth mentioning that NVIDIA cuSOLVER library [32] provides solvers for sparse and dense linear systems with different approaches including LU decomposition. However, cuSOLVER does not yet support GPU version of sparse LU decomposition. Hence, we cannot compare GSOFA against the symbolic factorization of cuSolver.

Fill2 algorithm is similar to Dijkstra's Single Source Shortest Path algorithm [33] in the sense that fill2 uses the maximum vertex ID on a path to represent the "distance" metric in Dijkstra's algorithm. The salient difference between these two algorithms lies in that one does not need to reduce the "distance" further if a fill-in is already detected in symbolic factorization. We also would like to point out that the previously proposed Δ-step optimization [34] for Dijkstra's algorithm is not effective here: because fill2 only allows vertices that are smaller than the source to be active,

Δ-step will further restrict the parallelism. However, the similarity between these two algorithms suggests that our design of fine-grained symbolic factorization, external GPU optimizations, and supernode detection can potentially provide performance enhancement and space-saving technique to the multi-source Dijkstra's algorithm [35].

9 Conclusion and Future Work

This paper introduces GSoFA the first, to the best of our knowledge, GPU-based sparse symbolic LU factorization system. In particular, we revamp the fill2 algorithm to enable fine-grained symbolic factorization, redesign supernode detection to expose massive parallelism, balanced workload and minimal inter-node communication, and introduce a three-pronged space optimization to handle large sparse matrices. Taken together, we scale GSoFA up to 264 GPUs with unprecedented performance. As the future work, we plan to integrate GSoFA into the state-of-the-art SuperLU_DIST package.

10 Acknowledgement

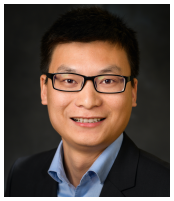
This work was in part supported by NSF CRII Award No. 2000722, CAREER Award No. 2046102, and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration (NNSA). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE, NNSA, or NSF.

References

[1] Timothy A Davis, et al. A Survey of Direct Methods for Sparse Linear Systems. *Acta Numerica*, 2016.
[2] Laura Grigori, et al. Parallel Symbolic Factorization for Sparse LU with Static Pivoting. *SIAM Journal on Scientific Computing*, 2007.
[3] E Lezar et al. GPU-based LU Decomposition for Large Method of Moments Problems. *Electronics letters*, 2010.
[4] Piyush Sao, et al. A Distributed CPU-GPU Sparse Direct Solver. In *Euro-Par*. Springer, 2014.
[5] Weifeng Liu, et al. A Synchronization-free Algorithm for Parallel Sparse Triangular Solves. In *Euro-Par*. Springer, 2016.
[6] Donald J Rose et al. Algorithmic Aspects of Vertex Elimination on Directed Graphs. *SIAP*, 1978.
[7] John R Gilbert et al. Elimination Structures for Unsymmetric Sparse LU Factors. *SIMAX*, 1993.
[8] John R Gilbert et al. Sparse Partial Pivoting in Time Proportional to Arithmetic Operations. *SISC*, 1988.
[9] Edsger W Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische mathematik*, 1959.
[10] James W Demmel, et al. A Supernodal Approach to Sparse Partial Pivoting. *SIMAX*, 1999.
[11] Hang Liu, et al. ibfs: Concurrent Breadth-first Search on GPUs. SIGMOD, 2016.
[12] Iain S Duff et al. The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices. *SIMAX*, 1999.
[13] William F Tinney et al. Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization. *Proceedings of the IEEE*, 1967.
[14] Elizabeth Cuthill et al. Reducing the Bandwidth of Sparse Symmetric Matrices. In *24th National Conference*, 1969.
[15] Brian W Kernighan et al. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 1970.
[16] George Karypis et al. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *Elsevier*, 1998.
[17] Tesla NVIDIA. NVIDIA Tesla V100 GPU Architecture, 2017.
[18] Lingda Li et al. Compiler Assisted Hybrid Implicit and Explicit GPU Memory Management Under Unified Address Space. In *SC*, 2019.
[19] Suite Sparse Matrix Collection. Suite Sparse Matrix Collection. Retrieved from https://sparse.tamu.edu/, 2020. Accessed: 2019, November 15.
[20] Xiaoye S Li et al. SuperLU_DIST: A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric Linear Systems. *TOMS*, 2003.
[21] Karypsis Lab. ParMETIS. Retrieved from http://glaros.dtc.umn.edu/gkhome/metis/parmetis/. Accessed: 2020, June 02.
[22] Hang Liu et al. Enterprise: Breadth-first Graph Traversal on GPUs. In *SC*, 2015.
[23] Duane Merrill, et al. Scalable GPU Graph Traversal. *Acm Sigplan Notices*, 47(8):117–128, 2012.
[24] Anil Gaihre, et al. XBFS: Exploring Runtime Optimizations for Breadth-first Search on GPUs. HPDC, 2019.
[25] NVIDIA. Cuda Toolkit Documentation. Retrieved from https://docs.nvidia.com/cuda/index.html. Accessed: 2020, August 11.
[26] Oak Ridge National Laboratory. Summit: America's newest and smartest supercomputer. Retrived from https://www.olcf.ornl.gov/summit/. Accessed: 2018, August 6.
[27] Shaoyi Peng et al. GLU3.0: Fast GPU-based Parallel Sparse LU Factorization for Circuit Simulation. *IEEE Design & Test*, 2020.
[28] Yangzihao Wang, et al. GUNROCK: A High-performance Graph Processing Library on the GPU. In *PPoPP*, 2016.
[29] Ling Ren, et al. Sparse LU Factorization for Parallel Circuit Simulation on GPU. In *DAC*, 2012.
[30] Kai He, et al. GPU-accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis. *IEEE VLSI Systems*, 2015.
[31] Stanley C Eisenstat et al. Exploiting Structural Symmetry in Unsymmetric Sparse Symbolic Factorization. *SIMAX*, 1992.
[32] CUDA Toolkit Documentation. Available at https://docs.nvidia.com/cuda/cusolver/.
[33] Edsger W Dijkstra et al. A Note on Two Problems in Connexion with Graphs. *Numerische mathematik*, 1959.
[34] Ulrich Meyer et al. Δ-stepping: A Parallelizable Shortest Path Algorithm. *Journal of Algorithms*, 2003.
[35] Piyush Sao, et al. A Supernodal All-pairs Shortest Path Algorithm. PPoPP, 2020.

Anil Gaihre is a Ph.D. candidate at the Department of Electrical and Computer Engineering, Stevens Institute of Technology. His research interests include graph theory, sparse linear algebra, high performance computing in multi-core computer architectures, and blockchain. His prior work experience involves working as a Lead Software Engineer at E&T Nepal Pvt. Ltd. that involved Research and Development on CFD simulations on GPUs.

Xiaoye Sherry Li is a Senior Scientist at Lawrence Berkeley National Laboratory. She has worked on diverse problems in high performance scientific computations, including parallel computing, sparse matrix computations, high precision arithmetic, and combinatorial scientific computing. She has (co)authored over 120 publications, and contributed to several book chapters. She is the lead developer of SuperLU, a widely-used sparse direct solver, and has contributed to the development of several other mathematical libraries, including ARPREC, LAPACK, PDSLin, STRUMPACK, and XBLAS. She is a SIAM Fellow and an ACM Senior Member.



Hang Liu is an Assistant Professor of Electrical and Computer Engineering at Stevens Institute of Technology. Prior to joining Stevens, he was an assistant professor at the Electrical and Computer Engineering Department of University of Massachusetts Lowell. He is on the editorial board for Journal of BigData: Theory and Practice, a program committee member for SC, HPDC, and IPDPS, and regular reviewer for TPDS and TC. He earned his Ph.D. degree from the George Washington University 2017. He is the Champion of the MIT/Amazon GraphChallenge 2018 and 2019, and one of the best papers awardee in VLDB '20.