Trojan Horse: Aggregate-and-Batch for Scaling Up Sparse Direct Solvers on GPU Clusters

Yida Li
SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
yida.li@student.cup.edu.cn

Siwei Zhang
SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
siwei.zhang@student.cup.edu.cn

Yiduo Niu
SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
yiduo.niu@student.cup.edu.cn

Yang Du
SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
yang.du@student.cup.edu.cn

Qingxiao Sun
SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
qingxiao.sun@cup.edu.cn

Zhou Jin
SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
jinzhou@cup.edu.cn

Weifeng Liu
SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
weifeng.liu@cup.edu.cn

Abstract

Sparse direct solvers are critical building blocks in a range of scientific applications on heterogeneous supercomputers. However, existing sparse direct solvers have not been able to well leverage the high bandwidth and floating-point performance of modern GPUs. The primary challenges are twofold: (1) the absence of a mechanism for aggregating small tasks to saturate the GPU, and (2) the lack of a mechanism for executing a diverse set of small tasks in batch mode on a single GPU.

We in this paper propose a strategy called Trojan Horse, which significantly enhances the execution efficiency of sparse direct solvers on GPU clusters. This mechanism divides each process's work into two stages: Aggregate (with two modules Prioritizer and Container) and Batch (with two modules Collector and Executor). In the Aggregate stage, a process first assesses the urgency of the input tasks through the Prioritizer module, and based on their priority, sends them to the Collector module or the Container module. In the batch stage, the Collector module receives high-priority heterogeneous tasks from the Prioritizer module and retrieves enough tasks from the Container module to send them to the Executor module for batch execution on GPU. In addition, our strategy is independent of solver libraries, and is integrated into SuperLU_DIST and PanguLU.

In the scale-up evaluation on a single NVIDIA A100 GPU, the Trojan Horse strategy delivers speedups of up to 418.79x (5.47x on average) for SuperLU_DIST and up to 5.59x (2.84x on average) for PanguLU. In the scale-out evaluation on two 16-GPU clusters from NVIDIA and AMD, respectively, Trojan Horse continues to deliver strong performance gains for both SuperLU_DIST and PanguLU across different GPU counts.

CCS Concepts: • **Mathematics of computing** → **Solvers**; • **Computing methodologies** → **Distributed algorithms**.

Keywords: Sparse direct solver, Task aggregation, Batched kernels, GPU cluster

ACM Reference Format:
Yida Li, Siwei Zhang, Yiduo Niu, Yang Du, Qingxiao Sun, Zhou Jin, and Weifeng Liu. 2026. Trojan Horse: Aggregate-and-Batch for Scaling Up Sparse Direct Solvers on GPU Clusters. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '26), January 31 – February 4, 2026, Sydney, NSW, Australia.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3774934.3786442

PPoPP '26, Sydney, NSW, Australia

ACM ISBN 979-8-4007-2310-0/2026/01
https://doi.org/10.1145/3774934.3786442

1 Introduction

Sparse direct methods [31, 41] use Gaussian elimination to solve a sparse linear system $Ax = b$, where A is a sparse matrix, typically representing the coefficients of a linear system, x is a vector of unknown variables, and b is a vector of constants on the right-hand side of the linear equations. A number of scientific fields, such as finite element analysis [46, 60, 84] and circuit simulation [55, 89, 101, 111], particularly require sparse direct solvers.

As the scale of the problems to be solved increases, it becomes essential to leverage heterogeneous supercomputers [1, 8, 9, 14, 16, 98]. However, current sparse direct solvers primarily emphasize scaling out to more compute nodes [44, 67, 68], often neglecting the need to scale up the efficiency of the individual GPUs in a cluster.

There are two primary factors limit current distributed heterogeneous sparse direct solvers SuperLU [66, 68], PaStiX [52, 70] and PanguLU [44] from effectively scaling up the computational power of a single process managing an individual GPU.

The first reason is the absence of a mechanism for collecting small tasks to saturate a GPU. Unlike dense direct methods, which provide sufficiently large tasks (typically involving dense submatrix blocks with thousands of orders and millions of elements) to well use a GPU [20], sparse direct methods generally only offer small submatrix blocks (often with tens of orders and thousands of nonzero elements). This limitation arises from the sparsity inherent in scientific problems and the use of the multifrontal [5, 40, 42] and supernodal [35, 36] methods, as well as the sparse blocking [44] schemes. It is evident that executing these small submatrices individually is insufficient to well utilize a high performance GPU. Therefore, it is required to first evaluate the dependencies of tasks, and then identify and collect small tasks that can be executed together on a single GPU.

The second reason is the lack of a mechanism for executing a diverse set of small tasks in parallel in batch mode on a single GPU. Specifically, current batched methods [50, 58, 62, 77] are good at executing similar linear algebra operations, but fail to effectively address the diverse characteristics of the small tasks generated by sparse direct solvers, including: (1) arbitrary matrix sizes (typically up to a few thousand orders); (2) variable sparsity levels (ranging from fully dense to highly sparse); (3) three kinds of kernels (i.e., (i) LU factorisation for diagonal blocks, (ii) triangular solves for row/column non-diagonal blocks, and (iii) Schur complement matrix multiplication for trailing submatrices); and (4) two kinds of dependency relationships (i.e., (i) strongly dependent tasks with a fixed execution order, and (ii) order-independent Schur complement tasks, where the only dependency arises during the final accumulation step, allowing atomic-add update operations despite potential write conflicts). To meet the aforementioned requirements, a further strategy is necessary to execute these parallelisable small tasks in batch mode on a single GPU.

To tackle the two issues outlined above, we in this paper propose an aggregate-and-batch strategy called Trojan Horse to accelerate the most time-consuming numeric factorisation phase of sparse direct solver. Unlike the existing task execution model of running batched dense kernels on the same level of the elimination tree (in the newer versions of SuperLU [53] and rank-structured STRUMPACK [26]) and executing relatively large sparse block tasks one-by-one without batching (in the PanguLU [44]), our Trojan Horse strategy divides each process's task management into two stages: `Aggregate` and `Batch`, and introduces four functional modules: (a) a `Prioritizer`, (b) a `Container`, (c) a `Collector`, and (d) an `Executor`. The first two modules belong to the `Aggregate` stage running on CPU, while the latter two belong to the `Batch` stage running on single GPU.

During the execution of the numeric phase, the tasks in our strategy are processed as follows: (1) In the `Aggregate` stage, the `Prioritizer` module of a process assesses the urgency of each task based on the overall task dependencies. If the task lies on the critical path, it is directly forwarded to the `Collector` module in the `Batch` stage; otherwise, it is placed in the `Container` module. (2) In the `Batch` stage, the `Collector` receives urgent tasks from the `Prioritizer` and gathers enough non-urgent tasks from the `Container` to better utilize the GPU resources. Finally, the `Executor` processes the heterogeneous tasks, with arbitary matrix sizes, variable sparsity levels, and three kinds of kernels, in a single-kernel batch mode. Upon completion, the results are communicated with other processes to progress the overall execution.

We integrate the Trojan Horse strategy into SuperLU_DIST and PanguLU, and evaluate their performance using 200 sparse matrices from 31 different kinds of the SuiteSparse Matrix Collection [33]. On an NVIDIA A100 GPU, compared to the latest original versions of SuperLU_DIST and PanguLU, the Trojan Horse strategy scales up the single-GPU execution efficiency by an average of 5.47x and 2.84x (up to 418.79x and 5.59x), respectively. Furthermore, on a 16-card NVIDIA H100 GPU cluster, the Trojan Horse strategy enhances the overall performance of SuperLU_DIST and PanguLU by an average of 3.5x and 1.9x, respectively.

This work makes the following contributions:

- We propose the Trojan Horse strategy for efficiently aggregating and batching fine-grained small tasks to saturate high-end GPUs.
- We integrate the Trojan Horse strategy into SuperLU_DIST and PanguLU to effectively improve their task management and kernel performance.
- We bring SuperLU_DIST and PanguLU obviously better scale-up throughput and comparable scale-out performance.

2 Background and Motivation

2.1 Sparse LU Factorisation

Sparse direct solvers, such as sparse LU factorisation, typically consist of three major phases: reordering, symbolic, and numeric, as shown in Figure 1. The reordering phase aims to permute the matrix A to improve computational and storage costs [7], the symbolic phase identifies the sparsity structures of the resulting matrices L and U [27, 45, 48], and the numeric phase performs the actual factorisation, which is generally the only phase processing a large amount of floating point operations.

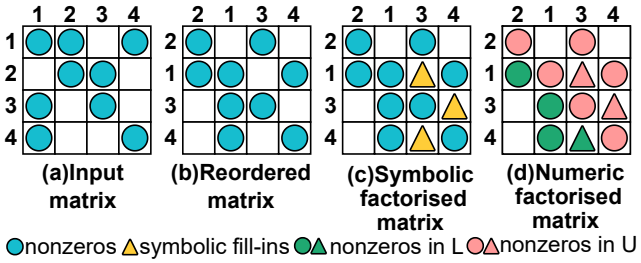


Figure 1. The major phases of sparse LU factorisation.

Figure 2 shows the time breakdown of the three phases of 10 matrices (see Tables 2 and 4) running with SuperLU on one core of an Intel Xeon 6230 CPU. As can be seen, the numeric phase spends most execution time, on average 97%, and is almost the only stage scales to a large amount of compute nodes [49]. This motivates us to focus on optimizing the numeric phase on heterogeneous GPU clusters.

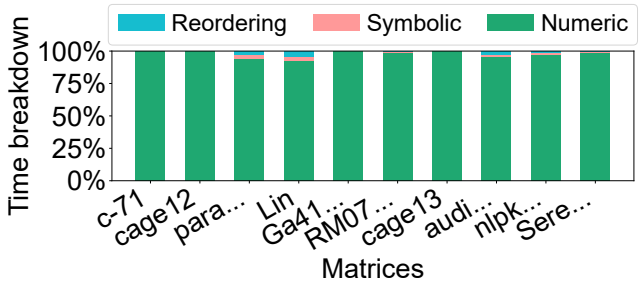


Figure 2. Time breakdown of the three phases.

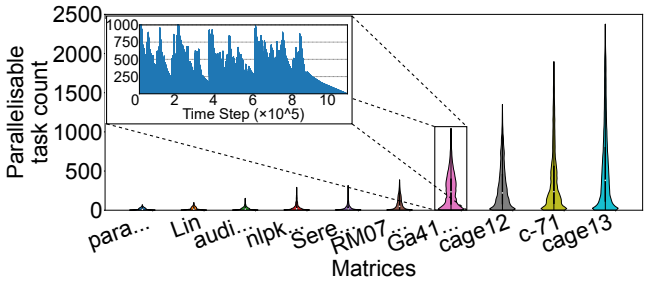
The main approaches for computing the numeric phase include multifrontal [40, 42], supernodal [35, 36], and sparse blocking [44] approaches. These methods break the matrix into small dense or sparse blocks, which can be processed independently. The block-cyclic data distribution is commonly employed to assign the submatrices to processes for efficient scheduling on elimination trees [9, 11]. These tasks may vary greatly in sizes and compute patterns, and irregular task dependencies complicate their parallelisation.

Due to the small sizes and the dependencies of the matrices associated with each task, traditional task management methods are not well-suited for leveraging modern GPUs. This limitation highlights the need for our Trojan Horse strategy to aggregate and batch small tasks for saturating high-end GPUs.

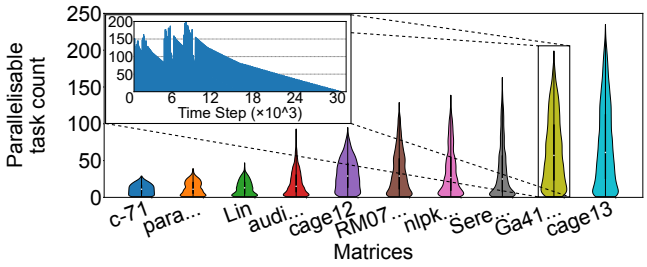
2.2 Aggregate: to Prepare More Tasks for a GPU

A right-looking sparse LU factorisation includes three kinds of compute tasks: local LU factorisation on diagonal blocks, triangular solve on row/column blocks, and Schur complement matrix multiplication on trailing blocks. SuperLU organizes dense tasks using supernodes, while PanguLU constructs sparse tasks using sparse blocks, and whether dense or sparse, their dependencies formed by an elimination tree or directed acyclic graphs (DAGs) are similar.

In the numeric phase, tasks that are mutually independent can be executed concurrently. To assess the potential degree of parallelism, we conduct a static analysis on the DAGs derived from the supernode and block structures of SuperLU and PanguLU. Specifically, we iteratively traverse a DAG, removing nodes of no in-degrees at each step, and record the number of tasks that can be executed in parallel, continuing this process until the entire DAG is eliminated to empty.



(a) Distribution of parallelisable tasks in SuperLU



(b) Distribution of parallelisable tasks in PanguLU

Figure 3. A static analysis of parallelisable task counts. Note that the supernodal tasks in SuperLU are in general much smaller than the sparse block tasks in PanguLU, leading to distinct total numbers of tasks and time steps.

Figure 3 presents the results of a static analysis of the distribution of parallelisable tasks during the execution of ten sparse matrices on SuperLU and PanguLU. In the violin

plots depicting the distributions, the width at each vertical position indicates the density of occurrences for a specific batch size (the wider the shape, the more frequently that batch size appears during factorisation). As can be seen, both solvers generate a substantial number of parallelisable tasks in the time steps (i.e., DAG levels). Taking the matrix 'Si41Ge41H72' highlighted, the highest numbers of tasks can run in parallel are 975 and 153 on SuperLU and PanguLU, respectively. The observation brings the potential to run the tasks in a batch mode throughout the matrix factorisation.

To ensure an adequate number of tasks while respecting dependency constraints, an effective runtime task aggregation approach is crucial. The strategy becomes even more complex, when considering the assignment of tasks to multiple processes distributed across multiple compute nodes in a 2D block-cyclic pattern. Furthermore, task priority must also be taken into account, necessitating the use of a task container to store low-priority tasks for deferred execution. Such considerations motivate us to design the `Aggregate` stage with two modules called `Prioritizer` and `Container` in our Trojan Horse strategy.

2.3 Batch: to Selectively Run the Tasks

The three types of tasks, i.e., diagonal LU factorisation, upper/lower triangular solve, and Schur complement matrix multiplication, may also work in parallel, leading to a requirement to batch diverse tasks. Figure 4 shows the diversity by giving an example of factorising a sparse matrix of size 6-by-6. The matrix is organized as 3x3 sparse blocks (blocks 6, 8 and 9 are dense, while the remaining blocks are sparse). There are in total 14 tasks (three diagonal LU factorisation, six triangular solve, and five Schur complement operations) and their dependencies are shown in the DAG of Figure 4.

It is possible to run the 14 tasks through a number of combinations corresponding to distinct scheduling considerations [81, 103]. In this example, we choose to combine tasks with high and low priorities, allowing non-critical tasks to be deferred and executed alongside critical ones, thereby optimizing the utilisation of GPU resources. The execution of this example includes five batches:

Tasks '2T' and '4T' (triangular solves on blocks 2 and 4) are triggered by task '1F' (LU factorisation on diagonal block 1) and are mutually independent, allowing them to run in a batch mode.

Tasks '5S0' (Schur update on block 5) and '7T' can also be executed in batch mode, despite involving different kernel operations.

Tasks '5F' and '8S0' are triggered by different diagonal blocks and can be batched, despite calling different kernels.

Tasks '3T' and '8T' are triggered by different diagonal blocks and can be batched. Additionally, since block 3 is sparse and block 8 is dense, the use of distinct kernels for each will improve performance.

Tasks '9S0' (sparse) and '9S1' (dense), triggered by different diagonal blocks, both compute Schur update on block 9. Although running them in parallel brings write conflict, their floating point operations are independent. Thus, batching them may be beneficial if the write conflict can be resolved.

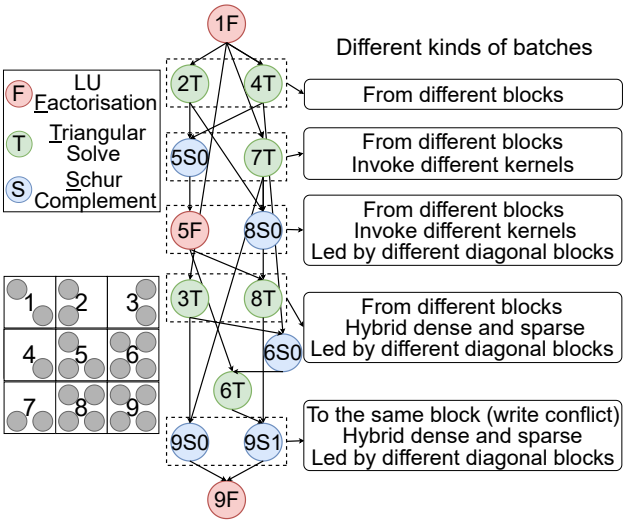


Figure 4. An example of the need to batch diverse tasks.

As illustrated by the above example, greater performance gains can be achieved by exploiting diverse parallelism. However, existing batched work fails to fully accommodate such diversities, including various input size and sparsity, kernel types, and potential write conflicts. A dedicated batched kernel is necessary. Such limitations motivate us to design the `Batch` stage with two modules called `Collector` and `Executor` in our Trojan Horse strategy.

3 Trojan Horse

3.1 Overview

The Trojan Horse focuses on scaling up the execution efficiency of a single GPU in heterogeneous GPU clusters, and in conjunction with existing distributed libraries such as SuperLU_DIST and PanguLU, enhances overall performance.

In traditional SuperLU_DIST and PanguLU solvers, the workflow of each process is basically three steps, i.e., receive a task, execute the task, and send results to other processes. Moreover, SuperLU can batch triangular solve and Schur complement tasks independently in one level of the elimination tree when the compute workload is large enough [49]. When the two solvers integrate with the Trojan Horse, the workflow of each process will be divided into two stages: `Aggregate` and `Batch`, and four functional modules: the `Prioritizer` and `Container` (both belong to the `Aggregate` stage), and the `Collector` and `Executor` (bothpipi belong to the `Batch` stage). Figure 5 illustrates the two stages and four modules of a process with the Trojan Horse.

The goal of the Aggregate stage is to identify and collect tasks. The Prioritizer module (see the upper left part of Figure 5) in this stage is responsible for making an initial judgment on the tasks received by the process, dividing them into two cases: If the task is on the critical path, it is marked as high priority and directly sent to the Collector of the Batch stage. Otherwise, it is marked as low priority and sent to the Container module (see the lower left part of Figure 5). Tasks in the Container are managed according to their criticality, and are deferred for execution. When they are executed depends on how many tasks are needed in the Batch stage to saturate GPU resources.

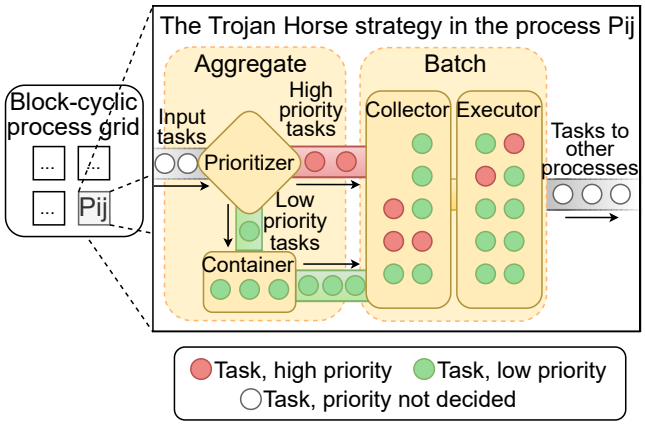


Figure 5. An overview of the Trojan Horse strategy with two stages Aggregate and Batch. The Aggregate stage has two modules Prioritizer (on CPU) and Container (on CPU), and the Batch stage has two modules Collector (on CPU) and Executor (on GPU).

The goal of the Batch stage is to efficiently execute tasks. The Collector module (see the left of the right part of Figure 5) extracts tasks from the Prioritizer and Container modules of the Aggregate stage. Once the task volume is sufficient to saturate the GPU or reaches delay threshold, the tasks are sent to the Executor module (see the right of the right part of Figure 5) for batch execution. The Executor module is also responsible for executing highly diverse tasks, including those that may be dense or sparse, involve LU factorisation, triangular solves, or Schur complement multiplication, and may have write conflicts between some tasks.

3.2 An Example of Using the Trojan Horse

Before the detailed explanation in the following subsections, we begin by presenting an example in Figure 6 that illustrates how the Trojan Horse strategy enhances the task execution behavior of SuperLU and PanguLU. Figure 6 has six subfigures. In Figure 6(a), each number on the blocked matrix labels a non-empty block, while numbers adjacent to the matrix represent the supernodes or indices of the diagonal blocks. In Figure 6(b), an elimination tree, or a DAG, of the numeric factorisation procedure is plotted. In Figure 6(c), the complete dependencies of all 22 tasks are shown in a more detailed DAG, and these tasks are divided into 5 parts by the diagonal blocks 0-4 triggering them.

Figures 6(d) and 6(e) show the timeline of SuperLU and PanguLU with four processes. To execute these tasks, SuperLU spends 10 time units, and PanguLU spends 11. SuperLU batches tasks of the same type and from the same elimination tree level. For example, three triangular solves in step 2 (labeled as purple ①), and four Schur updates in step 3 (labeled as purple ②) can be batched. The two batches are both associated with level 0 of elimination tree (containing supernodes 0 and 1). In contrast, PanguLU executes tasks based on priority and without batching.

Figure 6(f) shows the timeline of SuperLU or PanguLU with the Trojan Horse. On process 0, two tasks '7F' and '15S1' are batched in timestep 4 (labeled as purple ③), although their types are different. On process 1, two triangular solve tasks '6T' and '2T', though belong to different blocks, are batched in timestep 2 (labeled as purple ④). On process 2, two triangular solve tasks '4T' and '10T', are batched in timestep 2 (labeled as purple ⑤), although they are led by different diagonal blocks. The following two subsections, Aggregate stage and Batch stage, will illustrate this in detail.

In SuperLU, matrix blocks can have varying sizes, while in PanguLU, blocks may exhibit different sparsity. This example shows that the Trojan Horse strategy could batch tasks from different blocks, of different sizes, sparsity and types.

3.3 Aggregate Stage

In this subsection, we introduce the two modules named Prioritizer and Container in the Aggregate stage. The Aggregate stage functions as a task scheduler, supplying tasks to the Collector module within the Batch stage.

Module 1: Prioritizer

The Prioritizer is designed to tag tasks ready to be executed and separate them into high- and low-priority ones. Firstly, the Prioritizer needs to find tasks ready to be executed. It traverses the supernode elimination tree (Figure 6(b)) or the DAG (Figure 6(c)), and then identifies each node with no predecessors and ready to be executed. For example, at the first time step, nodes '1F' and '3F' in Figure 6(c) have no predecessors. They should be tagged ready.

Secondly, after tagging tasks, the Prioritizer determines which of them are more urgent. Tasks belonging to the same block are assigned the same priority, and those closer to the main diagonal are given higher priority to make the subsequent diagonal blocks ready to be factorised earlier. Here we use the difference between the row and the column index of a matrix block as its distance to the main diagonal. For example, if tasks '6T' (distance=1) and '12T' (distance=3) in Figure 6(c) are tagged available, '6T' should be more urgent than '12T'. When a task is determined urgent, it would be pushed

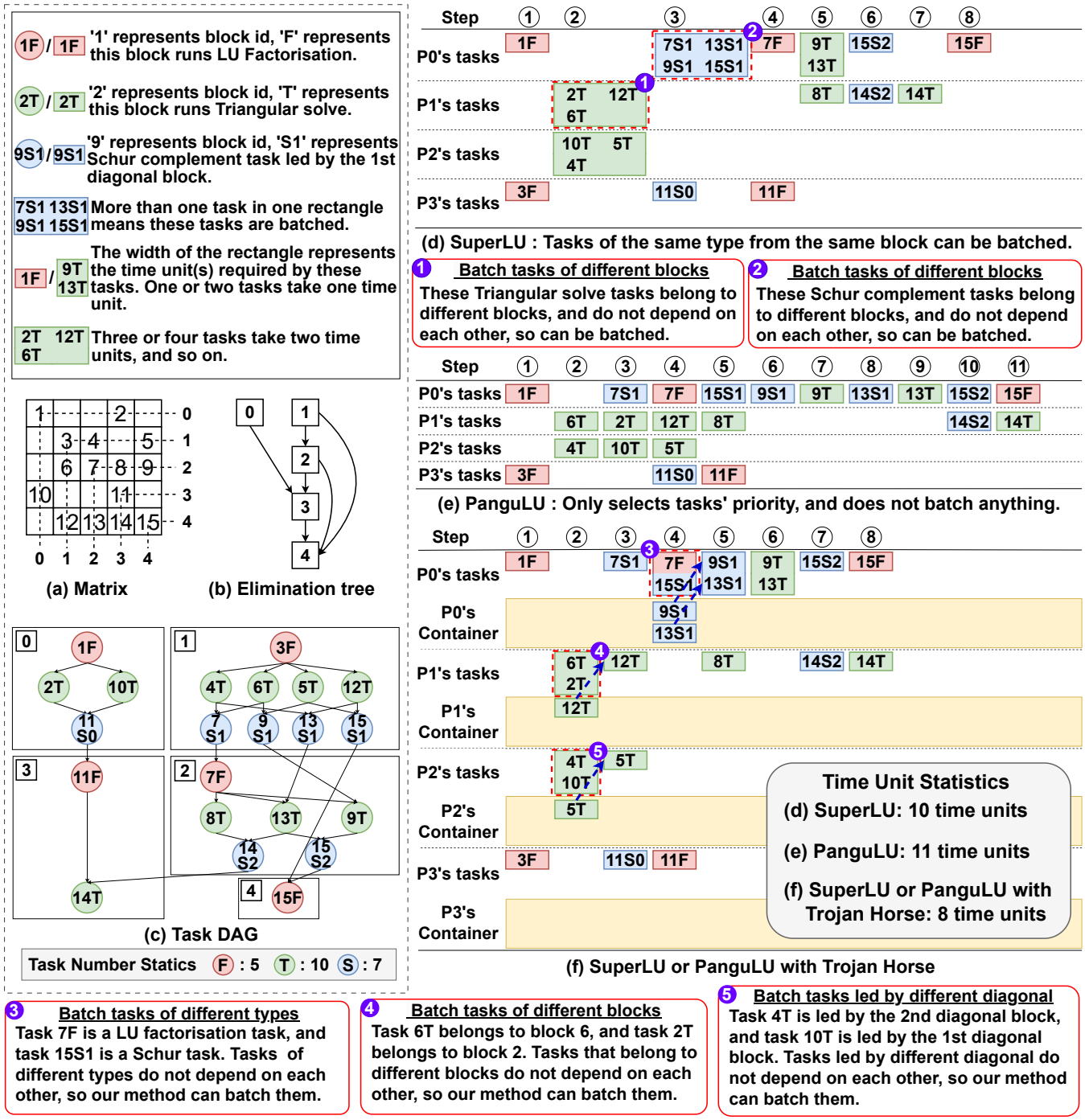


Figure 6. SuperLU and PanguLU without/with the Trojan Horse. (a) is the blocked matrix to be factorised. (b) is the elimination tree of numeric factorisation, and (c) is the complete dependencies of all 22 tasks. (d), (e) and (f) show the timeline of SuperLU and PanguLU, without/with the Trojan Horse to perform numeric factorisation with four processes.

to the `Collector` in the `Batch` stage. Otherwise, it would be temporarily stored in the `Container` for later execution.

Module 2: Container

The `Container` serves as a temporary buffer for tasks with relatively lower priority, as classified by the `Prioritizer`. These tasks are not immediately needed for execution but may become necessary to fill computational capability and maintain high GPU utilisation in later timesteps. As shown in lines 15 and 16 of Algorithm 1, the `Collector` requests

low-priority tasks from the `Container` when high-priority tasks are insufficient to saturate GPU resources.

To prevent low-priority tasks from being executed ahead of more urgent ones, the `Container` must always return the highest-priority task among those it stores. This requires maintaining strict order among stored tasks based on their priority. We adopt a priority queue (implemented as a heap) to manage tasks, ensuring that both insertion and retrieval operations are fast while preserving the priority ordering.

Algorithm 1: Task Collection in the Trojan Horse

```
1  DAG = BuildNumericFactDAG()
2  Container = []
3  while !DAG.isEmpty do
4      Collector = []
5      while !Container.isEmpty or DAG.haveAvailableTask do
6          UrgentTask = Prioritizer(DAG.AvailableTasks)
7          if Collector.isFull then
8              Container.Push(UrgentTask)
9              Container.PushAll(DAG.AvailableTasks)
10             break
11         end
12         Collector.Push(UrgentTask)
13     end
14     while !Collector.isFull do
15         NotUrgentTask = Container.Pop()
16         Collector.Push(NotUrgentTask)
17     end
18     GPU.AsyncExecutor(Collector)
19 end
```

3.4 Batch Stage

In this subsection, we introduce the two modules of the `Batch` stage: the `Collector` and the `Executor`. The `Collector` gets tasks from the `Aggregate` stage and the `Executor` executes them in batch mode on GPU.

Module 1: Collector

The `Collector` is responsible for assembling a batch of tasks from the `Aggregate` stage to be dispatched to the GPU for execution. As described in Algorithm 1, it follows a two-phase strategy: first, it receives high-priority tasks pushed by the `Prioritizer`; then, if capacity permits, it supplements the batch with lower-priority tasks from the `Container`. This ensures that the most urgent tasks are always considered first, while still maintaining high throughput.

To improve hardware utilisation, the `Collector` features a fixed capacity aligned with the count and shared memory size of GPU stream multiprocessor (SM). Each task consumes a certain number of CUDA blocks and a portion of shared memory. When a task is collected, the `Collector` calculates its CUDA block count and shared memory usage. Once the CUDA block count or the shared memory usage of collected tasks is too high, restricting the number of active blocks, the `Collector` is considered full and ready for execution.

If the `Collector` becomes full before all urgent tasks can be collected, the remaining urgent tasks are deferred by the `Prioritizer`, which pushes them to the `Container` for future collection. These tasks, despite being delayed by capacity constraints, will remain marked as urgent and will be prioritized for collection in the subsequent batch execution.

Module 2: Executor

The `Executor` module provides a flexible batched kernel supporting heterogeneous tasks. Each task can be independently customized, supporting four types: GETRF (LU factorisation), TSTRF (upper triangular solve), GESSM (lower triangular solve), and SSSSM (Schur complement GEMM). Tasks can operate on either sparse or dense matrix blocks and optionally enable atomic operations to avoid write conflicts.

To execute diverse tasks within a single kernel, we design an array mapping from CUDA blocks to tasks. Before kernel launch, the array elements are set to the starting block indeces of each task. During execution, each CUDA block finds the task should be executed via binary search on this array.

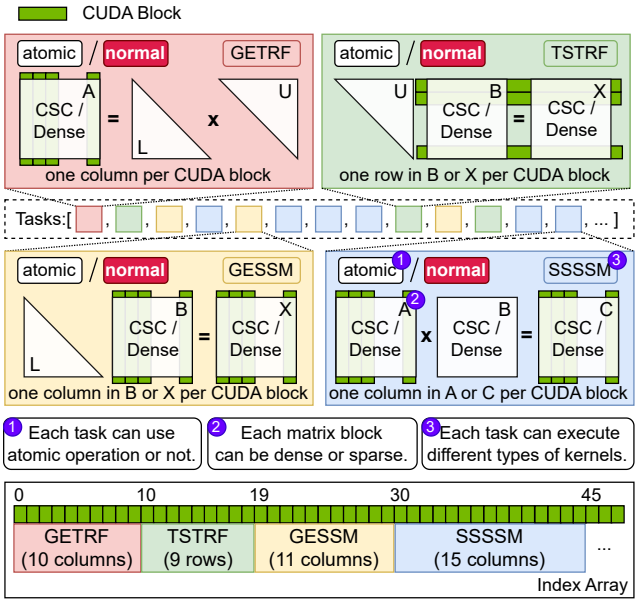


Figure 7. An overview of Batch stage. The upper part shows the implementation of each task type, and the lower part shows the mapping between tasks and CUDA blocks.

The lower part of Figure 7 shows the task-to-block mapping for these four tasks. The GETRF task, which processes 10 columns, is assigned 10 CUDA blocks, starting from block 0 (blocks 0 to 9). The TSTRF task, handling 9 rows, begins at block 10 and occupies 9 blocks (blocks 10 to 18). The subsequent GESSM task, corresponding to 11 columns, starts at block 19 and uses 11 blocks (blocks 19 to 29). Finally, the

SSSSM task, which involves 15 columns, starts from block 30 and occupies 15 blocks (blocks 30 to 44).

As illustrated in the top section of Figure 7, each GETRF task is assigned one CUDA block per column and follows a synchronization-free left-looking approach [110] for LU factorisation. Prior to factorisation, the input matrix is gathered into a dense global memory space to expedite element addressing. During factorisation, each CUDA block sequentially processes the elements within its assigned column, leveraging all available threads to perform vectorized multiplications for element updates. Once completed, the results are scattered back to the original sparse memory spaces.

TSTRF and GESSM tasks are each assigned one CUDA block per row (for TSTRF) or per column (for GESSM) of matrix B. When the corresponding row or column fits within shared memory, the CUDA block first gathers the data into dense shared memory before factorisation and scatters the results back afterward. Otherwise, the computation proceeds directly in global memory. In either case, each CUDA block processes its assigned row or column sequentially, with all threads in the block cooperating to perform parallel updates.

The SSSSM tasks employ a column-column multiplication method, where each element of matrix B independently multiplies a column of matrix A. For dense GEMM tasks, the `Executor` directly treats the value arrays as column-major dense data, avoiding the overhead of processing sparse indices.

3.5 Integration into SuperLU_DIST and PanguLU

The Trojan Horse is not an independent solver, and needs integration into a sparse direct solver to work. We integrate the Trojan Horse into SuperLU_DIST and PanguLU.

3.5.1 Integration into SuperLU_DIST. In SuperLU, the most significant difficulty is the high scheduling overhead. SuperLU uses very small supernode sizes, which causes the total task counts to increase significantly. The bottleneck arises at the task aggregation stage on the CPU. To overcome this challenge, we aggregate all vectors of matrix U in advance, therefore all Schur complement tasks in one supernode can be done in a relative larger GEMM.

3.5.2 Integration into PanguLU. PanguLU maintains its own internal task queue, making the integration of the Trojan Horse's `Container` nontrivial. We delegate all Schur complement tasks from the task queue of PanguLU to the Trojan Horse, while allowing the original task queue in PanguLU to manage all other task types. This strategy is based on two considerations: firstly, Schur complement tasks are numerous enough to represent the dominant workload; secondly, offloading these tasks does not disrupt the synchronization-free architecture that PanguLU originally maintains.

4 Experiments

4.1 Experimental Setup

In our experiments, we evaluate six solver variants: v9.1.0 of SuperLU_DIST [67] and v5.0.0 of PanguLU [44] as baselines, along with their respective versions enhanced with the Trojan Horse. For PanguLU with Trojan Horse, we also evaluate a version replacing the `Executor` module with four CUDA streams. We also evaluate PaStiX v6.4.0 scheduling with the 'dmdas' strategy of StarPU v1.4.8. For all solvers, their numeric factorisation phase always uses double precision. It is worth noting that a recent work, Caracal [88], is not evaluated because it is not open sourced, and its closed-source version cannot run on our platforms.

The evaluation is conducted in two parts: the first examines scalability on single-GPU (scale-up), while the second extends to distributed multi-GPU environments (scale-out).

The scale-up experiments, detailed in Sections 4.2–4.3, are conducted on NVIDIA RTX 5060Ti and 5090 GPUs (Table 1). These platforms share the same architecture but differ in theoretical performance, enabling a fair comparison of GPU utilisation. We use four matrices from SuiteSparse [33], previously employed in SuperLU and PanguLU benchmarks [44, 73]. They are moderately sized, large enough for GPU parallelism yet small enough for a single GPU memory capacity. To verify generality, we also evaluate 200 additional square matrices from SuiteSparse on an NVIDIA A100 40GB GPU.

GPU	#Cores	FP64 peak	Memory	B/W
RTX 5060Ti	4,608	0.37 TFlops	16 GB	0.45 TB/s
RTX 5090	21,760	1.64 TFlops	32 GB	1.79 TB/s
A100 PCIe	6,912	9.75 TFlops	40 GB	1.56 TB/s

Table 1. Three GPU platforms for scale-up evaluation.

Matrix	n(A)	nnz(A)	SuperLU nnz($L+U$)	PanguLU nnz($L+U$)
c-71	76.6K	860K	49.4M	24.9M
cage12	130K	2.03M	550M	537M
para-8	156K	2.09M	187M	178M
Lin	256K	1.77M	216M	194M

Table 2. The matrices tested in scale-up evaluation.

The scale-out experiments, detailed in Section 4.4, are conducted on two GPU clusters listed in Table 3, respectively equipped with 16 NVIDIA H100 GPUs (two nodes linked with 400 Gbps InfiniBand, eight GPUs per node) and 16 AMD MI50 GPUs (four nodes linked with 200 Gbps InfiniBand, four GPUs per node). Each GPU is managed by a dedicated MPI process, and the processes are evenly distributed across the evaluated nodes (for example, eight H100 GPUs are tested

on two nodes, with four GPUs activated per node). The six matrices used, listed in Table 4, are also from SuiteSparse [33] and were tested in earlier studies [44, 68, 90]. Their L and U are generally much larger than those in the scale-up tests, reflecting scale-out behavior on distributed platforms.

16 GPUs	#Cores	FP64 peak	Memory	B/W
H100 SXM	14,592	25.61 TFlops	80 GB	2.04 TB/s
MI50 PCIe	3,840	6.71 TFlops	16 GB	1.02 TB/s

Table 3. Two 16-card GPU clusters for scale-out evaluation.

Matrix	n(A)	nnz(A)	SuperLU nnz($L+U$)	PanguLU nnz($L+U$)
Ga41As41H72	268K	18.5M	4.61G	4.59G
RM07R	381K	37.4M	2.68G	2.14G
cage13	445K	7.48M	4.68G	4.66G
audikw_1	943K	77.6M	2.46G	2.43G
nlpkkt80	1.06M	28.1M	3.80G	3.28G
Serena	1.39M	64.1M	5.42G	5.38G

Table 4. The matrices tested in scale-out evaluation.

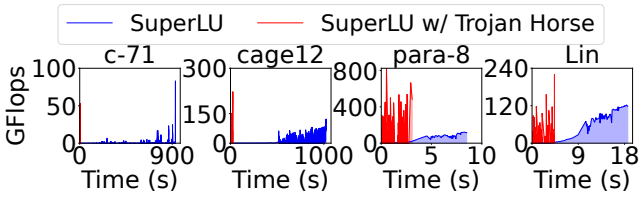
Across both experimental phases, we compile these solver libraries with Intel MPI 2021.1 and OpenBLAS 0.3.29. All experiments are performed by using CUDA 12.8 and ROCm 4.3. Based on prior tuning experience, we set the maximum supernode size in SuperLU to 256, the block size in PanguLU to 512, and the range of block size in PaStiX from 160 to 320, as these yields generally the best performance.

In Section 4.5, we compare four solver variants of SuperLU_DIST and PanguLU without and with the Trojan Horse on an NVIDIA H100 GPU against two CPU-based solvers SuperLU_DIST v9.1.0 and MUMPS v5.6.0 running on an Intel Xeon 6462C CPU of 32 cores (Sapphire Rapids) and 512 GB DDR5 memory at 4800 MT/s.

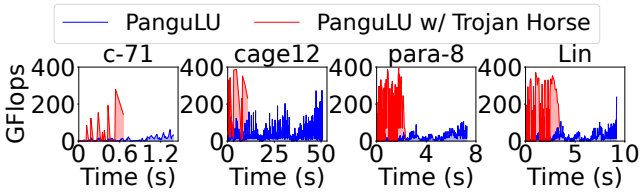
4.2 Scale-Up Evaluation

We first demonstrate the improvement in absolute performance of our work (Figure 8), then examine the scale-up effects on the RTX 5060Ti and RTX 5090 GPUs (Figure 9), and finally evaluate a broader set of matrices on the A100 GPU (Figure 10).

Figure 8 records the CUDA kernel execution performance during numeric factorisation, measured in GFLOPS on the y-axis over time on the x-axis. The blue line and the corresponding light blue area represent the original SuperLU or PanguLU without Trojan Horse, while the red line and area correspond to the two solvers enhanced with Trojan Horse. As can be seen, the red curves achieve substantially higher throughput and therefore complete the numeric factorisation much faster than the blue ones. Specifically, integrating Trojan Horse accelerates kernel execution by 15.02x for SuperLU and 2.92x for PanguLU. Consequently, the overall numeric factorisation efficiency is improved by 15.05x and 2.14x for the two solvers, respectively.



(a) SuperLU



(b) PanguLU

Figure 8. Numeric factorisation timelines of SuperLU and PanguLU without and with the Trojan Horse. The x-axis is the execution timeline, and the y-axis represents the throughput (in GFlops) of kernels running on the RTX 5090 GPU.

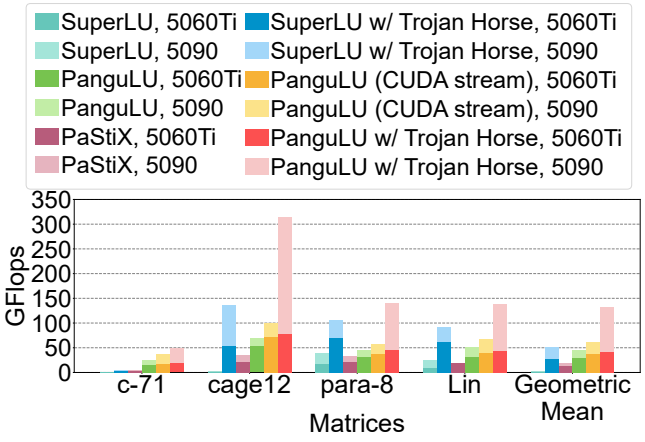
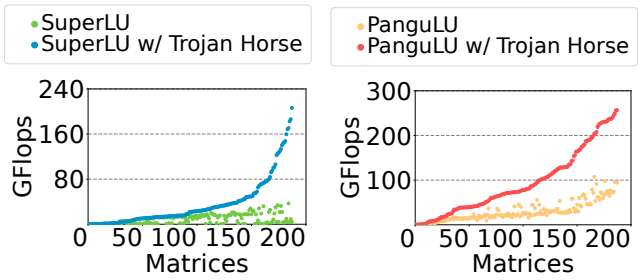


Figure 9. Numeric factorisation performance of solver variants across four matrices. Each bar corresponds to a solver variant, with the full segment indicating performance on the RTX 5090 and the lower segment on the RTX 5060Ti.

Figure 9 presents the numeric factorisation performance of different solver variants on the four example matrices. Across platforms, SuperLU achieves an average speedup of 1.09x (up to 1.54x) on the RTX 5090 compared to the RTX 5060Ti, increasing to 1.26x (up to 1.93x) when integrated with the Trojan Horse. As for PanguLU, before integrating

Trojan Horse, the RTX 5090 is only 1.56x (up to 1.83x) faster than the RTX 5060Ti; after applying Trojan Horse, the RTX 5090 becomes 3.22x (up to 4.00x) faster, approaching the ratio of the peak performance and memory bandwidth of the two GPUs (see Table 1). Notably, over the RTX 5060Ti, the performance gains by the RTX 5090 are obviously amplified when Trojan Horse is employed, highlighting its strong scale-up capability in leveraging increasingly powerful GPUs.

The performance improvement depends on both the solver design and the characteristics of the input matrix. PanguLU benefits more from scaling due to its higher degree of parallelism, whereas SuperLU remains constrained by longer task dependencies. For instance, the matrix 'cage12', with its numerous off-diagonal nonzeros, enables more effective task aggregation and higher GPU utilisation.



(a) SuperLU_DIST **(b)** PanguLU

Figure 10. Numeric factorisation scale-up evaluation on 200 matrices on the A100 GPU. The x-axis shows the matrices sorted by the performance of the solvers with Trojan Horse.

To assess the general speedup on various matrices, we conduct further performance evaluations on an NVIDIA A100 GPU using 200 square matrices from the SuiteSparse. These matrices cover 31 different kinds, a wide range of sizes, nonzeros in $L+U$, and flop counts. As shown in Figure 10, Trojan Horse yields an average (Geomean) speedup of 5.47x (up to 418.79x) for SuperLU, and 2.84x (up to 5.59x) for PanguLU.

4.3 Kernel Count Reduction and Time Breakdown

The reduction in kernel execution count reflects the effectiveness of the `Aggregate` stage. As shown in Table 5, for SuperLU, the total kernel count decreases to 1.10% on average, with a minimum reduction to 0.28%. Similarly, in Table 6, for PanguLU, the count is reduced to 1.48% on average, with a minimum of 0.37%. Despite the reductions, the total floating-point operations remain unchanged. With much fewer kernels, each execution often processes a much larger batch of work, leading to obviously higher GPU efficiency.

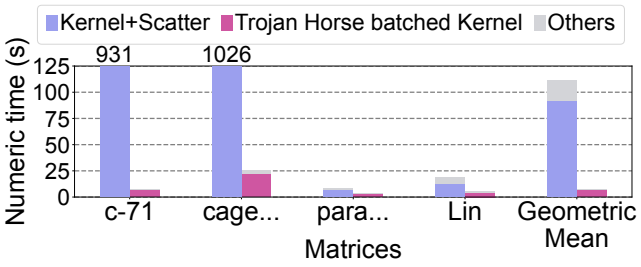
Figure 11 shows the total kernel time comparison between solvers without and with the Trojan Horse. After integrating, the kernel execution efficiency is improved by an average of 15.02x for SuperLU and 2.92x for PanguLU. Notably, with Trojan Horse, the kernel's share in the execution time basically remains unchanged, suggesting that the Trojan Horse enhances GPU utilisation with similar scheduling overhead.

Matrix	w/o Trojan Horse	w/ Trojan Horse	Rate
c-71	12,991,278	110,227	0.85%
cage12	28,722,440	80,157	0.28%
para-8	2,241,384	40,627	1.81%
Lin	3,345,581	112,727	3.37%
Geomean			**1.10%**

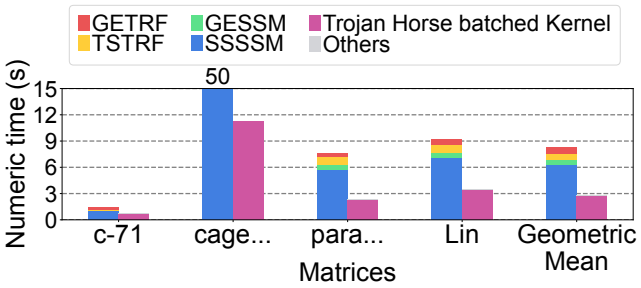
Table 5. Kernel count comparison of SuperLU_DIST.

Matrix	w/o Trojan Horse	w/ Trojan Horse	Rate
c-71	17,678	515	2.91%
cage12	226,568	847	0.37%
para-8	47,617	1,009	2.12%
Lin	81,844	1,699	2.08%
Geomean			**1.48%**

Table 6. Kernel count comparison of PanguLU.



(a) Numeric factorisation time breakdown of SuperLU_DIST



(b) Numeric factorisation time breakdown of PanguLU

Figure 11. Numeric factorisation time breakdown of two solvers without and with the Trojan Horse.

4.4 Scale-Out Evaluation

The scale-out evaluation tests strong scaling of six large matrices on two 16-card GPU clusters. When using 16 H100 GPUs, SuperLU with Trojan Horse achieves speedups of up

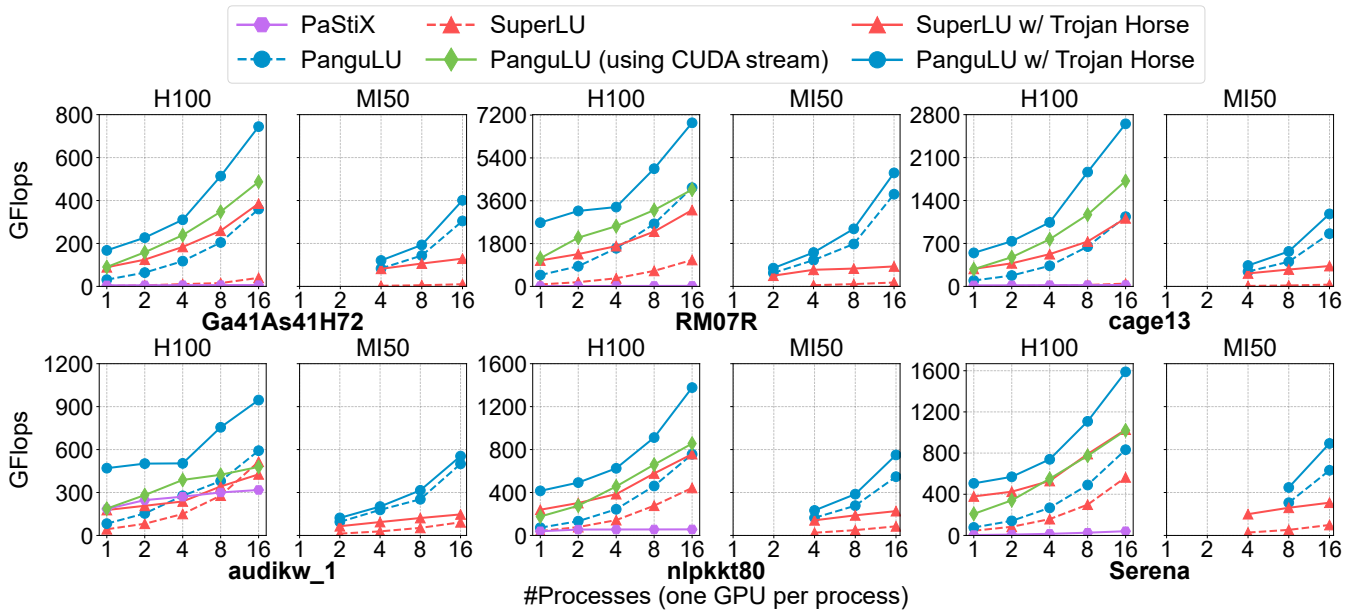


Figure 12. Scale-out comparison of three solvers, PaStiX with StarPU, SuperLU_DIST (without and with Trojan Horse), and PanguLU (without Trojan Horse, using CUDA stream, and with Trojan Horse), running six large matrices on 16-card NVIDIA H100 and AMD MI50 clusters. Some small GPU counts on the MI50 cluster cannot complete due to out-of-memory errors.

to 24.6x (3.5x on average) compared to its version without Trojan Horse, while PanguLU with Trojan Horse reaches up to 2.3x (1.9x on average). When using 16 MI50 GPUs, SuperLU with Trojan Horse delivers speedups of up to 12.8x (4.7x on average), and PanguLU with Trojan Horse achieves up to 1.4x (1.3x on average). Also, both solvers continue to deliver strong performance gains as the number of GPUs increases, and the versions with Trojan Horse are consistently faster than PaStiX and the CUDA stream-based PanguLU.

4.5 Comparison with Solvers on Modern CPUs

In Table 7, we compare three groups of performance data: (1) two existing GPU packages SuperLU_DIST and PanguLU without Trojan Horse (columns 2 and 3); (2) two existing CPU packages SuperLU_DIST and MUMPS (columns 4 and 5); and (3) two enhanced versions of SuperLU_DIST and PanguLU with Trojan Horse (columns 6 and 7). As shown, the two CPU libraries are often significantly faster than the two existing GPU implementations. However, when enhanced with Trojan Horse, the two GPU solvers match or surpass their CPU counterparts.

5 Related Work

A number of **software packages** were developed to accelerate sparse direct solvers. Representative libraries include UMFPACK [30], PARDISO [93], CHOLMOD [23], KLU [34] and SuiteSparseQR [32] for shared memory systems, as well as MUMPS [10], SuperLU [67, 92], PaStiX [51, 52, 70] and PanguLU [44] for distributed memory systems. Also, some solvers can use a single GPU for sparse Cholesky [57], LU [22], and QR [109]. In particular, SuperLU, PaStiX and PanguLU also support distributed heterogeneous systems. Compared to those studies, our Trojan Horse concentrates on scaling up the performance of modern GPUs, and can be integrated into existing solvers such as SuperLU and PanguLU.

The most commonly used **fundamental patterns of sparse direct solvers** contain the multifrontal method by Duff and Reid [42] and the supernodal method by Li and Demmal [35]. The two methods both find similar structure of columns and update level-2 dense BLAS to level-3 for higher processor utilisation. In contrast, PanguLU [44] preserves sparsity in 2D blocks and calls sparse BLAS for computation. In addition to the numeric methods, other critical components, such as reordering [99], symbolic factorisation [45, 48, 94], out-of-core management [56, 105], sparse triangular solve [12, 71, 73, 75, 76, 83, 107], sparse general matrix-matrix multiplication [24, 72], low-rank [6], batching [53] and performance modeling [25, 28] are also essential components. This paper shows that our Trojan Horse effectively accelerates the numeric phase, and works with both the supernodal and the sparse block methods.

There has been much work on **general scheduling techniques and runtime systems**. The studies are mostly for task representation [97], task placement [4, 61, 80, 81], data partitioning [82, 87, 104], communication [100], task stealing [79, 95], dynamic task scheduling [78, 85, 102], accelerator-aware scheduling [86], reduced synchronization [29, 103], reliability [17, 18], energy efficiency [64], scheduling algorithm

Matrix	SuperLU GPU (w/o Trojan Horse) Time | Perf.	PanguLU GPU (w/o Trojan Horse) Time | Perf.	SuperLU CPU Time | Perf.	MUMPS CPU Time | Perf.	SuperLU GPU (w/ Trojan Horse) Time | Perf.	PanguLU GPU (w/ Trojan Horse) Time | Perf.
cage13	25141 s | 3 GFlops	897 s | 96 GFlops	1143 s | 75 GFlops	**201 s | 428 GFlops**	301 s | 286 GFlops	**157 s | 548 GFlops**
Ga41As41H72	10679 s | 9 GFlops	792 s | 119 GFlops	425 s | 222 GFlops	**141 s | 668 GFlops**	279 s | 338 GFlops	**148 s | 636 GFlops**
RM07R	1157 s | 17 GFlops	197 s | 99 GFlops	92 s | 212 GFlops	**41 s | 476 GFlops**	86 s | 227 GFlops	**35 s | 557 GFlops**
audikw_1	267 s | 43 GFlops	140 s | 83 GFlops	**19 s | 609 GFlops**	29 s | 399 GFlops	65 s | 178 GFlops	**24 s | 482 GFlops**
nlpkkt80	700 s | 41 GFlops	395 s | 72 GFlops	**43 s | 665 GFlops**	Fail	119 s | 240 GFlops	**68 s | 421 GFlops**
Serena	1248 s | 46 GFlops	733 s | 78 GFlops	**81 s | 703 GFlops**	**110 s | 518 GFlops**	150 s | 380 GFlops	112 s | 508 GFlops

Table 7. Performance comparison of sparse direct solvers across six large matrices on an Intel Xeon Gold 6462C CPU and an NVIDIA H100 GPU. The table evaluates execution time (Time in seconds) and floating-point performance (Perf. in GFlops) for SuperLU_DIST, MUMPS and PanguLU. Columns include CPU libraries (SuperLU CPU and MUMPS CPU) and GPU libraries without or with Trojan Horse. The fastest result for each matrix is indicated by an **underlined and bold** font, while the second-fastest result is shown in **bold**. "Fail" indicates a solver's inability to factorise the matrix due to internal errors.

benchmark [96] and autotuning [54]. Also, a number of task-level runtime systems such as PaRSEC [3] and StarPU [15] could enhance sparse factorisation [63, 88]. However, though they can be efficient in dense problems [38], it is hard to effectively merge runtime systems with sparse direct solvers, mainly because that the solvers are too complex to rewrite in a runtime system with general scheduling objectives. Instead, sparse direct solvers often prefer specific scheduling schemes, such as asynchronous scheduling [9, 11], task layout [47], critical path improvement [59, 106], and communication-avoiding methods [91]. In this paper, we show that the Trojan Horse works as a lightweight plug-in to efficiently aggregate and batch a large amount of small tasks inside solvers.

Dense, sparse and batched kernels are fundamental working units of sparse direct solvers. Based on a series of standard interfaces [37], their fast implementations [39] can efficiently run the tasks in sparse direct solvers. Also, the widely studied batched kernels include GEMM [2], dense matrix factorisation [13, 21], as well as sparse computations [58, 77]. In this work, the batched interface in the Trojan Horse is more complete compared to existing work, and demonstrates high utilisation for modern GPUs.

6 Conclusion: Toward the Onset of a Renaissance for GPU-Accelerated Sparse Direct Solvers

In this paper, we have proposed the Trojan Horse strategy to significantly enhance the execution efficiency of sparse direct solvers on modern GPU clusters. We argue that it marks a clear starting point toward a broader Renaissance in GPU-accelerated sparse direct solvers.

Since Duff and Reid introduced the multifrontal method in 1983 [42], sparse direct solvers have pursued the promise of parallelism on CPUs for more than four decades. Over the past twenty years, general-purpose GPU computing has delivered dramatically higher peak floating-point performance and memory bandwidth than CPUs, translating into clear performance gains for bandwidth-intensive iterative methods [19, 43, 65, 74, 108]. However, unfortunately, due to highly interdependent fine-grained tasks, sparse direct methods are far from saturating modern GPUs and have therefore remained less competitive on GPUs than their CPU counterparts, as reflected in the first four data columns of Table 7.

Today, with the introduction of the Trojan Horse strategy, this longstanding barrier has been overcome. The strategy restructures each process's workflow into two stages: Aggregate and Batch, grouping one to two orders of magnitude more small tasks to efficiently saturate high-end GPUs, as demonstrated by both scale-up and scale-out evaluations. The approach is also solver-independent and has been integrated into SuperLU_DIST and PanguLU. For the first time, GPU-accelerated sparse direct solvers finally achieve overall performance comparable to or better than their CPU counterparts, as shown in the last four data columns of Table 7.

We believe that more advanced scheduling techniques and faster kernels can further accelerate sparse direct solvers on GPUs. Therefore, the work presented in this paper serves only as a starting point and opens the door to a broader Renaissance of sparse direct solvers on GPUs.

Data Availability Statement

The artifact for this paper is publicly available on Zenodo [69].

Acknowledgments

We are very grateful to all reviewers for their invaluable comments. Weifeng Liu is the corresponding author of this paper. This work was partially supported by the National Key R&D Program of China (Grant No. 2023YFB3001604), and the National Natural Science Foundation of China (Grant No. 62372467, No. U23A20301, No. 62402525, and No. 92473107). We thank En Shao and Leping Wang for providing us with access to the H100 GPU cluster for testing. We would also like to thank Haocheng Lian and Hongwei Zeng for their help in tuning GPU kernels.

References

[1] A. Abdelfattah, H. Anzt, J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, and A. YarKhan. 2016. Linear Algebra Software for Large-Scale Accelerated Multicore Computing. *Acta Numerica* 25 (2016).

[2] Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. 2020. Matrix Multiplication on Batches of Small Matrices in Half and Half-Complex Precisions. *J. Parallel and Distrib. Comput.* 145 (2020).

[3] Sameh Abdulah, Qinglei Cao, Yu Pei, George Bosilca, Jack. Dongarra, Marc G. Genton, David E. Keyes, Hatem Ltaief, and Ying Sun. 2022. Accelerating Geostatistical Modeling and Prediction With Mixed-Precision Computations: A High-Productivity Approach With PaRSEC. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022).

[4] Gagan Agrawal, Alan Sussman, and Joel Saltz. 1995. An Integrated Runtime and Compile-Time Approach for Parallelizing Structured and Block Structured Applications. *IEEE Transactions on Parallel and Distributed Systems* 6, 7 (1995).

[5] Emmanuel Agullo, Patrick R. Amestoy, Alfredo Buttari, Abdou Guermouche, Jean-Yves L'Excellent, and François-Henry Rouet. 2016. Robust Memory-Aware Mappings for Parallel Multifrontal Factorizations. *SIAM Journal on Scientific Computing* 38, 3 (2016), C256–C279.

[6] Patrick Amestoy, Olivier Boiteau, Alfredo Buttari, Matthieu Gerest, Fabienne Jézéquel, Jean-Yves L'Excellent, and Theo Mary. 2024. Communication Avoiding Block Low-Rank Parallel Multifrontal Triangular Solve with Many Right-Hand Sides. *SIAM J. Matrix Anal. Appl.* 45, 1 (2024), 148–166.

[7] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. 2004. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Software* 30, 3 (2004).

[8] Patrick R. Amestoy, I.S. Duff, and J.-Y. L'Excellent. 2000. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer Methods in Applied Mechanics and Engineering* 184, 2 (2000), 501–520.

[9] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. 2001. A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM J. Matrix Anal. Appl.* 23, 1 (2001).

[10] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. 2001. MUMPS: A General Purpose Distributed Memory Sparse Solver. In *PARA*.

[11] Patrick R. Amestoy, Iain S. Duff, and Christof Vömel. 2004. Task Scheduling in an Asynchronous Distributed Memory Multifrontal Solver. *SIAM J. Matrix Anal. Appl.* 26, 2 (2004).

[12] Patrick R. Amestoy, Jean-Yves L'Excellent, and Gilles Moreau. 2019. On Exploiting Sparsity of Multiple Right-Hand Sides in Sparse Direct Solvers. *SIAM Journal on Scientific Computing* 41, 1 (2019), A269–A291.

[13] Hartwig Anzt, Jack. Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. 2017. Variable-Size Batched LU for Small Matrices and Its Integration into Block-Jacobi Preconditioning. In *ICPP*.

[14] Hartwig Anzt, Axel Huebl, and Xiaoye S. Li. 2024. Then and Now: Improving Software Portability, Productivity, and 100× Performance. *Computing in Science & Engineering* 26, 1 (2024).

[15] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par*.

[16] Roscoe Bartlett, Irina Demeshko, Todd Gamblin, Glenn Hammond, Michael Heroux, Jeffrey Johnson, Alicia Klinvex, Xiaoye Li, Lois McInnes, J. David Moulton, Daniel Osei-Kuffuor, Jason Sarich, Barry Smith, James Willenbring, and Ulrike Meier Yang. 2017. xSDK Foundations: Toward an Extreme-scale Scientific Software Development Kit. *Supercomputing Frontiers and Innovations* 4, 1 (2017).

[17] Michael Bauer, Wonchan Lee, Elliott Slaughter, Zhihao Jia, Mario Di Renzo, Manolis Papadakis, Galen Shipman, Patrick McCormick, Michael Garland, and Alex Aiken. 2021. Scaling implicit parallelism via dynamic control replication. In *PPoPP*.

[18] Michael Bauer, Elliott Slaughter, Sean Treichler, Wonchan Lee, Michael Garland, and Alex Aiken. 2023. Visibility Algorithms for Dynamic Dependence Analysis and Distributed Coherence. In *PPoPP*.

[19] Nathan Bell and Michael Garland. 2009. CUSP: Generic Parallel Algorithms for Sparse Matrix and Graph Computations.

[20] Noel Chalmers, Jakub Kurzak, Damon Mcdougall, and Paul Bauman. 2023. Optimizing High-Performance Linpack for Exascale Accelerated Architectures. In *SC*.

[21] Ali Charara, David Keyes, and Hatem Ltaief. 2019. Batched Triangular Dense Linear Algebra Kernels for Very Small Matrix Sizes on GPUs. *ACM Trans. Math. Software* 45, 2 (2019).

[22] Xiaoming Chen, Yu Wang, and Huazhong Yang. 2013. NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 2 (2013).

[23] Yanqing Chen, Timothy A. Davis, William W. Hager, and Sivasankaran Rajamanickam. 2008. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. *ACM Trans. Math. Software* 35, 3 (2008).

[24] Helin Cheng, Wenxuan Li, Yuechen Lu, and Weifeng Liu. 2023. HASpGEMM: Heterogeneity-Aware Sparse General Matrix-Matrix Multiplication on Modern Asymmetric Multicore Processors. In *ICPP*.

[25] Younghyun Cho, Surim Oh, and Bernhard Egger. 2020. Performance Modeling of Parallel Loops on Multi-Socket Platforms Using Queueing Systems. *IEEE Transactions on Parallel and Distributed Systems* 31, 2 (2020).

[26] Lisa Claus, Pieter Ghysels, Wajih Halim Boukaram, and Xiaoye Sherry Li. 2025. A graphics processing unit accelerated sparse direct solver and preconditioner with block low rank compression. *The International Journal of High Performance Computing Applications* 39, 1 (2025).

[27] Michel Cosnard and Laura Grigori. 2001. A parallel algorithm for sparse symbolic LU factorization without pivoting on out-of-core matrices. In *ICS*.

[28] Thanh Tuan Dao, Jungwon Kim, Sangmin Seo, Bernhard Egger, and Jaejin Lee. 2015. A Performance Model for GPUs with Caches. *IEEE Transactions on Parallel and Distributed Systems* 26, 7 (2015).

[29] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Vishwesh Jatala, Keshav Pingali, V. Krishna Nandivada, Hoang-Vu Dang, and Marc Snir. 2024. Gluon-Async: A Bulk-Asynchronous System for Distributed and Heterogeneous Graph Analytics. In *PACT*.

[30] Timothy A. Davis. 2004. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Software* 30, 2 (2004).

[31] Timothy A. Davis. 2006. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics.

[32] Timothy A. Davis. 2011. Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization. *ACM Trans. Math. Software* 38, 1 (2011).

[33] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1 (2011).

[34] Timothy A. Davis and Ekanathan Palamadai Natarajan. 2010. Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems. *ACM Trans. Math. Software* 37, 3 (2010).

[35] 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).

[36] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. 1999. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIAM J. Matrix Anal. Appl.* 20, 4 (1999).

[37] Jack. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software* 16, 1 (1990).

[38] Jack. Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Panruo Wu, Ichitaro Yamazaki, Asim Yarkhan, Maksims Abalenkovs, Negin Bagherpour, Sven Hammarling, Jakub Šístek, David Stevens, Mawussi Zounon, and Samuel D. Relton. 2019. PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP. *ACM Trans. Math. Software* 45, 2 (2019).

[39] Jack. Dongarra, Mark Gates, Jakub Kurzak, Piotr Luszczek, and Yaohung M. Tsai. 2018. Autotuning Numerical Dense Linear Algebra for Batched Computation With GPU Hardware Accelerators. *Proc. IEEE* 106, 11 (2018).

[40] Iain S. Duff. 1986. Parallel implementation of multifrontal schemes. *Parallel Comput.* 3, 3 (1986).

[41] Iain S. Duff, Albert M. Erisman, and John K. Reid. 2017. *Direct Methods for Sparse Matrices.* Oxford University Press.

[42] Iain S. Duff and John K. Reid. 1983. The Multifrontal Solution of Indefinite Sparse Symmetric Linear. *ACM Trans. Math. Software* 9, 3 (1983).

[43] Robert D. Falgout, Ruipeng Li, Björn Sjögreen, Lu Wang, and Ulrike Meier Yang. 2021. Porting hypre to heterogeneous computer architectures: Strategies and experiences. *Parallel Comput.* 108 (2021).

[44] Xu Fu, Bingbin Zhang, Tengcheng Wang, Wenhao Li, Yuechen Lu, Enxin Yi, Jianqi Zhao, Xiaohan Geng, Fangying Li, Jingwen Zhang, Zhou Jin, and Weifeng Liu. 2023. PanguLU: A Scalable Regular Two-Dimensional Block-Cyclic Sparse Direct Solver on Distributed Heterogeneous Systems. In *SC*.

[45] Anil Gaihre, Xiaoye Sherry Li, and Hang Liu. 2022. gSoFa: Scalable Sparse Symbolic LU Factorization on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022).

[46] Mark S. Gockenbach. 2006. *Understanding and Implementing the Finite Element Method.* Society for Industrial and Applied Mathematics.

[47] Changjiang Gou, Ali Al Zoobi, Anne Benoit, Mathieu Faverge, Loris Marchal, Grégoire Pichon, and Pierre Ramet. 2020. Improving Mapping for Sparse Direct Solvers. In *Euro-Par*.

[48] Laura Grigori, John R. Gilbert, and Michel Cosnard. 2009. Symbolic and Exact Structure Prediction for Sparse Gaussian Elimination with Partial Pivoting. *SIAM J. Matrix Anal. Appl.* 30, 4 (2009).

[49] Laura Grigori and Xiaoye S. Li. 2002. A new scheduling algorithm for parallel sparse LU factorization with static pivoting. In *SC*.

[50] Azzam Haidar, Ahmad Abdelfattah, Mawussi Zounon, Stanimire Tomov, and Jack. Dongarra. 2018. A Guide for Achieving High Performance with Very Small Matrices on GPU: A Case Study of Batched LU and Cholesky Factorizations. *IEEE Transactions on Parallel and Distributed Systems* 29, 5 (2018).

[51] Pascal Hénon, Pierre Ramet, and Jean Roman. 2002. On finding approximate supernodes for an efficient ILU(k) factorization. *Parallel Comput.* 34, 6-8 (2002), 345–362.

[52] Pascal Hénon, Pierre Ramet, and Jean Roman. 2002. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Comput.* 28, 2 (2002), 301–321.

[53] Michael Heroux, Wajih Boukaram, Yuxi Hong, Yang Liu, Tianyi Shi, and Xiaoye S Li. 2024. Batched sparse direct solver design and evaluation in SuperLU_DIST. *The International Journal of High Performance Computing Applications* 38, 6 (2024).

[54] Shintaro Iwasaki and Kenjiro Taura. 2016. Autotuning of a Cut-Off for Task Parallel Programs. In *MCSoC*.

[55] Zhou Jin, Wenhao Li, Yinuo Bai, Tengcheng Wang, Yicheng Lu, and Weifeng Liu. 2024. Machine Learning and GPU Accelerated Sparse Linear Solvers for Transistor-Level Circuit Simulation: A Perspective Survey (Invited Paper). In *ASP-DAC*.

[56] Changyeon Jo, Hyunik Kim, Hexiang Geng, and Bernhard Egger. 2020. RackMem: A Tailored Caching Layer for Rack Scale Computing. In *PACT*.

[57] M. Ozan Karsavuran, Esmond G. Ng, and Barry W. Peyton. 2025. GPU Accelerated Sparse Cholesky Factorization. In *SC*.

[58] Aditya Kashi, Pratik Nayak, Dhruva Kulkarni, Aaron Scheinberg, Paul Lin, and Hartwig Anzt. 2022. Batched sparse iterative solvers on GPU for the collision operator for fusion plasma simulations. In *IPDPS*.

[59] Enver Kayaaslan and Bora Uçar. 2014. Reducing elimination tree height for parallel LU factorization of sparse unsymmetric matrices. In *HiPC*.

[60] Tzanio Kolev, Paul Fischer, Misun Min, Jack Dongarra, Jed Brown, Veselin Dobrev, Tim Warburton, Stanimire Tomov, Mark S Shephard, Ahmad Abdelfattah, Valeria Barra, Natalie Beams, Jean-Sylvain Camier, Noel Chalmers, Yohann Dudouit, Ali Karakus, Ian Karlin, Stefan Kerkemeier, Yu-Hsiang Lan, David Medina, Elia Merzari, Aleksandr Obabko, Will Pazner, Thilina Rathnayake, Cameron W Smith, Lukas Spies, Kasia Swirydowicz, Jeremy Thompson, Ananias Tomboulides, and Vladimir Tomov. 2021. Efficient exascale discretizations: High-order finite element methods. *The International Journal of High Performance Computing Applications* 35, 6 (2021).

[61] Milind Kulkarni, Patrick Carribault, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. 2008. Scheduling strategies for optimistic parallel execution of irregular programs. In *SPAA*.

[62] Jakub Kurzak, Hartwig Anzt, Mark Gates, and Jack. Dongarra. 2016. Implementation and Tuning of Batched Cholesky Factorization and Solve for NVIDIA GPUs. *IEEE Transactions on Parallel and Distributed Systems* 27, 7 (2016).

[63] Xavier Lacoste, Mathieu Faverge, George Bosilca, Pierre Ramet, and Samuel Thibault. 2014. Taking Advantage of Hybrid Systems for Sparse Direct Solvers via Task-Based Runtimes. In *IPDPS*.

[64] Dong Li, Bronis R de Supinski, Martin Schulz, Kirk Cameron, and Dimitrios S. Nikolopoulos. 2010. Hybrid MPI/OpenMP power-aware computing. In *IPDPS*.

[65] Ruipeng Li and Yousef Saad. 2013. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing* 63 (2013).

[66] Xiaoye S. Li. 2005. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Software* 31, 3 (2005).

[67] Xiaoye S. Li and James W. Demmel. 2003. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Software* 29, 2 (2003).

[68] 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. Software* 49, 1 (2023).

[69] Yida Li. 2025. Trojan Horse: Aggregate-and-Batch for Scaling Up Sparse Direct Solvers on GPU Clusters. doi:10.5281/zenodo.17706095

[70] Alycia Lisito, Mathieu Faverge, Grégoire Pichon, and Pierre Ramet. 2024. Enhancing Sparse Direct Solver Scalability Through Runtime System Automatic Data Partition. In *WAMTA*.

[71] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S. Duff, and Brian Vinter. 2016. A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves. In *Euro-Par*.

[72] Weifeng Liu and Brian Vinter. 2014. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *IPDPS*.

[73] Yang Liu, Nan Ding, Piyush Sao, Samuel Williams, and Xiaoye Sherry Li. 2023. Unified Communication Optimization Strategies for Sparse Triangular Solver on CPU and GPU Clusters. In *SC*.

[74] Yuechen Lu, Lijie Zeng, Tengcheng Wang, Xu Fu, Wenxuan Li, Helin Cheng, Dechuang Yang, Zhou Jin, Marc Casas, and Weifeng Liu. 2024. AmgT: Algebraic Multigrid Solver on Tensor Cores. In *SC*.

[75] Zhengyang Lu and Weifeng Liu. 2023. TileSpTRSV: a tiled algorithm for parallel sparse triangular solve on GPUs. *CCF Transactions on High Performance Computing* 5, 2, 129–143.

[76] Zhengyang Lu, Yuyao Niu, and Weifeng Liu. 2020. Efficient Block Algorithms for Parallel Sparse Triangular Solve. In *ICPP*.

[77] Piotr Luszczek, Ahmad Abdelfattah, Hartwig Anzt, Atsushi Suzuki, and Stanimire Tomov. 2024. Batched sparse and mixed-precision

linear algebra interface for efficient use of GPU hardware accelerators in scientific applications. *Future Generation Computer Systems* 160 (2024).

[78] Robert L. McGregor, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. 2005. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *IPDPS*.

[79] Prasoon Mishra and V. Krishna Nandivada. 2024. COWS for High Performance: Cost Aware Work Stealing for Irregular Parallel Loop. *ACM Transactions on Architecture and Code Optimization* 21, 1 (2024).

[80] Frank Mueller. 2000. Distributed Shared-Memory Threads: DSM-Threads. In *Workshop on Run-Time Systems for Parallel Programming*.

[81] V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. 2013. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Transactions on Programming Languages and Systems* 35, 1 (2013).

[82] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. 2000. Is Data Distribution Necessary in OpenMP?. In *SC*.

[83] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: a tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *PPoPP*.

[84] Maciej Paszyński, David Pardo, Carlos Torres-Verdín, Leszek Demkowicz, and Victor Calo. 2010. A parallel direct solver for the self-adaptive hp Finite Element Method. *J. Parallel and Distrib. Comput.* 70, 3 (2010).

[85] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *PLDI*.

[86] Vignesh T. Ravi, Michela Becchi, Wei Jiang, Gagan Agrawal, and Srimat Chakradhar. 2012. Scheduling Concurrent Applications on a Cluster of CPU-GPU Nodes. In *CCGRID*.

[87] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *EuroSys*.

[88] Jie Ren, Tingxuan Zhong, Yuxi Hong, Guofeng Feng, Xincheng Wang, Weile Jia, Hatem Ltaief, and David Elliot Keyes. 2025. Caracal: A GPU-Resident Sparse LU Solver with Lightweight Fine-Grained Scheduling. In *SC*.

[89] Ponnuswamy. Sadayappan and V. Visvanathan. 1989. Efficient sparse matrix factorization for circuit simulation on vector supercomputers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 8, 12 (1989).

[90] Piyush Sao, Ramakrishnan Kannan, Xiaoye Sherry Li, and Richard Vuduc. 2019. A communication-avoiding 3D sparse triangular solver. In *ICS*.

[91] Piyush Sao, Xiaoye S. Li, and Richard Vuduc. 2019. A communication-avoiding 3D algorithm for sparse LU factorization on heterogeneous systems. *J. Parallel and Distrib. Comput.* 131 (2019).

[92] Piyush Sao, Xing Liu, Richard Vuduc, and Xiaoye Li. 2015. A Sparse Direct Solver for Distributed Memory Xeon Phi-Accelerated Systems. In *IPDPS*.

[93] Olaf Schenk and Klaus Gärtner. 2004. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems* 20, 3 (2004).

[94] Oguz Selvitopi, Xiaoye S. Li, and Aydin Buluc. 2025. Performance-Portable Symbolic Factorization through Common Graph Operations. In *SC Workshop*.

[95] Shumpei Shiina and Kenjiro Taura. 2022. Distributed Continuation Stealing is More Scalable than You Might Think. In *CLUSTER*.

[96] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S. Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, Wonchan Leek, Sean Treichlerk, Patrick McCormick, and Alex Aiken. 2020. Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance. In *SC*.

[97] Rupanshu Soi, Michael Bauer, Sean Treichler, Manolis Papadakis, Wonchan Lee, Patrick McCormick, Alex Aiken, and Elliott Slaughter. 2021. Index launches: scalable, flexible representation of parallel task groups. In *SC*.

[98] Anne Trefethen, Nick Higham, Iain Duff, and Peter Coveney. 2009. Developing a High-Performance Computing/Numerical Analysis Roadmap. *The International Journal of High Performance Computing Applications* 23, 4 (2009).

[99] James D. Trotter, Sinan Ekmekçibaşı, Johannes Langguth, Tugba Torun, Emre Düzakın, Aleksandar Ilic, and Didem Unat. 2023. Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs. In *SC*.

[100] Jeffrey S. Vetter and Frank Mueller. 2002. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. In *IPDPS*.

[101] 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 *DAC*.

[102] Yusheng Weijiang, Shruthi Balakrishna, Jianqiao Liu, and Milind Kulkarni. 2015. Tree dependence analysis. In *PLDI*.

[103] Matthew Whitlock, Hemanth Kolla, Sean Treichler, Philippe Pébay, and Janine C. Bennett. 2018. Scalable Collectives for Distributed Asynchronous Many-Task Runtimes. In *IPDPSW*.

[104] Kai Wu, Jie Ren, and Dong Li. 2018. Runtime Data Management on Non-Volatile Memory-based Heterogeneous Memory for Task-Parallel Programs. In *SC*.

[105] Yang Xia, Peng Jiang, Gagan Agrawal, and Rajiv Ramnath. 2023. End-to-End LU Factorization of Large Matrices on GPUs. In *PPoPP*.

[106] Ichitaro Yamazaki and Xiaoye S. Li. 2012. New Scheduling Strategies and Hybrid Programming for a Parallel Right-looking Sparse LU Factorization Algorithm on Multicore Cluster Systems. In *IPDPS*.

[107] Ichitaro Yamazaki, Sivasankaran Rajamanickam, and Nathan Ellingwood. 2020. Performance Portable Supernode-based Sparse Triangular Solver for Manycore Architectures. In *ICPP*.

[108] Dechuang Yang, Yuxuan Zhao, Yiduo Niu, Weile Jia, En Shao, Weifeng Liu, Guangming Tan, and Zhou Jin. 2024. Mille-feuille: A Tile-Grained Mixed Precision Single-Kernel Conjugate Gradient Solver on GPUs. In *SC*.

[109] Sencer Nuri Yeralan, Timothy A. Davis, Wissam M. Sid-Lakhdar, and Sanjay Ranka. 2017. Algorithm 980: Sparse QR Factorization on the GPU. *ACM Trans. Math. Software* 44, 2 (2017).

[110] Jianqi Zhao, Yao Wen, Yuchen Luo, Zhou Jin, Weifeng Liu, and Zhenya Zhou. 2022. SFLU: Synchronization-Free Sparse LU Factorization for Fast Circuit Simulation on GPUs. In *DAC*.

[111] Kasia Świrydowicz, Eric Darve, Wesley Jones, Jonathan Maack, Shaked Regev, Michael A. Saunders, Stephen J. Thomas, and Slaven Peleš. 2022. Linear solvers for power grid optimization problems: A review of GPU-accelerated linear solvers. *Parallel Comput.* 111 (2022).

Artifact Description

This artifact for our submission #231, entitled "Trojan Horse: Aggregate-and-Batch for Scaling Up Sparse Direct Solvers on GPU Clusters," includes the following components: (1) Prerequisites, introducing the hardware and software requirements; (2) Introduction to Each Figure, analyzing data and trends from each figure in our experimental results; and (3) Step-by-Step Instructions, a detailed evaluation guide for reproducing all experiments, available in either a 30-minute fast mode or a full mode.

1 Artifact DOI

https://doi.org/10.5281/zenodo.17706095

2 Docker Image

https://hub.docker.com/repository/docker/jakebomber/trojan_horse_ae/tags/release_v1

3 Prerequisites

Quick Guide: Running AE needs x86_64 machines with 5060Ti, 5090 and 8-card A100 respectively, with NVIDIA driver >= 570.124.06, Docker 28 with NVIDIA Container Toolkit. Require 128GB main memory for 5060Ti and 5090, and 256GB for 8-card A100. Require 200GB disk space. Network is recommended for convenience.

- **Libraries and Versions**.
  - Intel MPI 2021 or above.
  - OpenBLAS 0.3.26 or above.
  - NVIDIA GPU Driver 570.124.06 or above.
  - NVIDIA CUDA Toolkit 12.4 or above.
  - ParMETIS 4.0.3 or above, METIS 5.1.0 or above.
  - Build Tools: CMake 3.2.0 or above, GCC 11.0.0 or above, GFortran 11.0.0 or above.
  - Python Libraries: Python 3.10.16, pandas 2.2.3, matplotlib 3.9.1, numpy 2.1.3, scipy 1.15.2, openpyxl 3.1.0.

  All of these libraries are prepared in our provided Docker image.
- **Hardware Requirements**.
  - **GPU**: For scale-up evaluation: NVIDIA RTX 5060Ti, NVIDIA RTX 5090 and NVIDIA A100 80G; For scale-out evaluation: 8-card NVIDIA A100 80G GPU cluster.
  - **Disk Space**: About 200GB, sufficient space to store about 200 test matrices and all programs.
  - **Memory**: 128GB for single GPUs and 256GB for clusters.
- **Matrices**. 200 matrices from SuiteSparse Matrix Collection (publicly available at https://sparse.tamu.edu) are needed in total, and all of them are included in our Docker image.

4 Step-by-Step Instructions

Quick Guide: Please run Docker image "trojan_horse_ae_lite_matrices.docker.img" on 5060Ti, 5090 and 8-card A100 respectively. In "/ppopp26trojanhorseae" of the Docker container, running script "<AE_FAST=1> bash run_<machine>_docker.sh" to get results in directory "/ppopp26trojanhorseae/results". Without or with "AE_FAST=1" before the command line represents full or 30-minutes-fast AE execution. If network is available, each machine automatically upload results to a configured SSH server. Running "bash generate_all_figures_online.sh", the results will be automatically fetched and merged, then generate all figures into "/ppopp26trojanhorseae/figures".
Execution Time Estimation: We provided 30-minutes-fast mode for tests on each machine (5060Ti, 5090 and 8-card A100). If fast mode is disabled, the execution time will be about 2h, 3h and 120h on 5060Ti, 5090 and 8-card A100 respectively.

4.1 Checking Docker Version (28 or above) and NVIDIA CUDA Driver Version (570.124.06 or above)

$ docker –version
$ nvidia-smi

4.2 Confirming the Processor Architecture (x86_64)

$ uname -m

4.3 Running Docker: Importing Images and Launching Containers

You can directly run
$ sudo docker pull jakebomber/trojan_horse_ae:release_v1
on your servers to import the Docker image.

4.4 Running the Code Inside the Docker Container

Once inside the container, you can list the files in the current directory using:
$ ls
You would see a directory named `ppopp26trojanhorseae`. Enter this directory.
$ cd ppopp26trojanhorseae
Running Scripts on Different Machines. Please note that above operations are same on three machines. The following operations are different on three machines.

- **On the 5060Ti:**
  If you want to run the script with the 30-minutes-fast mode, use:
  $ AE_FAST=1 bash run_5060Ti_docker.sh
  In fast mode, ensure the output contains: *Fast AE mode is enabled (about 25 minutes).*
  If you want to run the full matrix set, use:
  $ bash run_5060Ti_docker.sh

Ensure the output contains: *The Full AE would be very slow (about 2 hours).*

- **On the 5090:**
  To run the the script with the 30-minutes-fast mode, use:
  $ AE_FAST=1 bash run_5090_docker.sh
  Ensure the output contains: *Fast AE mode is enabled (about 30 minutes).*
  To run the full matrix set, use:
  $ bash run_5090_docker.sh
  Ensure the output contains: *The Full AE would be very slow (about 3 hours).*
- **On the 8-card A100:**
  To run the script with the 30-minutes-fast mode, use:
  $ AE_FAST=1 bash run_A100_docker.sh
  Ensure the output contains: *Fast AE mode is enabled (about 30 minutes).*
  To run the full matrix set, use:
  $ bash run_A100_docker.sh
  Ensure the output contains: *The Full AE would be very slow (about 120 hours).*

4.5 Collecting and Merging CSV Results from Multiple Machines

Please to manually rename the `results` directories generated on each machine and copy to `/ppopp26trojanhorseae` inside the Docker container on the 5060Ti.

First, outside the Docker container on the 5090 and A100 respectively, copy the `results` folder from inside the container to the outside host machine:

$ sudo docker cp TrojanHorse:/ppopp26trojanhorse/results ./results

Next, on your computer, copy the `results` folders from the 5090 and A100 machines to the 5060Ti host:

$ scp -r user@5090:~/results user@5060Ti:~/results_5090

$ scp -r user@A100:~/results user@5060Ti:~/results_A100

Connect to the 5060Ti machine:

$ ssh user@5060Ti

Check the data copied earlier from the 5090 and A100 machines:

$ ls

Copy the data into the Docker container:

$ sudo docker cp results_5090 TrojanHorse:/ppopp26trojanhorseae/

$ sudo docker cp results_A100 TrojanHorse:/ppopp26trojanhorseae/

Re-enter the Docker container:

$ sudo docker start TrojanHorse

$ sudo docker attach TrojanHorse

4.6 Generating Figures

The script will automatically merge `results`, `results_5090`, and `results_A100`, then generate figures:

$ bash generate_all_figures_offline.sh

Copy the generated figures out of Docker and download them to your local machine.

Outside the 5060Ti Docker, on the host:

$ sudo docker cp TrojanHorse:/ppopp26trojanhorseae/figures .

On your computer:

$ scp -r user@5060Ti:~/figures .

And the figures will be downloaded to your computer.