SnuSOLVER: Optimizing Sparse Direct Solvers for Heterogeneous Systems

Chaewon Kim
Department of Computer Science and Engineering
Seoul National University
Seoul, Republic of Korea
chaewon@aces.snu.ac.kr

Jaehwan Lee
Department of Computer Science and Engineering
Seoul National University
Seoul, Republic of Korea
jaehwan@aces.snu.ac.kr

Jinpyo Kim
Department of Computer Science and Engineering
Seoul National University
Seoul, Republic of Korea
jinpyo@aces.snu.ac.kr

Dohyun Kim
Institute of Computer Technology
Seoul National University
Seoul, Republic of Korea
dohyun.p.kim@snu.ac.kr

Kyusu Ahn*
Research Center
Samsung Display Co., Ltd.
Yongin, Republic of Korea
kyusu.ahn@snu.ac.kr

Hyung Uk Cho
Research Center, Samsung Display Co., Ltd.
Yongin, Republic of Korea
hyunguk.cho@samsung.com

Seungin Baek
Research Center, Samsung Display Co., Ltd.
Yongin, Republic of Korea
s.i.baek@samsung.com

Jaejin Lee
Dept. of Data Science, Dept. of Computer Science and Engineering
Seoul National University
Seoul, Republic of Korea
jaejin@snu.ac.kr

Abstract

Achieving scalability in sparse direct solvers is crucial for addressing the complexity of real-world systems. This paper proposes SnuSOLVER, a library for sparse direct solvers. Unlike conventional approaches, which often apply kernel selection heuristically within supernodal or frontal methods, the SnuSOLVER adopts a structured, two-phase execution strategy tailored to the characteristics of each level in the nested dissection hierarchy. It ensures optimal performance across all hierarchical levels of computation. We evaluate SnuSOLVER on an eight-node heterogeneous cluster with AMD CPUs and NVIDIA GPUs using 31 sparse matrices of varying sizes and domains. Experimental results demonstrate that SnuSOLVER outperforms the state-of-the-art solvers SuperLU_DIST and STRUMPACK and underscore the scalability, efficiency, and adaptability of SnuSOLVER, establishing it as a robust solution for sparse direct solvers on modern heterogeneous computing systems.

CCS Concepts

• **Computing methodologies** → **Parallel computing methodologies**.

Keywords

Sparse LU solver, Scalability, Heterogeneous systems, GPU

ACM Reference Format:
Chaewon Kim, Jaehwan Lee, Jinpyo Kim, Dohyun Kim, Kyusu Ahn, Hyung Uk Cho, Seungin Baek, and Jaejin Lee. 2025. SnuSOLVER: Optimizing Sparse Direct Solvers for Heterogeneous Systems. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721145.3734531

1 Introduction

Solving sparse linear systems is fundamental to improving our understanding of nature. A well-known example is the simulation of the time evolution of large-scale systems described with partial differential equations (PDEs) in domains including circuit simulation, computational fluid dynamics, and structural mechanics. Existing approaches to solving sparse linear systems can be categorized into two classes:

*He conducted this work as a Ph.D. student at Seoul National University.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3734531

direct and *iterative*. The former computes the matrix inverse where some pivoting is involved. The latter applies preconditioning iteratively, resembling numerical methods such as Newton's method. Iterative methods may reduce the computational complexity to less than matrix multiplication at the cost of longer execution times for convergence and stability issues [30]. Direct methods exhibit some benefits over iterative methods, including predictable behavior in accuracy and execution time.

LU factorization plays a key role in direct linear solvers because solving triangular systems of equations is computationally straightforward. However, one major challenge in factorizing sparse matrices is the introduction of fill-ins, new nonzero elements that appear during the process. These fill-ins increase the number of nonzero elements, leading to higher computational loads and significant memory usage, making fill-in minimization a critical objective. To address this challenge, sparse direct LU solvers often interpret matrices as graphs using adjacency matrices [4].

Dissection heuristics, such as nested dissection, are widely used to minimize fill-ins by partitioning the graph into two disconnected subgraphs [9]. By reducing fill-ins, these heuristics improve computational efficiency and memory utilization. Despite the heavy computational load of such algorithms, effectively utilizing heterogeneous computing systems, such as GPU-based systems, is challenging due to the irregular and sparse nature of the computations in sparse direct solvers.

Workload imbalance and irregular memory accesses are two significant challenges in deploying sparse direct solvers on heterogeneous computing systems. Sparse matrices inherently have an uneven distribution of nonzero elements, making it difficult to assign computational tasks evenly across processing units, leading to performance bottlenecks and resource underutilization. As factorization progresses, fill-ins further exacerbate this imbalance. Additionally, the pointer-based data structures, such as CSR or CSC formats, used to represent sparse matrices result in non-sequential memory accesses, which are inefficient for the memory hierarchies of modern CPUs and GPUs.

These challenges hinder both workload distribution and memory bandwidth utilization, particularly in GPU-accelerated environments. Moreover, sparse direct solvers involve frequent inter-process communication, especially in distributed systems. Efficiently overlapping computation and communication is essential to minimize idle time, yet this is complicated by the irregular data dependencies inherent in sparse matrix computations.

This paper addresses these challenges, such as workload imbalance, irregular memory accesses, and communication scheduling, and proposes optimizing techniques for sparse direct solvers for heterogeneous systems.

The main contributions of this paper are summarized as follows:

- We propose SnuSOLVER, a library that employs a two-phase execution strategy based on nested dissection. Computation progresses from bottom to top in a hierarchical submatrix tree. Sparse kernels at lower levels exploit sparsity, while dense kernels at higher levels enhance throughput. This structure reduces synchronization and communication overhead, offering a novel alternative to traditional supernodal and frontal methods.
- SnuSOLVER achieves both high throughput and low overhead by tailoring computations to the characteristics of submatrices. Its hierarchical design enables parallel processing of independent submatrices and minimizes synchronization. Efficient data transfers between processes and across CPU-GPU boundaries further improve scalability.
- We detail our implementation, including data distribution, kernel selection, memory layout optimization, and overlapping computation and communication. Each design choice contributes to performance gains.
- Evaluation on 31 sparse matrices across diverse domains shows that SnuSOLVER outperforms previous solvers on an eight-node heterogeneous cluster (each with one AMD 32-core CPU and four NVIDIA V100 GPUs). It achieves up to 10.82× and 2.82× speedups over SuperLU_DIST and STRUMPACK, respectively, for factorization and solve phases. It establishes a new benchmark for heterogeneous sparse solvers.

2 Background and Related Work

This section provides background and discusses related work to the paper.

2.1 LU Factorization

LU factorization is a fundamental technique for solving systems of linear equations, performing matrix inversion, and computing determinants. It decomposes a given matrix A into the product of a lower triangular matrix L and an upper triangular matrix U. This process involves two primary computational tasks: *panel factorization* and *Schur complement updates*. Panel factorization is executed using operations such as `GETRF` (GEneral TRiangular Factorization) for the initial decomposition and `TRSM` (TRiangular Solve with Multiple right-hand sides) for triangular solves. Schur complement updates, which account for the effect of eliminating rows and columns, are computed using `GEMM` (GEneral Matrix Multiply). This sequence is repeated, block by block, until the entire matrix is factorized [10, 21].

For dense matrices, LU factorization has been extensively studied, leading to highly optimized algorithms and libraries. In the case of sparse matrices, additional complexities arise due to the need to minimize fill-ins, new non-zero elements introduced during factorization, to preserve the sparsity of the matrix. Sparse matrix factorization is crucial for handling large matrices efficiently with limited memory and computational resources. However, applying dense matrix techniques directly to sparse matrices often results in excessive fill-ins and increased computational costs, failing to exploit the sparsity effectively.

Despite these differences, the core tasks of sparse LU factorization—panel factorization and Schur complement updates, remain fundamentally similar to those in the dense case. Over the years, numerous specialized methods [8, 11] have been developed to address the unique challenges of sparsity, ensuring efficient computation while maintaining the structural advantages of sparse matrices. However, existing methods [15, 37] struggle with scalability and efficiency in modern heterogeneous systems, particularly in addressing workload imbalance, irregular memory accesses, and excessive communication overhead.

2.2 Sparse Direct Solver

Sparse direct LU solvers typically follow a sequence of phases: *analysis*, *symbolic factorization*, *numerical factorization*, and *solve*.

The analysis phase preprocesses the matrix to improve numerical stability and computational efficiency. It determines the elimination tree and partitions the matrix to optimize parallel computation. The *nested dissection heuristic* [19, 20] is often employed during this phase to minimize fill-ins. Completely eliminating fill-ins during LU factorization is generally infeasible. Thus, it is essential to reduce them as much as possible. Nested dissection is a widely used and effective strategy for minimizing fill-ins, although the resulting ordering is not guaranteed to be optimal. Libraries like ParMETIS [20] are commonly used to apply the heuristic.

The symbolic factorization phase precomputes the structure of the factorized matrix without performing numerical computations. By doing so, it identifies nonzero fill-ins and allocates memory efficiently in advance. The actual factorization occurs in the numerical factorization phase by filling the predefined structure with numerical values. This phase typically involves repeated panel factorization (`GETRF`, `TRSM`) and Schur complement updates (`GEMM`). Once the matrix is factorized, triangular solves are performed to compute the final solution to the system of equations in the solve phase.

Sparse solvers face significant challenges in optimizing workloads and achieving efficient parallelization, especially in heterogeneous environments. They need to address two key aspects: *high throughput* and *communication minimization*. Maximizing computational throughput with multiple processors is critical for scalability. Techniques like *supernodal method* or *frontal method* are designed to densify computations and exploit high-performance dense matrix kernels, such as `BLAS`, for efficient numerical factorization. By leveraging hierarchical parallelization, methods based on nested dissection can reduce communication and synchronization overhead caused by fine-grained parallelization.

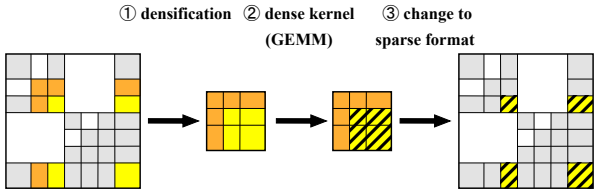


Figure 1: Densification used in supernodal or frontal methods. The hatched area indicates the modification. A portion of the sparse matrix is gathered to form a dense matrix. Dense kernels, such as `GEMM`, are then applied to the dense matrix. The results are subsequently converted back to sparse format and propagated to the original matrix.

2.3 Supernodal and Frontal Methods

The supernodal and frontal methods [3, 11] are two widely used techniques for sparse matrix factorization. Both methods enhance computational efficiency by grouping nonzero elements into dense submatrices, allowing optimized dense matrix operations.

In the supernodal method, nonzero elements are aggregated into supernodes—blocks of columns treated as dense matrices during factorization. Similarly, the frontal method forms dense submatrices (i.e., fronts) during the elimination process. However, zero elements may also be included during the formation of dense matrices, which can increase the computational workload. This introduces a trade-off between computational cost and throughput, making it critical to group elements efficiently to prevent excessive computational overhead. Figure 1 illustrates this *densification* process, where nonzero elements are grouped into dense blocks, enabling efficient computation through dense kernels, such as Intel's oneMKL [18] and NVIDIA's cuBLAS [27].

2.4 Nested Dissection

Nested dissection [5, 14, 25] is a graph partitioning technique that provides a matrix ordering beneficial for parallel sparse matrix computations. It divides an undirected graph $G = (V, E)$ into three parts: two disjoint connected subgraphs and

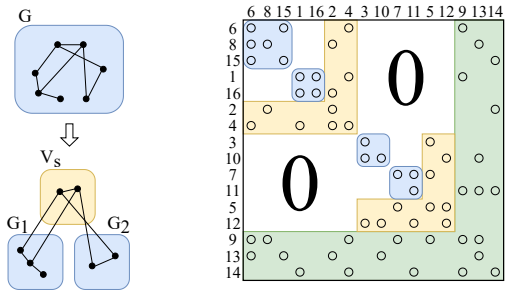


Figure 2: Graph dissection. (a) A graph G is dissected to two disjoint connected subgraphs G_1 and G_2 by a vertex separator V_S. (b) The permuted matrix based on nested dissection.

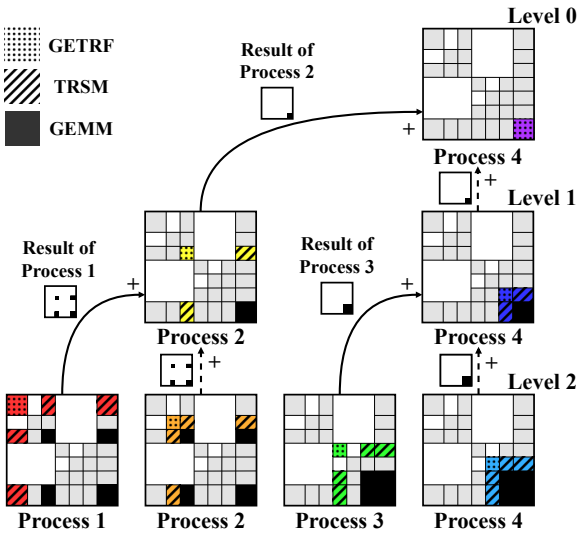


Figure 3: Parallelization and data communication of LU factorization based on nested dissection.

a vertex separator V_S, as shown in Figure 2(a). This recursive process minimizes the separator size while balancing the subgraphs, resulting in a hierarchical structure suitable for divide-and-conquer strategies.

In practice, nested dissection is typically performed using tools such as ParMETIS, which is designed to compute high-quality vertex separators efficiently. The effectiveness of nested dissection improves when the input matrix exhibits high sparsity, as the separator can more easily isolate large independent subgraphs. In this paper, we assume that high-quality partitioning is provided via ParMETIS.

Permuting matrices. The partitions from nested dissection are used to permute the sparse matrix, as illustrated in Figure 2(b). The permutation groups rows and columns corresponding to the vertex separator V_S at the end, effectively separating the matrix into independent submatrices. This structure reduces fill-ins and creates sparsity patterns that can be efficiently exploited during factorization. This property is commonly utilized in many open-source sparse solvers, such as SuperLU_DIST, STRUMPACK, and PanguLU.

In addition to reduced fill-ins, the permutation provides another key advantage. Since no edges exist between the disjoint connected submatrices, the corresponding off-diagonal submatrices in the permuted matrix become entirely zero. For example, in Figure 2(b), the upper-left 7×7 submatrix reflects a dissection where the vertex separator {2,4} separates {6,8,15} and {1,16}. There are no edges between these two sets, so the off-diagonal blocks corresponding to their interaction contain only zeros. When the nested dissection is applied recursively, many such zero submatrices appear throughout the matrix. Such a distributed sparsity pattern of distinguishable zero submatrices can be exploited in the factorization process to reduce computation and communication by skipping operations on zero blocks.

Factorization procedure by nested dissection. Factorization using nested dissection follows a hierarchical divide-and-conquer approach. The result of recursive nested dissection can be represented as a tree structure, where each node corresponds to a diagonal submatrix. This tree captures the dependence between submatrices during the factorization process.

At each tree level, submatrices can be independently factorized (i.e., panel factorization), while the Schur complement is computed to propagate and merge the results to upper levels. The detailed steps of this factorization procedure are illustrated in Figure 3. In the figure, the colored and black regions within each matrix represent the currently active computational regions. Each matrix operation corresponds to a specific step—`GETRF`, `TRSM`, or `GEMM`—being applied to a particular submatrix.

Since submatrices at the same level are independent and can be processed concurrently, Figure 3 further illustrates how different processes are assigned to handle individual submatrices. The process ID responsible for each submatrix is indicated below the corresponding matrix. Communication between processes occurs only when transitioning to the next level, and rather than requiring all-to-all communication, it is limited to pairs of dependent processes. This localized, pairwise communication strategy significantly reduces synchronization and communication overhead.

2.5 Existing Sparse Direct Solvers

Numerous efforts have been made to optimize sparse solvers, especially for distributed memory systems. MUMPS [1] is an early example that uses an asynchronous multifrontal

Table 1: Summary of existing sparse direct solvers.

Solver	Factorization method	Parallelization		Nested dissection
		for CPUs	for GPUs	
MUMPS [1]	Multifrontal	✓		✓
SuperLU_DIST [23]	Supernodal	✓		
STRUMPACK [35]	Multifrontal	✓		✓
SuperLU_DIST 3D [36]	Supernodal	✓	✓	✓
STRUMPACK GPU [15]	Multifrontal	✓	✓	✓
PanguLU [13]	Supernodal	✓	✓	✓
SnuSOLVER	Two-phase	✓	✓	✓

method, incorporating pivoting techniques for numerical robustness. SuperLU_DIST [23, 24, 36] reduces communication overhead significantly, making it scalable. STRUMPACK [35] combines sparse direct methods with matrix compression to exploit low-rank properties. PanguLU [13] introduces a novel approach to regularize matrix structures, improving computational efficiency and reducing memory overhead. However, many existing solvers are limited in their ability to fully utilize heterogeneous architectures, such as GPU-based cluster systems.

This paper addresses these limitations by proposing a new parallelization method with optimization techniques tailored for modern GPU-based heterogeneous systems. Table 1 summarizes the key features of various sparse solvers, highlighting the distinctions of the proposed method.

3 Motivation & Methodology

This section describes the motivation of SnuSOLVER and its overall methodology.

3.1 Limitations of Previous Approaches

Supernodal and frontal methods are widely used methods, but achieving scalability and efficiency in heterogeneous systems remains challenging due to two major limitations. One is that they focus on maximizing computational unit utilization through densification and optimized BLAS libraries. However, densification can introduce significant overhead for highly sparse submatrices. Our evaluation of SuperLU_DIST on a CPU using Intel's oneMKL reveals that, despite the theoretical maximum throughput of 70 GFLOP/sec, the achieved performance is less than 1 GFLOP/sec. This inefficiency arises because the supernodal method continues densification regardless of the submatrix size or structure. The resulting GFLOPS falls far below the theoretical performance of a single core, highlighting the limitations of densification in handling sparse submatrices effectively. Moreover, the frequent format conversions between sparse and dense matrices exacerbate these issues, further increasing computational costs.

The other is that their parallelization often suffers from substantial communication and synchronization overhead.

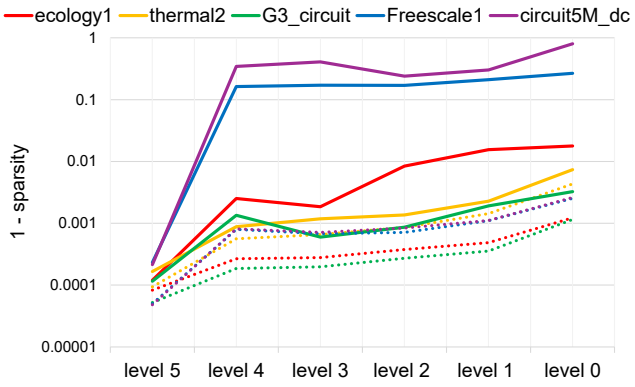


Figure 4: The proportion of nonzero elements in the submatrices at each level for some sparse matrices. Solid lines show values after factorization, while dotted lines represent values before factorization.

The uneven distribution of workloads leads to frequent delays and inefficiencies. For example, our analysis shows that computation accounted for less than 15% of the total execution time in SuperLU_DIST, with the majority of time spent on synchronization and communication.

3.2 Observations

Nested Dissection (ND) provides a framework to address the above limitations by enabling hierarchical partitioning and parallelization. Our observations reveal its untapped potential for scalability and performance optimization. Submatrices at different levels of the nested dissection tree exhibit varying sparsity patterns, enabling adaptive optimization. By selectively applying sparse or dense kernels based on the sparsity of each submatrix, performance can be significantly improved. As shown in Figure 4, the sparsity pattern differs across levels. Dotted lines indicate sparsity before factorization, while solid lines represent the pattern after factorization. Factorization results in a much denser structure, highlighting opportunities for targeted optimization.

Extending nested dissection to deeper levels allows for finer partitioning of workloads by balancing the sizes of submatrices. Although this approach has traditionally been considered counterproductive due to communication overhead, our methodology demonstrates that these challenges can be mitigated through efficient memory layouts. Previous approaches, such as SuperLU_DIST, have incorporated the properties of nested dissection (ND), they often fell short of fully exploiting its hierarchical structure. The SuperLU_DIST 3D [36] asserts that the advantages of nested dissection are most effective when the matrix is divided into up to four submatrices (i.e., level 2). This limitation stems from the inherent constraints of the supernodal method, which restricts the number of submatrices that can be effectively processed.

In contrast, our approach overcomes these limitations by proposing a new computational method, enabling better utilization of the benefits of ND. This motivates our approach to redesign ND-based parallelization, allowing for enhanced scalability and performance.

3.3 Methodology

SnuSOLVER employs a two-phase execution strategy within an ND-based framework to optimize sparse direct solvers. Unlike traditional approaches such as the supernodal or frontal method, SnuSOLVER adopts a distinct factorization scheme, performing computations in a fundamentally different way. First, it partitions the matrix into submatrices based on ND, ensuring the number of leaf submatrices corresponds to the number of processes, with each process handling one leaf submatrix to maximize parallel efficiency (e.g., up to 5 levels for 32 processes). Then, it assigns submatrices at the lowest level to individual threads for sparse computation, minimizing synchronization overhead.

Two-phase execution. SnuSOLVER's two-phase execution method is described as follows. The first phase is the *sparse phase*. At the lowest levels of the tree, computations are performed using sparse kernels (`GETRF`, `TRSM`, and `GEMM`). This phase minimizes computational overhead by focusing on smaller-sized submatrices, processed using a straightforward implementation with a single thread to process each submatrix without leveraging parallelization or vector units. This approach avoids unnecessary overhead while maintaining simplicity for sparse computations.

The second phase is the *dense phase*. SnuSOLVER performs a transition to the dense phase for higher levels of the tree. At these levels, all data (or submatrices) are transformed into dense data structures resembling dense linear algebra workflows, such as High-Performance LINPACK (HPL) [10, 21, 32], rather than sparse matrix solvers. All kernels (`GETRF`, `TRSM`, and `GEMM`) are executed using optimized dense libraries, such as oneMKL [18], cuSOLVER [28], and cuBLAS [27]. Handling all computations within dense data structures enables seamless GPU offloading. This approach improves GPU utilization by reducing data transfer overhead and achieving higher computational throughput.

SnuSOLVER employs the two-phase execution strategy within an ND-based framework to optimize sparse direct solvers. This approach provides a structured methodology for organizing computations and defining their execution order, regardless of the underlying hardware (e.g., CPU or GPU). The benefits of this strategy are described as follows:

- **Reduced synchronization and communication:** The hierarchical structure of ND allows for effective synchronization, as shown in Table 4. Despite a high computational workload, the execution time is relatively small, resulting in higher GFLOPS. Its balanced workload distribution minimizes synchronization delays, enabling efficient parallel computation. It also eliminates unnecessary format conversions, reducing additional computational costs.
- **Proper usage of computation kernels:** It ensures optimal usage of computational resources for both sparse and dense regions. Sparse kernels are selected based on sparsity patterns to maximize throughput. For regions with high sparsity, dense kernels are avoided to reduce unnecessary computation. For regions with low sparsity, dense matrix processing is leveraged to achieve higher throughput.
- **Efficient GPU offloading:** GPU offloading is particularly effective due to the continuous computation patterns in higher levels of the ND tree. By performing consecutive operations (all `GETRF`, `TRSM`, and `GEMM`) directly on the GPU, redundant data transfer is minimized, further improving performance.

4 Design and Implementation

This section delves into the design and implementation of our approach. Our optimization target is the factorization and solve steps. The overall process follows the binary tree structure depicted in Figure 3. Our target environment is a GPU-based cluster, in which each node is equipped with multi-core CPUs and multiple GPUs. We implement the sparse LU solver using single-threaded multiple MPI processes. The libraries used include OpenMPI [16] for message passing, Intel's oneMKL [18] for BLAS operations on the CPUs, and CUDA [27–29] for the GPUs.

4.1 Parallelization for Multi-core CPUs

In our approach, the matrix is well partitioned into submatrices by ND. Submatrices at the same level are independent, so it is beneficial for different MPI processes to handle them to the maximum extent allowed by the number of available processes. While partitioning the given matrix into more submatrices than the number of processes may introduce more parallelism, it may incur unnecessary parallelization overhead for each process to handle excessive submatrices. On the other hand, partitioning the given matrix into fewer submatrices than the number of processes may not fully utilize parallelism provided by the underlying system. Thus, we finish the nested dissection when the number of submatrices in the final level matches the number of processes, assuming one-to-one mapping between MPI processes and physical CPU cores.

Distributing the workload. ND is inherently optimized for balanced subgraph (i.e., submatrix) sizes. Since submatrices are of similar sizes, our process scheduling method ensures that the execution time of each process is approximately identical, mitigating the workload imbalance. We make the process that handles a child node to handle its parent node, enabling it to halve the data transferred to the process for the parent. Figure 3 illustrates how processes divide the workload. A different process processes each diagonal submatrix except the vertex separator at Level 2. Each diagonal submatrix at Level 1 is handled by one of the processes that processed its children; in this case, they are handled by Process 2 and Process 4. Similarly, the single submatrix at Level 0 is handled by the process that handled one of its children, which in this case is Process 4.

4.2 Kernel Selection

The three primary operations—`GETRF`, `TRSM`, and `GEMM` —can be performed using highly optimized computation kernels tailored for either sparse or dense matrix formats on CPUs. Dense matrices are particularly well-suited for GPUs, as they can fully leverage the GPU's massively parallel architecture. In contrast, sparse matrices, due to their irregular distribution of nonzero elements, are less computationally intensive and often suffer from low resource utilization on GPUs, making them less efficient. As a result, CPUs are typically more appropriate for processing sparse matrices than GPUs.

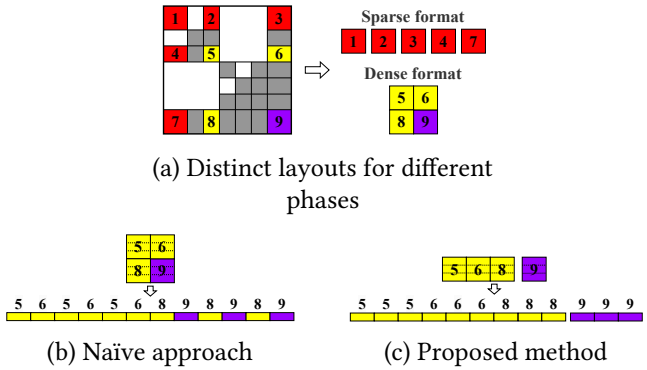
While it is possible to convert a sparse matrix into a dense matrix format to benefit from GPU acceleration, this conversion increases the number of operations, as previously zero elements must now be included in the computation. Therefore, selecting the appropriate matrix format—whether sparse or dense—and processor type (CPU or GPU) is essential for maximizing factorization performance.

As sparse matrices undergo nested dissection, their sparsity generally decreases progressively at higher levels. This reduction is caused by the nature of the matrix structure and the fill-ins introduced during factorization, as submatrices are aggregated and passed upward. Figure 4 illustrates this trend for several large, sparse graphs used in our evaluation, which were processed using nested dissection to produce a binary tree of height five (up to level 5). The increasing density at higher levels highlights the benefit of using sparse kernels at the lower levels and switching to dense kernels for the higher levels, where the matrices become significantly denser.

PanguLU [13] takes a similar approach by implementing various computation kernels and dynamically selecting the most appropriate kernel for each submatrix via a decision tree. However, this approach incurs significant overhead due to matrix format conversions. In contrast, SnuSOLVER avoids the complexity and overhead of dynamic kernel selection by structurally determining the appropriate kernel for each level. This *static* kernel selection makes the computation flow simpler and more predictable, enabling more efficient management of the entire computation process, including operation order, synchronization, and communication. Additionally, it eliminates the need for data format conversions, reducing overhead. Table 2 summarizes the kernel implementations used in SnuSOLVER.

Table 2: Summary of kernel implementations.

Kernel	Sparse	Dense (CPU)	Dense (GPU)
GETRF	Custom implementation	oneMKL [18]	cuSOLVER [28]
TRSM			
GEMM			cuBLAS [27]



(a) Distinct layouts for different phases

(b) Naïve approach (c) Proposed method

Figure 5: Memory layout for storing submatrices.

4.3 Matrix Formats for Sparse phase

We store the submatrices at the lowest level in a sparse format, while those at higher levels are stored in a dense format (note that level 0 is the highest level). This approach is based on using appropriate kernels according to the sparsity of submatrices. Our observations indicate that submatrices become denser as the level goes up. At the lowest level, we maintain the sparse format and perform `GETRF` and `TRSM`. When executing `GEMM`, the output of `GEMM` is converted to a dense format and passed to the upper level. For all subsequent levels, the operations of the `GETRF`, `TRSM`, and `GEMM` continue in the dense format.

Dual-sparse formats per submatrix. Sparse matrices are typically stored using the Compressed Sparse Row (CSR) format. While CSR is efficient for full matrix operations, it is less suited for scenarios where submatrices need to be frequently extracted or manipulated. This limitation arises because the pointer-based structure of CSR complicates the extraction and management of submatrices, especially when these submatrices are stored in non-contiguous memory

regions. To address this issue, we store each submatrix independently in the CSR format. In addition, we introduce the Compressed Sparse Column (CSC) format as well, storing indices in both formats. The CSR format facilitates efficient row-based iteration, while the CSC format excels in column-based iteration. This dual format approach increases memory usage by approximately 1.5 times compared to using a single CSR or CSC format, presenting a trade-off between memory efficiency and computational speed. However, in the context of sparse matrices, where speed is often prioritized over memory usage, this trade-off is generally acceptable. Moreover, transitioning between formats incurs linear time complexity in relation to the number of non-zero elements (nnz). This overhead is negligible compared to the square or cubic time complexities associated with GEMM operations and similar computations.

4.4 Memory Layouts for Dense phase

To manage dense submatrices efficiently, we merge the dense submatrices for a process together and allocate a single memory space for them, as shown in Figure 5(a). It improves cache locality. However, simply storing the merged matrix naïvely is inefficient because the actual space allocation is one-dimensional, leading to parts of a single submatrix being stored non-contiguously, as shown in Figure 5(b).

To address this problem, we arrange the 2D submatrices contiguously and then allocate the memory space of the rows across the submatrices accessed by TRSM (submatrices 5, 6, and 8) in a contiguous memory block as shown in Figure 5(c). This approach further reduces memory usage and provides better access patterns. It avoids cache pollution caused by accessing unnecessary data. For example, submatrices 5, 6, 8, and 9 are accessed to save the result of GEMM. The naïve approach may access a part of the submatrix 9 when accessing the submatrix 8 during GEMM. In addition, it minimizes communication overhead. For example, when the submatrix 9 needs to be transferred from Level 1 to Level 0, having the rows of the submatrix stored contiguously, as shown in Figure 5(c), simplifies the data transfer process, reducing the overhead.

4.5 Communication and Computation Overlapping

During the factorization process, it is possible to hide most communication time within the computation time. In Figure 3, the black submatrix for Process 1 represents the result of GEMM that needs to be transferred to Process 2 to further process the submatrices at Level 1. Four independent GEMM must be performed at Level 2. As soon as the GEMM operation on each submatrix completes, the data is sent to the appropriate process at Level 1 (i.e., Process 2 or Process 4), performing the next GEMM operation to overlap computation and communication.

4.6 Offloading to GPUs

While approaches like SuperLU_DIST [37] only offload GEMM to the GPUs, PanguLU [13] implements various kernels for GETRF, TRSM, and GEMM to execute them on GPUs depending on the size and sparsity of submatrices using a decision tree. SnuSOLVER differs in that it offloads not just a kernel but the entire dense phase as a consecutive procedure to GPUs. Memory space allocation is performed in the same manner as for the CPU.

Number of processes. For factorization on CPUs, we make the number of MPI processes equal to the number of CPU cores. However, the target GPU system typically has fewer GPUs than CPU cores. Thus, we distribute the GPUs evenly across the processes. Ideally, each GPU should handle an equal number of processes. In addition, assigning processes that frequently communicate with each other to the same GPU will significantly reduce the communication overhead. Thus, we prioritize allocating processes closer in the tree (in Figure 3) to the same GPU, minimizing communication between GPUs.

GPU offloading. For computational efficiency, the lowest-level submatrices processed by sparse kernels on the CPUs are not offloaded to the GPUs, as their high sparsity makes them unsuitable for GPU acceleration. However, the computations for submatrices at all subsequent levels are offloaded to the GPUs. When transferring submatrices to the GPUs, we keep them in their original sparse format to minimize communication overhead. The conversion to dense format is performed on the GPUs.

Submatrices at the lowest level are processed using GETRF and TRSM in the dual-sparse formats on the CPUs. The submatrices from TRSM stored in the sparse format are transferred to the GPUs. Then, the sparse GEMM kernel is invoked on the GPUs. Since their sparsity is high enough, using the sparse GEMM kernel is more efficient than using the dense GEMM kernel. At the subsequent levels, we use GETRF, TRSM, and GEMM kernels provided by cuBLAS and cuSolver on the GPUs.

Memory layouts. We also use the memory layout shown in Figure 5(c) for the dense submatrices on the GPUs to facilitate coalesced memory accesses. For communication between GPUs, we employ the GPUDirect [29], which enables efficient data transfer by bypassing the CPU. Unlike other LU factorization libraries that repeatedly offload only a specific kernel to the GPU, we offload operations at all consecutive levels to the GPUs, but the lowest level GETRF and TRSM, which can fully leverage the GPUDirect mechanism.

Table 3: Test matrices used in the experiments.

Name	N	nnz	nnz/N	Problem Domain	Usage in Prior Work
rajat25	87,190	606,489	7.0	Circuit Simulation	KLU [7]
ASIC_100ks	99,190	578,890	5.8	Circuit Simulation	KLU [7]
thermomech_TC	102,158	711,558	7.0	Thermal	ParILUT [2]
lung2	109,460	492,564	4.5	CFD	Yilmaz et al. [39]
torso2	115,967	1,033,473	8.9	2D/3D	Yilmaz et al. [39]
dc2	116,835	766,396	6.6	Subsequent Circuit Simulation	Dufrechou et al. [12]
twotone	120,750	1,206,265	10.0	Frequency Domain Circuit Simulation	SuperLU_DIST [23]
xenon2	157,464	3,866,688	24.6	Materials	STRUMPACK [15]
c-73	169,422	1,279,274	7.6	Optimization Problem Sequence	STRUMPACK [15]
scircuit	170,998	958,936	5.6	Circuit Simulation	STRUMPACK [15]
ohne2	181,343	6,869,939	37.9	Semiconductor Device	STRUMPACK [15]
hvdc2	189,860	1,339,638	7.1	Power Network	Manguoglu et al. [26]
thermomech_dM	204,316	1,423,116	7.0	Thermal	ParILUT [2]
thermomech_dK	204,316	2,846,228	13.9	Thermal	Manguoglu et al. [26]
ss1	205,282	845,089	4.1	Semiconductor Process	Petrushov et al. [33]
HTC_336_4438	226,340	783,496	3.5	Power Network	Li et al. [22]
Raj1	263,743	1,300,261	4.9	Circuit Simulation	GLU [17, 31]
ASIC_320ks	321,671	1,316,085	4.1	Circuit Simulation	GLU [17, 31]
rajat24	358,172	1,946,979	5.4	Circuit Simulation	KLU [7]
parabolic_fem	525,825	3,674,625	7.0	CFD	Petrushov et al. [33]
ASIC_680ks	682,712	1,693,767	2.5	Circuit Simulation	GLU [17, 31]
tmt_sym	726,713	5,080,961	7.0	Electromagnetics	ParILUT [2]
tmt_unsym	917,825	4,584,801	5.0	Electromagnetics	GLU [17]
ecology1	1,000,000	4,996,000	5.0	2D/3D	SuperLU_DIST [36], PanguLU [13]
webbase-1M	1,000,005	3,105,536	3.1	Weighted Digraph	Dufrechou et al. [12]
thermal2	1,228,045	8,580,313	7.0	Thermal Problem	GLU [17]
G3_circuit	1,585,478	7,660,826	4.8	Circuit Simulation	SuperLU_DIST [36], PanguLU [13]
memchip	2,707,524	13,343,948	4.9	Circuit Simulation	STRUMPACK [15]
Freescale1	3,428,755	17,052,626	5.0	Circuit Simulation	STRUMPACK [15]
circuit5M_dc	3,523,317	14,865,409	4.2	Circuit Simulation	Wang et al. [38]
rajat31	4,690,002	20,316,253	4.3	Circuit Simulation	Manguoglu et al. [26]

Table 4: Performance comparison of numerical factorization between SnuSOLVER and SuperLU_DIST on `ASIC_680ks` matrix using a single 32-core CPU (2.35GHz with AVX512 vector unit). The table presents detailed statistics of GEMM kernels and total computations across 32 CPU cores. In the GEMM computation section (GEMM), the number of operations (GFLOPs), the number of GEMM kernel invocations (Number), the execution time (Time), and the performance (GFLOP/sec) are shown. In the factorization section, the total number of operations (GFLOPs), the computation time (Compute), the total factorization time (Total), and the performance (GFLOP/sec) are presented. Additionally, the number of MPI Send/Recv calls (MPI call) is included to indicate the reduction in synchronization overhead achieved by SnuSOLVER.

		GEMM				Factorization				
		GFLOPs	Number	Time (sec)	GFLOP/sec	GFLOPs	Compute (sec)	Total (sec)	GFLOP/sec	MPI call
SuperLU_DIST		0.07	66,160	0.077	0.85	1.65	0.311	2.292	0.72	3,100,920
SnuSOLVER	Sparse	-	-	-	-	0.23	0.024	0.024	9.40	-
	Dense	4.23	541	0.067	62.85	4.35	0.043	0.088	49.61	-
	Total	-	-	-	-	4.58	0.067	0.116	39.62	374

After completing all computations, the final L and U will be stored on the GPUs where the solve phase will be performed.

4.7 Solve Phase

After factorization (i.e., $A = LU$), we must perform computations to solve the target equation $Ax = LUx = b$. This is done in two separate steps, solving $Ly = b$ and $Ux = y$. Solving $Ly = b$ is similar to factorization since it also finds U, such that $LU = A$. Removing `GETRF` and reducing a matrix operation to a vector operation will lead to solving the equation. However, solving $Uy = b$ is different because matrix multiplication is not commutable, and U's position changes from right to left compared to $LU = A$. The solve process for L progresses from bottom to top, propagating the results of `GEMV` to the higher levels. Conversely, the solve process for U progresses from higher to lower levels, delivering the results of `GEMV`. Other aspects of the overall process proceed similarly to factorization. The solve phase accounts for a

Table 5: Configuration of a node in the eight-node cluster.

CPU	1 × AMD EPYC 7452 32-core 2.35GHz
Main Memory	8 × DDR4-2666 64GB
GPU	4 × NVIDIA Tesla V100 32GB PCIe
NIC	Mellanox ConnectX-6 Infiniband HDR
OS	Ubuntu 20.04.6 LTS (Kernel 5.4.0-100)
GPU Driver	550.54.15 (CUDA 12.4)

relatively small portion of execution time compared to factorization. Since the factorization results, L and U, are not required to be transferred back to the CPU, only the communication of the target vector b and the solution vector x is necessary.

5 Evaluation

In this section, we evaluate the performance of SnuSOLVER by comparing it with existing methods on a GPU cluster.

5.1 Experimental Setup

Test matrices. Table 3 shows our test sparse matrices from various application domains. They are all obtained from the SuiteSparse Matrix Collection [6], formerly known as the University of Florida Sparse Matrix Collection, a widely used publicly available real-world sparse matrix benchmark suite [1, 13, 15, 23, 36]. The test set includes matrices of various sizes (N, the number of rows/columns), sparsity levels (nnz/N, the average number of nonzero elements per row/column), and types: symmetric, asymmetric, and symmetric positive definite. As indicated in Table 3, all the test matrices are used in previous approaches, which we compare with the proposed method.

Preprocessing the matrices. To ensure a fair end-to-end performance comparison of the method with existing approaches, we preprocess all the test matrices with identical scaling and maximum weight matching before conducting the experiments. In addition, we use ParMETIS [20] (or METIS [19]) to undergo the same fill-in reducing reordering process, which is the default option of SuperLU_DIST and STRUMPACK as well. In this manner, we ensure an equivalent experimental setup across all approaches to compare.

Comparison baselines. We set SuperLU_DIST (v8.2.1) and STRUMPACK (v7.1.0) as comparison baselines. SuperLU_DIST and STRUMPACK are state-of-the-art GPU-accelerated sparse direct solvers targeted for large-scale distributed memory parallel systems [15, 23]. For SuperLU_DIST, we choose an algorithm between 2D and 3D versions that performs better for each matrix as its baseline for comparison. The total number of threads, defined as the product of the number of MPI processes and the number of OpenMP threads, is kept consistent across all baselines in the experiments for fair comparison. After trying various combinations of the number of MPI processes and the number of OpenMP threads for all the test matrices in each baseline, we report the best performance result.

System configurations. We evaluate the proposed method using a cluster of eight nodes, each node equipped with a single AMD EPYC 7452 32-core CPU and four NVIDIA Tesla V100 32GB GPUs connected through PCIe. The details of our node configuration are shown in Table 5. The implementations, including all the baselines, are configured with gcc-9.4.0, CMake-3.26.3, OpenMPI-4.1.6, OpenMP-4.5, METIS-5.1.0, and ParMETIS-4.0.3. CUDA 12.4 is used with the NVIDIA driver version 550.54.15. Other necessary software packages varying from baseline to baseline are installed faithfully following the respective manuals.

5.2 Single Matrix Case Study

To illustrate the advantages of the proposed two-phase methodology, we present a performance comparison between SnuSOLVER and SuperLU_DIST for the `ASIC_680ks` matrix. Although this case study focuses on a single matrix, the performance trends observed here are consistent across diverse matrices, demonstrating the robustness of SnuSOLVER.

The results in Table 4 highlight two key distinctions between the two methods. First, SnuSOLVER significantly reduces the number of GEMM calls compared to SuperLU_DIST. While the supernodal method in SuperLU_DIST generates numerous kernel calls for small dense submatrices, leading to inefficient resource utilization, SnuSOLVER's two-phase strategy boosts performance. The sparse phase minimizes computation at the lower levels of the elimination tree by reducing unnecessary operations, while the dense phase efficiently handles higher levels where submatrices become denser. By using sparse kernels for sparse regions and large dense kernels for denser regions, SnuSOLVER avoids unnecessary fragmentation of computations and excessive kernel calls, thereby improving throughput and computational efficiency despite involving more total operations.

Second, the methods differ notably in terms of overhead. In SuperLU_DIST, computation accounts for only a small portion of the total execution time due to synchronization, communication overhead, and frequent format transitions. By contrast, SnuSOLVER 's two-phase execution strategy minimizes these overheads, allowing most of the total execution time to be spent on computation. This is further supported by the comparison of MPI Send/Recv call counts shown in Table 4. Since SnuSOLVER performs no additional

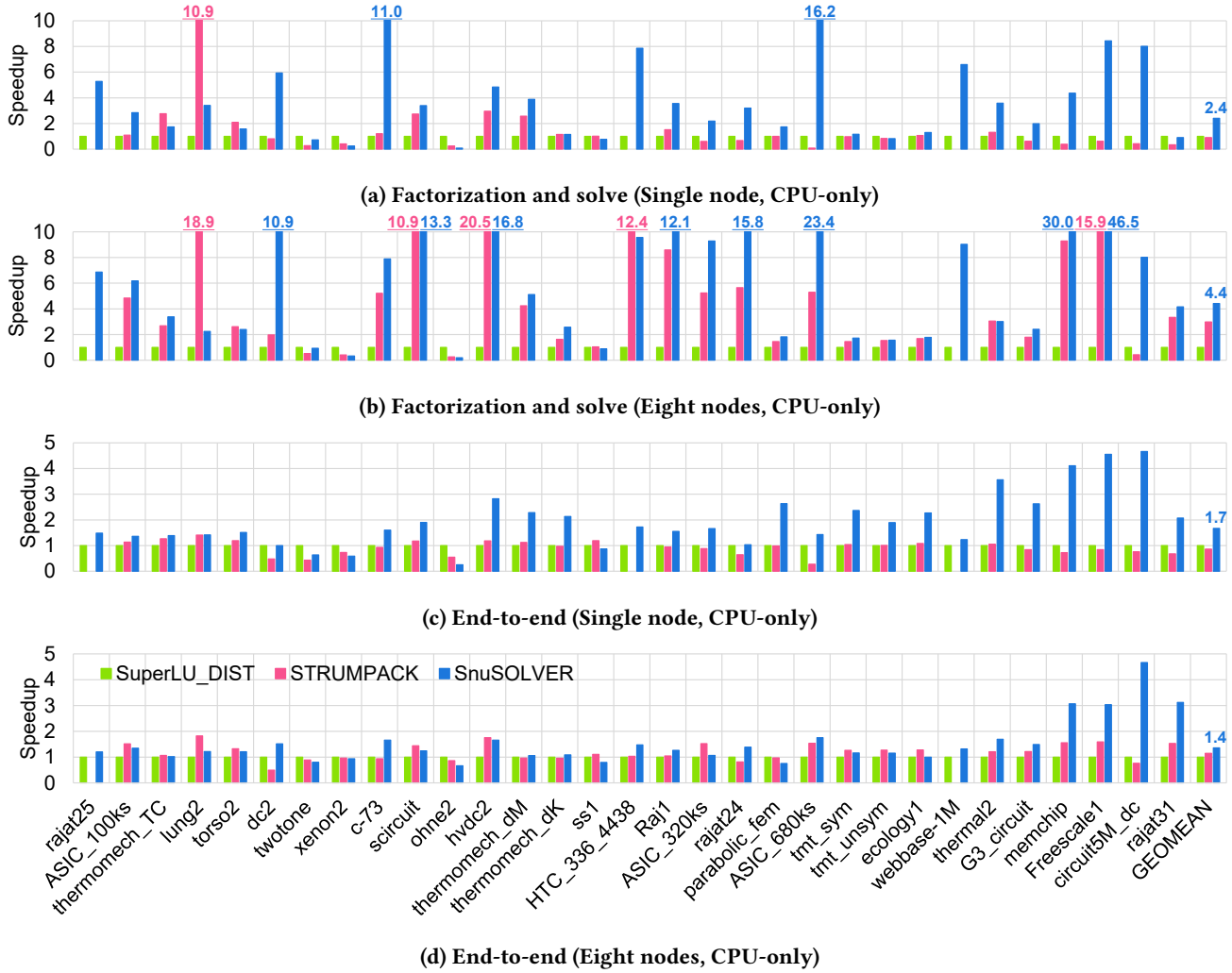


Figure 6: Performance on CPUs. The test matrices are arranged in an ascending order of size from left to right. STRUMPACK fails factorization on `rajat25`, `HTC_336_4438`, and `webbase-1M` matrices.

synchronization or communication outside of MPI Send/Recv during the factorization phase, we use these counts as a representative metric for overhead. While SuperLU_DIST incurs a high number of MPI function calls throughout execution, the significantly lower number of MPI calls in SnuSOLVER highlights its efficiency in reducing communication and synchronization costs. This approach reduces computational costs and outperforms traditional methods like the supernodal, which applies a uniform strategy across all levels.

SnuSOLVER achieves an efficiency of 39.62 GFLOP/s, significantly outperforming SuperLU_DIST's 0.72 GFLOP/s. These results underscore the effectiveness of the two-phase method in addressing key inefficiencies inherent in the traditional methods (i.e., supernodal or frontal method).

Advantage of dual-sparse format. We also evaluate the impact of the dual-sparse format on performance. For the matrix `ASIC_680ks`, the sparse phase execution time with the CSR format improves from 0.36 seconds to 0.24 seconds with the dual-sparse format, representing a 1.5× performance improvement. A geometric mean of performance improvement across all the test matrices is 1.15×, demonstrating consistent speedup of the dual-sparse format.

5.3 Effect of Memory Layout

Table 6 summarizes the performance of dense submatrix operations, measured using MKL GEMM, CUDA GEMM, and MPI Send/Recv, across various matrix sizes on our experimental heterogeneous system (comprising AMD CPUs and NVIDIA GPUs). For the MPI Send/Recv comparison, we used

Table 6: Effect of memory layout.

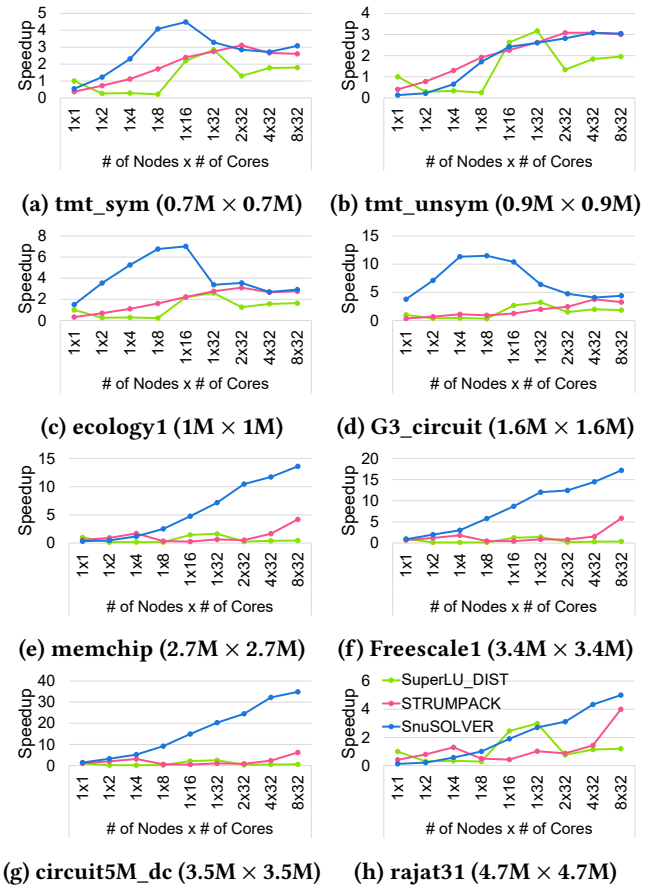
Operation	Layout	Matrix size (N×N)		
		512	1024	2048
MKL GEMM (GFLOPS)	Naïve	81.99	86.72	90.86
	Proposed	85.23	89.17	91.09
	Speedup over Naïve	1.04	1.03	1.00
CUDA GEMM (TFLOPS)	Naïve	2.78	4.55	5.60
	Proposed	2.75	4.96	5.60
	Speedup over Naïve	0.99	1.09	1.00
MPI Send/Recv (GB/s)	Naïve	0.64	1.05	1.03
	Proposed	0.86	2.00	1.84
	Speedup over Naïve	1.34	1.90	1.79

MPI APIs that support non-contiguous memory communication in the discontinuous layout to ensure fairness. While the total data volume remained the same in both cases, the performance differences stem solely from the memory layout. The results demonstrate that SnuSOLVER's contiguous memory layout improves computational efficiency and reduces communication overhead compared to the naïve discontinuous layout. This improvement results from enhanced cache locality and simplified data access patterns, leading to higher throughput for both computation and communication.

5.4 Performance on CPUs

We evaluate and compare the numerical factorization and the end-to-end performance on multi-core CPUs.

Factorization and solve. Figure 6(a) and 6(b) show the speedup of SnuSOLVER and STRUMPACK over SuperLU_DIST in numerical factorization and solve phase on the single node system and eight-node cluster. The numerical factorization phase is the most compute-intensive part of a sparse linear solver. The solve phase's execution time is relatively small (approximately 10%) compared to the factorization phase. The performance of SnuSOLVER significantly outperforms both SuperLU_DIST and STRUMPACK as shown in Figure 6(a) and Figure 6(b). On the single node system with 32 CPU cores, the average speedup of SnuSOLVER over SuperLU_DIST in numerical factorization and solve is 2.40, reaching up to 16.21. The average speedup on the eight-node cluster with 256 CPU cores increases to 4.42, with a maximum of 46.51. Similarly, the average speedup over STRUMPACK is 2.40 on the single node and 1.42 on the eight-node cluster. For `xenon2` and `ohne2`, SnuSOLVER is worse than SuperLU_DIST due to the relatively high density of these matrices. Such relatively dense matrices typically result in larger vertex separators during nested dissection [34]. SnuSOLVER, which performs sparse computations only at the lowest level while handling the rest with the dense format, becomes less efficient when the size of the vertex separator is large. However, as SnuSOLVER scales effectively, the performance gap between SnuSOLVER and SuperLU_DIST on those two matrices narrows down when running them on the 8-node cluster.



(a) tmt_sym (0.7M × 0.7M) (b) tmt_unsym (0.9M × 0.9M)

(c) ecology1 (1M × 1M) (d) G3_circuit (1.6M × 1.6M)

(e) memchip (2.7M × 2.7M) (f) Freescale1 (3.4M × 3.4M)

(g) circuit5M_dc (3.5M × 3.5M) (h) rajat31 (4.7M × 4.7M)

Figure 7: Scalability of the numerical factorization and solve on multi-core CPUs. The x-axis shows the number of CPU cores. For example, 8×32 denotes a total of 256 CPU cores from eight nodes (single 32-core CPU per node) participating in parallel computation.

End-to-end time. The end-to-end time refers to the total execution time of the solver, including analysis, symbolic factorization, numerical factorization, and solve phases, excluding the time spent on scaling and maximum weight matching, which are the preprocessing steps for numerical stability. Despite being a crucial performance comparison metric, the end-to-end execution time of the solver has not been showcased as an evaluation result in the previous studies [13, 15, 23].

To this end, we also evaluate the end-to-end performance of the solvers. Figure 6(c) and 6(d) summarize the end-to-end performance comparison results. The results show that SnuSOLVER achieves average speedups of 1.67 and 1.96

on the single node over SuperLU_DIST and STRUMPACK, respectively. In addition, average speedups of 1.35 over SuperLU_DIST and 1.18 over STRUMPACK are observed on the 8-node cluster.

The end-to-end speedups are lower than those observed in the numerical factorization phase because the analysis step—performed identically across all solvers using an external library (ParMETIS)— accounts for a significant portion of the total execution time, sometimes more than half. Since our optimizations focus exclusively on the factorization and solve phases (with solve being negligible), the overall benefit is naturally bounded by the cost of the analysis phase (i.e., ParMETIS). Furthermore, the smaller speedups observed in end-to-end comparison for multi-node settings compared to single-node configurations are primarily due to fixed costs such as process spawning, GPU initialization, and analysis time, which remain constant regardless of the number of nodes. As the factorization time decreases with more nodes, these fixed components become more dominant, limiting the scalability visible in the end-to-end measurements.

The speedup of SnuSOLVER mainly comes from avoiding unnecessary matrix conversions, using optimal computation kernels, and using efficient memory layouts. It is particularly noteworthy that SnuSOLVER proves to be particularly effective for the most time-consuming, large-scale, high-sparsity matrices (e.g., `memchip`, `Freescale1`, `circuit5M_dc`, and `rajat31`), which are inherently difficult to process and compute, and are not well-handled by the previous approaches.

5.5 Scalability on CPUs

Figure 7 shows SnuSOLVER's scalability of the numerical factorization and solve phases on the 8-node cluster using up to 256 CPU cores. In this experiment, we intentionally select the eight largest matrices from our test set, with sizes (N, the number of rows/columns) ranging from 0.7M to 4.7M. We exclude `webbase-1M` because STRUMPACK fails to factorize it. We also leave out `Thermal2` as all baselines demonstrate poor scalability due to `Thermal2`'s low computational intensity.

We see that SnuSOLVER scales much better than baselines on multi-core CPUs. SnuSOLVER exhibits consistently better scalability in most cases, while others suffer from workload imbalance and communication overhead. The proposed method demonstrates significantly improved speedup on the eight nodes, especially for large matrices, such as `rajat31`. The behavior for matrices `tmt_sym`, `ecology1`, and `G3_circuit` appear unusual. The speedup decreases when the number of processes increases from 16 to 128 and then increases again on 256 processes. This is because relatively large vertex separators used by nested dissection for them make sparse `GEMM` more efficient than dense `GEMM` at higher levels. This issue could be solved by applying sparse `GEMM` at even higher levels for these specific matrices.

Table 7: Numerical factorization and solve's speedup comparison of GPU offloading on the eight-node cluster. SLD and STP refer to SuperLU_DIST and STRUMPACK, respectively.

Matrix	Factorization and solve on 32 GPUs					
	Over SLD on the GPUs			Over the solver itself on the CPUs		
	SLD	STP	SnuSOLVER	SLD	STP	SnuSOLVER
tmt_sym	1.00	1.43	3.07	0.92	0.90	1.64
tmt_unsym	1.00	1.64	3.18	0.95	1.01	1.64
ecology1	1.00	2.01	3.80	0.87	1.03	1.85
webbase-1M	1.00	Failure	17.50	1.07	Failure	2.07
thermal2	1.00	2.57	6.17	0.89	0.75	1.82
G3_circuit	1.00	3.55	6.79	0.96	1.90	2.71
memchip	1.00	6.98	32.21	0.97	0.73	1.04
Freescale1	1.00	10.52	42.39	0.98	0.74	0.89
circuit5M_dc	1.00	8.62	48.28	1.01	0.87	0.87
rajat31	1.00	4.03	12.25	0.99	1.20	2.91
GEOMEAN	**1.00**	**3.63**	**10.82**	**0.96**	**0.95**	**1.64**

Table 8: End-to-end speedup comparison of GPU offloading on the eight-node cluster. SLD and STP refer to SuperLU_DIST and STRUMPACK, respectively.

Matrix	End-to-end on 32 GPUs					
	Over SLD on the GPUs			Over the solver itself on the CPUs		
	SLD	STP	SnuSOLVER	SLD	STP	SnuSOLVER
tmt_sym	1.00	1.38	1.43	0.84	0.92	1.04
tmt_unsym	1.00	1.38	1.45	0.87	0.94	1.10
ecology1	1.00	1.62	1.28	0.82	1.04	1.05
webbase-1M	1.00	Failure	1.37	0.98	Failure	1.02
thermal2	1.00	1.35	2.08	0.87	0.96	1.07
G3_circuit	1.00	1.56	2.06	0.91	1.18	1.26
memchip	1.00	1.60	3.13	0.94	0.97	0.96
Freescale1	1.00	1.62	3.27	0.95	0.97	1.03
circuit5M_dc	1.00	1.61	3.98	0.98	0.98	1.10
rajat31	1.00	1.64	4.10	0.96	1.03	1.26
GEOMEAN	**1.00**	**1.52**	**2.20**	**0.91**	**1.00**	**1.08**

5.6 GPU Offloading

We evaluate the performance of SnuSOLVER when computations are offloaded to 32 GPUs on the target eight-node cluster. Similar to the scalability experiments on CPUs, we conduct GPU offloading experiments using the ten largest matrices from our test set, as larger matrices typically achieve higher GPU offloading efficiency. We observe that GPU offloading becomes more efficient when fewer nodes are used, as each GPU is tasked with handling a larger computational load. That is, fewer nodes show high GPU utilization.

Factorization and solve. In Table 7, SuperLU_DIST and STRUMPACK exhibit trivial or even worse performance improvements in their GPU versions compared to their corresponding CPU versions for some matrices (e.g., `tmt_sym`, `thermal2`, `memchip`). The CPU version represents solvers running on the eight nodes, each equipped with a single 32-core CPU. This is attributed to the overhead associated with communication to the GPUs, which offsets the reduction in factorization time.

However, SnuSOLVER achieves an average speedup of 1.64 over its CPU version. Moreover, SnuSOLVER outperforms SuperLU_DIST with an average speedup of 10.82 and STRUMPACK with 2.82. The speedup mainly comes from the better utilization of GPU resources by offloading consecutive procedures, reducing communication overhead through process placement, and minimizing data transfer overhead by leveraging sparse-to-dense conversion directly on the GPUs.

End-to-end execution time. Table 8 compares the end-to-end performance when offloading the workload to GPUs. SnuSOLVER achieves an average speedup of 2.20 over SuperLU_DIST and 1.52 over STRUMPACK. The speedup of SnuSOLVER's GPU version over its CPU version is 1.08, which is slightly higher than that of SuperLU_DIST (0.91) or STRUMPACK (1.00), but the difference is marginal. This is because the CPU version of SnuSOLVER is better optimized than that of SuperLU_DIST or STRUMPACK.

Indirect comparison with PanguLU. PanguLU [13] is excluded from the baseline for two reasons: One is that an end-to-end performance measurement for PanguLU cannot be performed because it saves the intermediate result to files before computation, causing a significant delay in the end-to-end execution time. The other is that it does not include the implementation of the solve phase, which presents a considerable challenge in verifying the correctness.

Although the experimental setups differ, we carry out an indirect performance comparison by examining the results reported in the literature using the same benchmark matrices (e.g., `ecology1` and `G3_circuit`) used in both experiments. According to the performance reported by PanguLU [13], it demonstrates 2–3× performance improvement in the numerical factorization step over SuperLU_DIST on `ecology1` and `G3_circuit` matrices on GPUs. PanguLU's target machine is a multi-node GPU cluster where each node has four NVIDIA A100 GPUs and two Intel Xeon 8180 2.5 GHz CPUs, similar to our system configuration. However, SnuSOLVER achieves a speedup of about 4–7 over SuperLU_DIST on the same matrices. This comparison highlights the superior performance of SnuSOLVER, even when considering the differences in the experimental setup.

6 Conclusion

This paper presents SnuSOLVER, a sparse direct LU solver optimized for GPU-based heterogeneous clusters, implementing the two-phase method based on nested dissection. By leveraging nested dissection and the hierarchical structure of partitioned subgraphs, SnuSOLVER addresses key limitations of the traditional supernodal and frontal methods. The two-phase method introduces optimizations, including balanced workload distribution across MPI processes, memory layouts optimized for cache and communication efficiency, and efficient overlapping of computation and communication. Our evaluation demonstrates SnuSOLVER's consistent and significant performance gains of sparse matrix factorization on 31 test matrices. Specifically, in the numerical factorization and solve phases using 256 CPUs, SnuSOLVER achieves average speedups of 4.42× and 1.42× over SuperLU_DIST and STRUMPACK, respectively. Using 32 GPUs, it achieves average speedups of 10.82× and 2.82×, respectively. In terms of the end-to-end performance using 256 CPUs, SnuSOLVER achieves average speedups of 1.35 and 1.18, and using 32 GPUs, it achieves 2.20 and 1.52, respectively, highlighting its scalability and efficiency.

Acknowledgments

This work was partially supported by the National Research Foundation of Korea (NRF) under Grant No. RS-2023-00222663 (Center for Optimizing Hyperscale AI Models and Platforms), and by the Institute for Information and Communications Technology Promotion (IITP) under Grant No. 2018-0-00581 (CUDA Programming Environment for FPGA Clusters) and No. RS-2025-02304554 (Efficient and Scalable Framework for AI Heterogeneous Cluster Systems), all funded by the Ministry of Science and ICT (MSIT) of Korea. Additional support was provided by the BK21 Plus Program for Innovative Data Science Talent Education (Department of Data Science, SNU, No. 5199990914569) and the BK21 FOUR Program for Intelligent Computing (Department of Computer Science and Engineering, SNU, No. 4199990214639), both funded by the Ministry of Education (MOE) of Korea. This work was also partially supported by Samsung Display Co., Ltd. and the Artificial Intelligence Industrial Convergence Cluster Development Project, funded by the MSIT and Gwangju Metropolitan City. Research facilities were provided by ICT at Seoul National University.

References

[1] Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. 2000. MUMPS: a general purpose distributed memory sparse solver. In *International Workshop on Applied Parallel Computing*. Springer, 121–130.

[2] Hartwig Anzt, Edmond Chow, and Jack Dongarra. 2018. ParILUT—A New Parallel Threshold ILU Factorization. *SIAM Journal on Scientific Computing* 40, 4 (2018), C503–C519. https://doi.org/10.1137/16M1079506

[3] C. Cleveland Ashcraft, Roger G. Grimes, John Gregg Lewis, Barry W. Peyton, Horst D. Simon, and Petter E. Bjørstad. 1987. Progress in Sparse Matrix Methods for Large Linear Systems On Vector Supercomputers. *International Journal of High Performance Computing Applications* 1 (1987), 10 – 30. https://api.semanticscholar.org/CorpusID:62698847

[4] Matthias Bollhöfer, Olaf Schenk, Radim Janalik, Steve Hamm, and Kiran Gullapalli. 2020. *State-of-the-Art Sparse Direct Solvers*. Springer International Publishing, Cham, 3–33. https://doi.org/10.1007/978-3-030-43736-7_1

[5] Thang Nguyen Bui and Curt Jones. 1993. A Heuristic for Reducing Fill-In in Sparse Matrix Factorization. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, PP 1993, Norfolk, Virginia, USA, March 22-24, 1993*, Richard F. Sincovec, David E. Keyes, Michael R. Leuze, Linda R. Petzold, and Daniel A. Reed (Eds.). SIAM, 445–452.

[6] 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

[7] Timothy A. Davis and Ekanathan Palamadai Natarajan. 2010. Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems. *ACM Trans. Math. Softw.* 37, 3 (2010). https://doi.org/10.1145/1824801.1824814

[8] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. 1999. A Supernodal Approach to Sparse Partial Pivoting. *SIAM J. Matrix Anal. Appl.* 20, 3 (1999), 720–755. https://doi.org/10.1137/S0895479895291765 arXiv:https://doi.org/10.1137/S0895479895291765

[9] Inderjit S Dhillon, Yuqiang Guan, and Brian Kulis. 2007. Weighted graph cuts without eigenvectors a multilevel approach. *IEEE transactions on pattern analysis and machine intelligence* 29, 11 (2007), 1944–1957.

[10] Jack Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience* 15 (08 2003), 803–820. https://doi.org/10.1002/cpe.728

[11] I. S. Duff and J. K. Reid. 1983. The Multifrontal Solution of Indefinite Sparse Symmetric Linear. *ACM Trans. Math. Softw.* 9, 3 (sep 1983), 302–325. https://doi.org/10.1145/356044.356047

[12] Ernesto Dufrechou and Pablo Ezzatti. 2018. A New GPU Algorithm to Compute a Level Set-Based Analysis for the Parallel Solution of Sparse Triangular Systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 920–929. https://doi.org/10.1109/IPDPS.2018.00101

[13] Xu Fu, Bingbin Zhang, Tengcheng Wang, Wenhao Li, Yuechen Lu, Enxin Yi, Jianqi Zhao, Xiaohan Geng, Fangying Li, Jingwen Zhang, et al. 2023. PanguLU: A scalable regular two-dimensional block-cyclic sparse direct solver on distributed heterogeneous systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[14] Alan George. 1973. Nested Dissection of a Regular Finite Element Mesh. *SIAM J. Numer. Anal.* 10, 2 (1973), 345–363. https://doi.org/10.1137/0710032

[15] Pieter Ghysels and Ryan Synk. 2022. High performance sparse multifrontal solvers on modern GPUs. *Parallel Comput.* 110 (2022), 102897. https://doi.org/10.1016/j.parco.2022.102897

[16] William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press.

[17] Kai He, Sheldon X. D. Tan, Hai Wang, and Guoyong Shi. 2016. GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 3 (2016), 1140–1150. https://doi.org/10.1109/TVLSI.2015.2421287

[18] Intel. 2024. Intel OneAPI Math Kernel Library (MKL). https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html Accessed: 2024-07-30.

[19] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997

[20] George Karypis and Vipin Kumar. 1998. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *J. Parallel and Distrib. Comput.* 48, 1 (1998), 71–95. https://doi.org/10.1006/jpdc.1997.1403

[21] Jinpyo Kim, Hyungdal Kwon, Jintaek Kang, Jihwan Park, Seungwook Lee, and Jaejin Lee. 2022. SnuHPL: high performance LINPACK for heterogeneous GPUs. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–12.

[22] Ang Li, Radu Serban, and Dan Negrut. 2015. *A Hybrid GPU-CPU Parallel CM Reordering Algorithm for Bandwidth Reduction of Large Sparse Matrices*. Technical Report.

[23] Xiaoye S. Li and James W. Demmel. 2003. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Softw.* 29, 2 (jun 2003), 110–140. https://doi.org/10.1145/779359.779361

[24] Xiaoye S. Li, Paul Lin, Yang Liu, and Piyush Sao. 2023. Newly Released Capabilities in the Distributed-Memory SuperLU Sparse Direct Solver. *ACM Trans. Math. Softw.* 49, 1, Article 10 (mar 2023), 20 pages. https://doi.org/10.1145/3577197

[25] Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. 1979. Generalized Nested Dissection. *SIAM J. Numer. Anal.* 16, 2 (1979), 346–358. https://doi.org/10.1137/0716027

[26] Murat Manguoglu. 2011. A domain-decomposing parallel sparse linear system solver. *J. Comput. Appl. Math.* 236, 3 (2011), 319–325. https://doi.org/10.1016/j.cam.2011.07.017 Aspects of Numerical Algorithms, Parallelization and Applications.

[27] NVIDIA. 2024. cuBLAS. https://developer.nvidia.com/cublas Accessed: 2024-07-30.

[28] NVIDIA. 2024. cuSOLVER. https://developer.nvidia.com/cusolver Accessed: 2024-07-30.

[29] NVIDIA. 2024. GPUDirect. https://developer.nvidia.com/gpudirect Accessed: 2024-07-30.

[30] Richard Peng and Santosh Vempala. 2021. Solving sparse linear systems faster than matrix multiplication. In *Proceedings of the 2021 ACM-SIAM symposium on discrete algorithms (SODA)*. SIAM, 504–521.

[31] 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. https://doi.org/10.1109/MDAT.2020.2974910

[32] Antoine Petitet, R. Clint Whaley, Jack J. Dongarra, and Andy Cleary. 2004. HPL-A portable implementation of the high-performance Linpack benchmark for distributed-memory computers. *http://www.netlib.org/benchmark/hpl/* (2004).

[33] Andrey Petrushov and Boris Krasnopolsky. 2023. Automated tuning for the parameters of linear solvers. *J. Comput. Phys.* 494 (2023), 112533.

https://doi.org/10.1016/j.jcp.2023.112533

[34] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. 1990. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM J. Matrix Anal. Appl.* 11, 3 (1990), 430–452. https://doi.org/10.1137/0611030

[35] François-Henry Rouet, Xiaoye S Li, Pieter Ghysels, and Artem Napov. 2016. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Transactions on Mathematical Software (TOMS)* 42, 4 (2016), 1–35.

[36] Piyush Sao, Xiaoye Sherry Li, and Richard Vuduc. 2018. A communication-avoiding 3D LU factorization algorithm for sparse matrices. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 908–919.

[37] Piyush Sao, Richard Vuduc, and Xiaoye Sherry Li. 2014. A Distributed CPU-GPU Sparse Direct Solver. In *Euro-Par 2014 Parallel Processing*, Fernando Silva, Inês Dutra, and Vítor Santos Costa (Eds.). Springer International Publishing, Cham, 487–498.

[38] Tengcheng Wang, Wenhao Li, Haojie Pei, Yuying Sun, Zhou Jin, and Weifeng Liu. 2023. Accelerating Sparse LU Factorization with Density-Aware Adaptive Matrix Multiplication for Circuit Simulation. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC56929.2023.10247767

[39] Buse Yilmaz. 2021. Graph Transformation and Specialized Code Generation For Sparse Triangular Solve (SpTRSV). *arXiv preprint arXiv:2103.11445* (2021).