PPoPP 2026

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

Yida Li, Siwei Zhang, Yiduo Niu, Yang Du, Qingxiao Sun, Zhou Jin, Weifeng Liu

Super Scientific Software Laboratory (SSSLab)

China University of Petroleum-Beijing

Sydney, Australia

Feb. 3, 2026

Outline

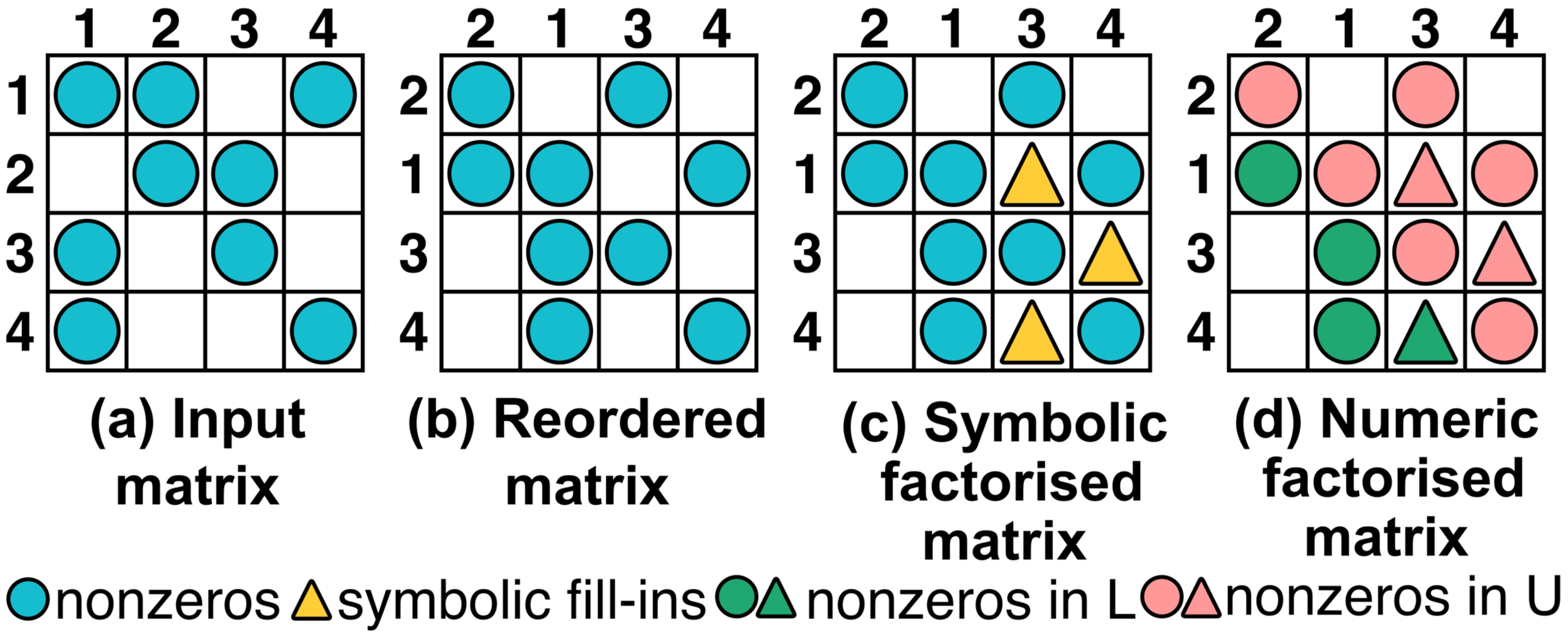
- Background
  1. Sparse LU Factorisation
  2. Task Dependencies Restrict Concurrency
  3. Single Task Is Too Small For a GPU
- Motivations
  1. Aggregate: to Prepare More Tasks for a GPU
  2. Batch: to Selectively Run the Tasks in Parallel
- Trojan Horse
  1. Overview
  2. An Example to Use the Trojan Horse
  3. Functional Modules of the Trojan Horse (Priortizer, Container, Collector & Executor)
- Experiments
  1. Experimental Setup
  2. Scale-Up Evaluation
  3. Scale-Out Evaluation
  4. Comparison with CPU solvers
- Conclusion

Outline

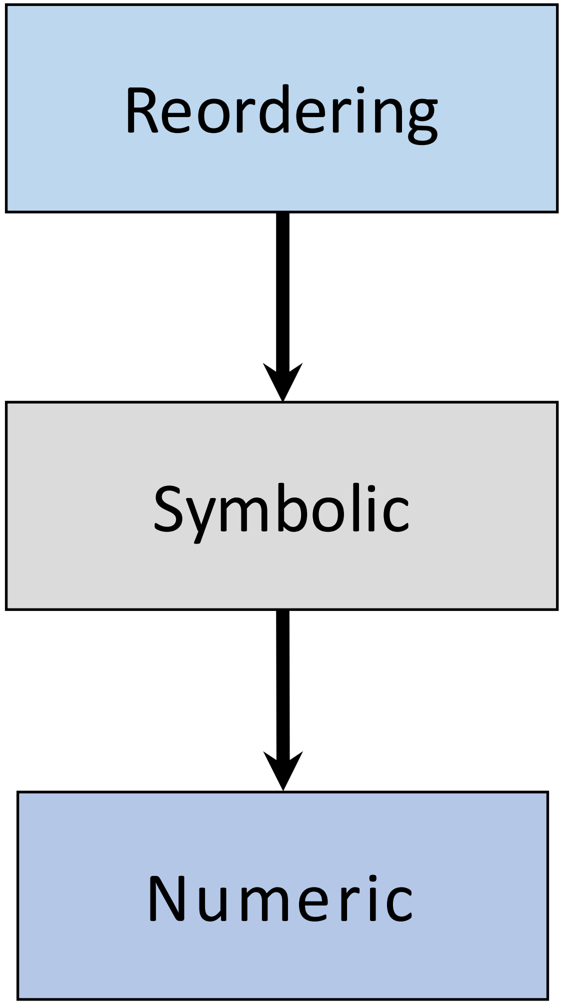
- Background
  ① Sparse LU Factorisation
  ② Task Dependencies Restrict Concurrency
  ③ Single Task Is Too Small For a GPU
- Motivations
  ① Aggregate: to Prepare More Tasks for a GPU
  ② Batch: to Selectively Run the Tasks in Parallel
- Trojan Horse
  ① Overview
  ② An Example to Use the Trojan Horse
  ③ Functional Modules of the Trojan Horse (Priortizer, Container, Collector & Executor)
- Experiments
  ① Experimental Setup
  ② Scale-Up Evaluation
  ③ Scale-Out Evaluation
  ④ Comparison with CPU solvers
- Conclusion

Three Phases of Sparse LU Factorisation

Sparse LU factorisation includes three major phases: reordering, symbolic and numeric factorisation.



Three Phases of Sparse LU Factorisation



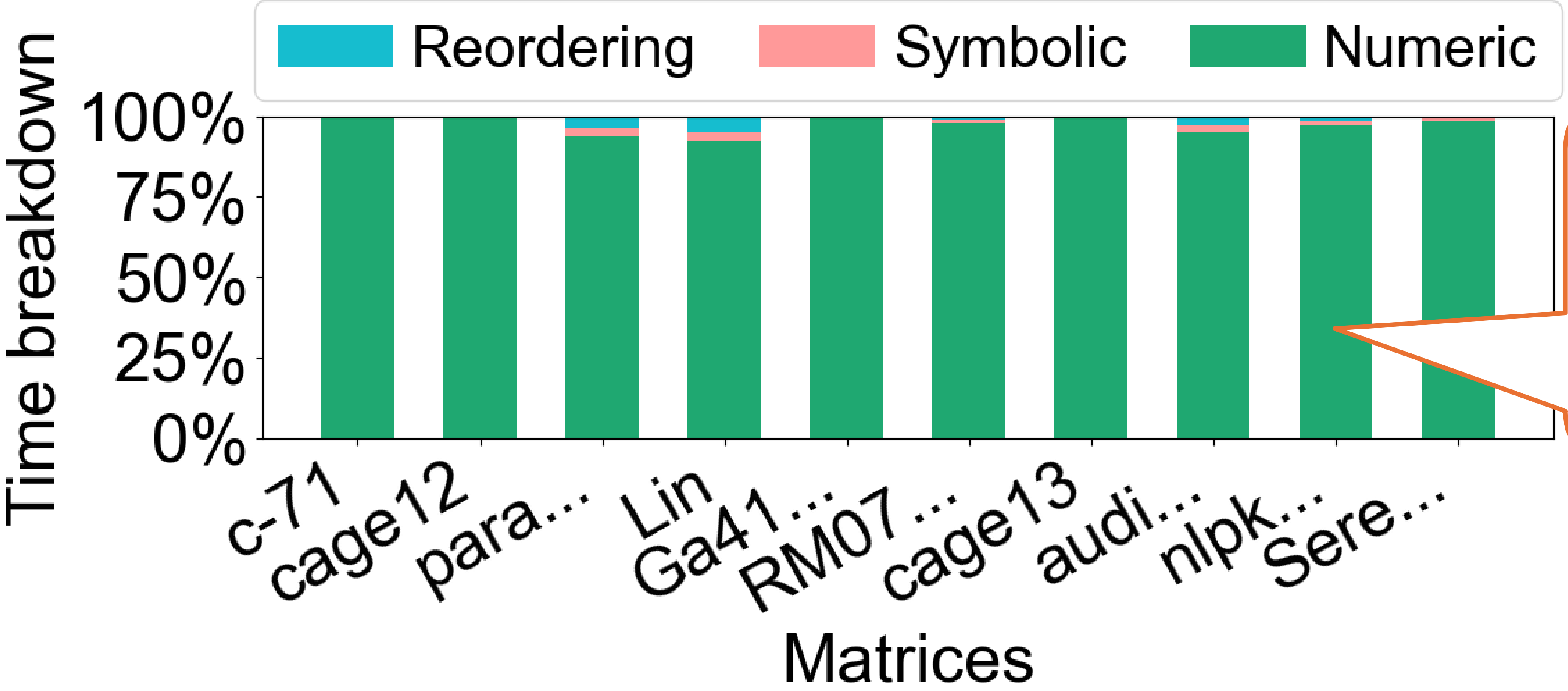
The reordering phase aims to permute the matrix *A* to reduce fill-in elements.

The symbolic factorisation phase identifies the structures of the sparse factor matrices *L* and *U*.

The numeric factorisation phase determines the value of *L* and *U*, which is generally the only stage processing a large amount of floating point operations.

Time Breakdown of Sparse LU factorisation

The numeric phase spends most of the time, which motivates us to investigate a strategy for optimising the numeric phase on heterogeneous GPU clusters.



The numeric factorisation phase spends most execution time, on average 97%, and is almost the only phase that scales to a large amount of compute nodes.

Sparse direct solver: SuperLU 9.1.0
CPU: AMD Ryzen 9 9950X (one core)

Outline

- Background
  - ① Sparse LU Factorisation
  - ② Task Dependencies Restrict Concurrency
  - ③ Single Task Is Too Small For a GPU
- Motivations
  - ① Aggregate: to Prepare More Tasks for a GPU
  - ② Batch: to Selectively Run the Tasks in Parallel
- Trojan Horse
  - ① Overview
  - ② An Example to Use the Trojan Horse
  - ③ Functional Modules of the Trojan Horse (Priortizer, Container, Collector & Executor)
- Experiments
  - ① Experimental Setup
  - ② Scale-Up Evaluation
  - ③ Scale-Out Evaluation
  - ④ Comparison with CPU solvers
- Conclusion

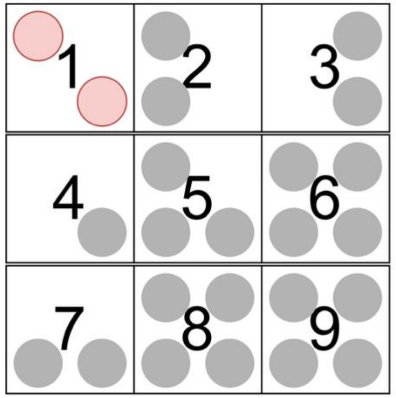
Task Dependencies Restrict Concurrency

There are three task types (LU factorisation, triangular solve and Schur complement) in sparse LU factorisation. This figure shows the task dependency of factorising a 6-by-6 blocked matrix.

F LU Factorisation

T Triangular Solve

S Schur Complement



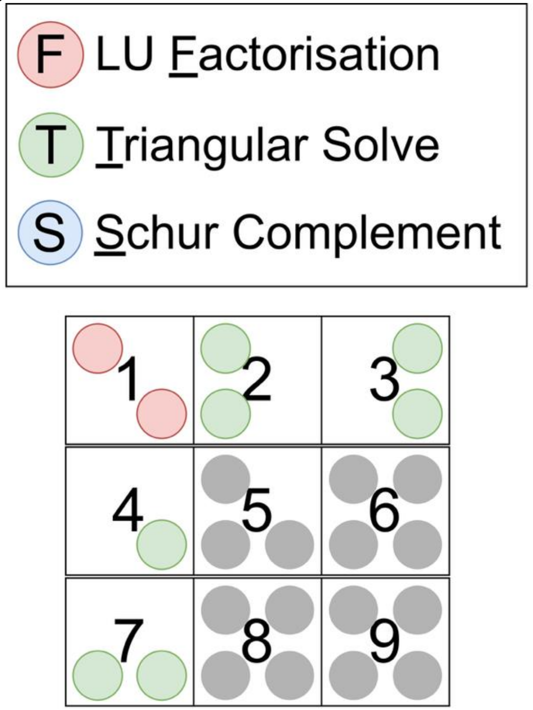
Task Number Statics
LU factorisation x3
Triangular solve x6
Schur complement x5

Fristly, task ‘1F’ starts.

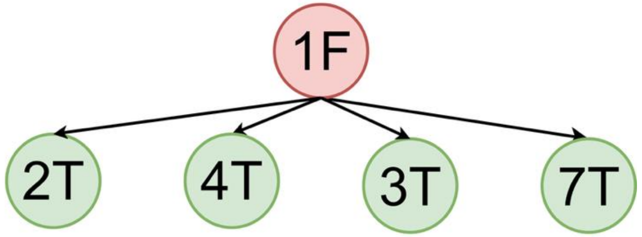
Task Dependencies Restrict Concurrency

超级科学软件实验室
Super Scientific Software Laboratory

There are three task types (LU factorisation, triangular solve and Schur complement) in sparse LU factorisation. This figure shows the task dependency of factorising a 6-by-6 blocked matrix.



Task Count Statistics
LU factorisation x3
Triangular solve x6
Schur complement x5

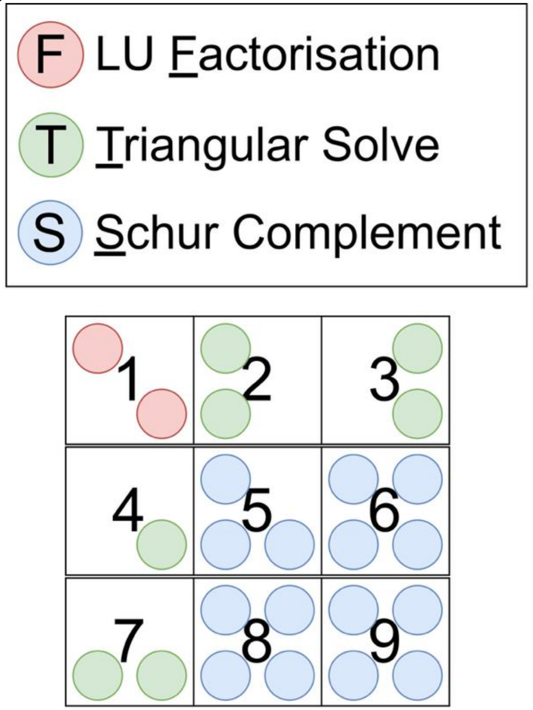


Fristly, task '1F' starts.

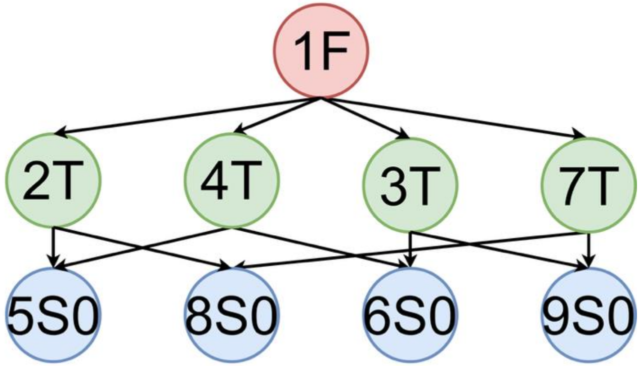
Then, four triangular solve tasks can be executed, depending on the result of '1F'.

Task Dependencies Restrict Concurrency

There are three task types (LU factorisation, triangular solve and Schur complement) in sparse LU factorisation. This figure shows the task dependency of factorising a 6-by-6 blocked matrix.



Task Count Statistics
LU factorisation x3
Triangular solve x6
Schur complement x5



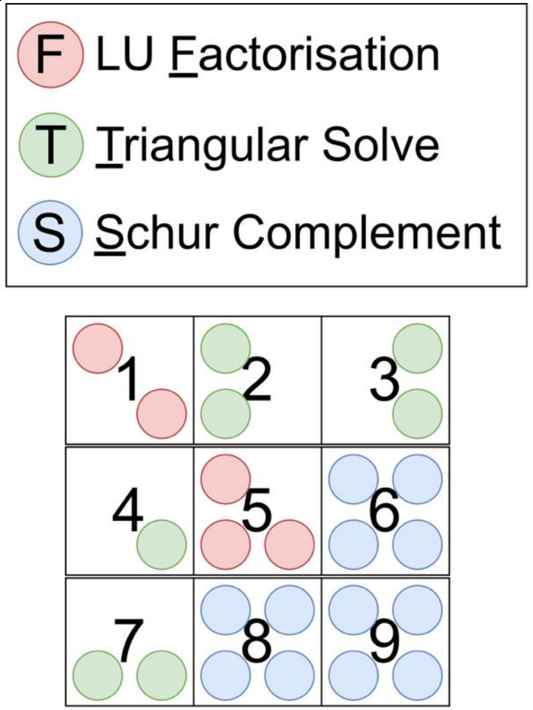
Fristly, task ‘1F’ starts.

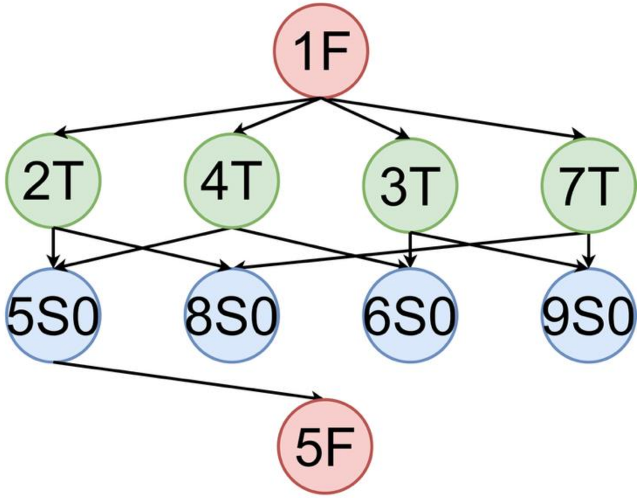
Then, four triangular solve tasks can be executed, depending on the result of ‘1F’.

After this, four Schur complement tasks can be executed, depending on previous results.

Task Dependencies Restrict Concurrency

There are three task types (LU factorisation, triangular solve and Schur complement) in sparse LU factorisation. This figure shows the task dependency of factorising a 6-by-6 blocked matrix.





Task Count Statistics
LU factorisation x3
Triangular solve x6
Schur complement x5

Fristly, task ‘1F’ starts.

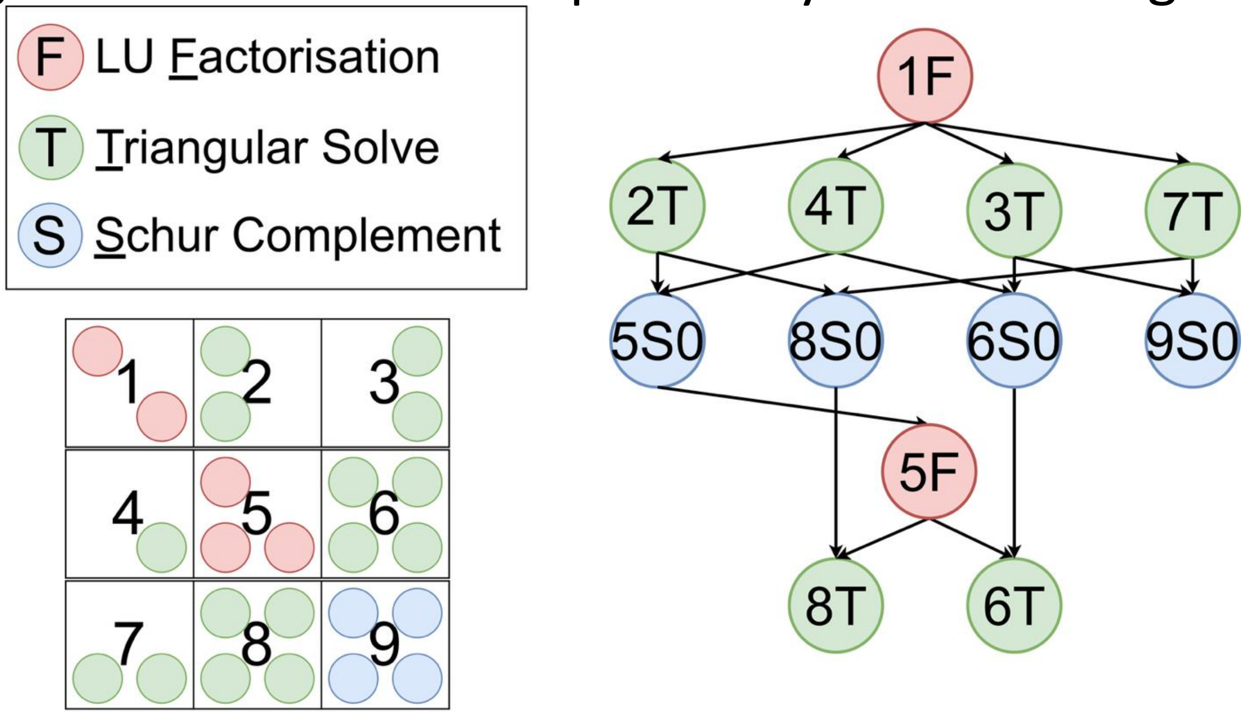
Then, four triangular solve tasks can be executed, depending on the result of ‘1F’.

After this, four Schur complement tasks can be executed, depending on previous results.

The dependencies between tasks are complex in sparse LU factorisation, which restricts concurrency.

Task Dependencies Restrict Concurrency

There are three task types (LU factorisation, triangular solve and Schur complement) in sparse LU factorisation. This figure shows the task dependency of factorising a 6-by-6 blocked matrix.



Task Count Statistics
LU factorisation x3
Triangular solve x6
Schur complement x5

Fristly, task '1F' starts.

Then, four triangular solve tasks can be executed, depending on the result of '1F'.

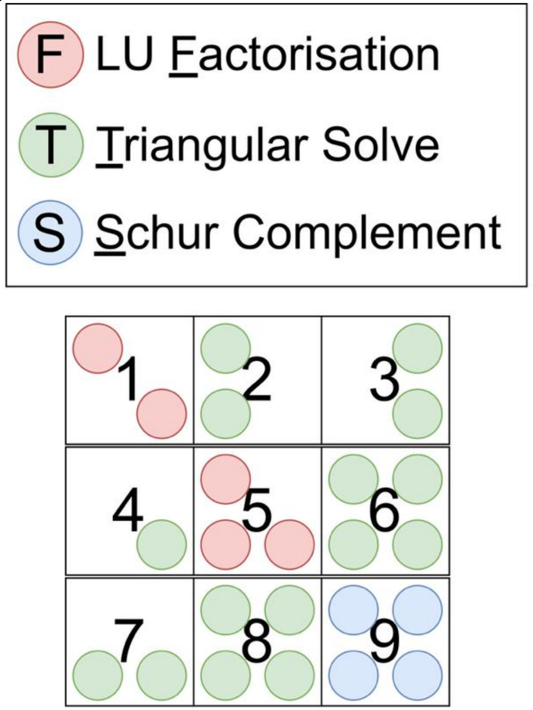
After this, four Schur complement tasks can be executed, depending on previous results.

The dependencies between tasks are complex in sparse LU factorisation, which restricts concurrency.

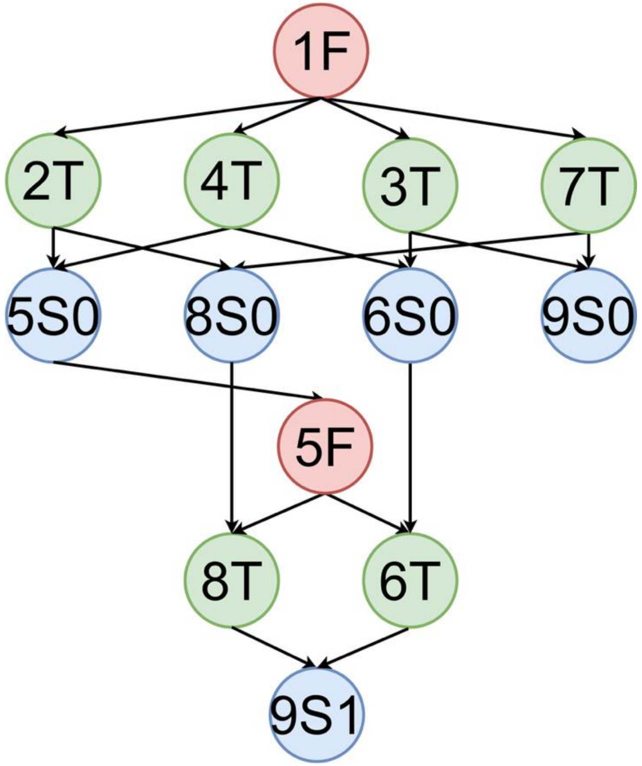
Task Dependencies Restrict Concurrency

超级科学软件实验室
Super Scientific Software Laboratory

There are three task types (LU factorisation, triangular solve and Schur complement) in sparse LU factorisation. This figure shows the task dependency of factorising a 6-by-6 blocked matrix.



Task Count Statistics
LU factorisation x3
Triangular solve x6
Schur complement x5



Fristly, task '1F' starts.

Then, four triangular solve tasks can be executed, depending on the result of '1F'.

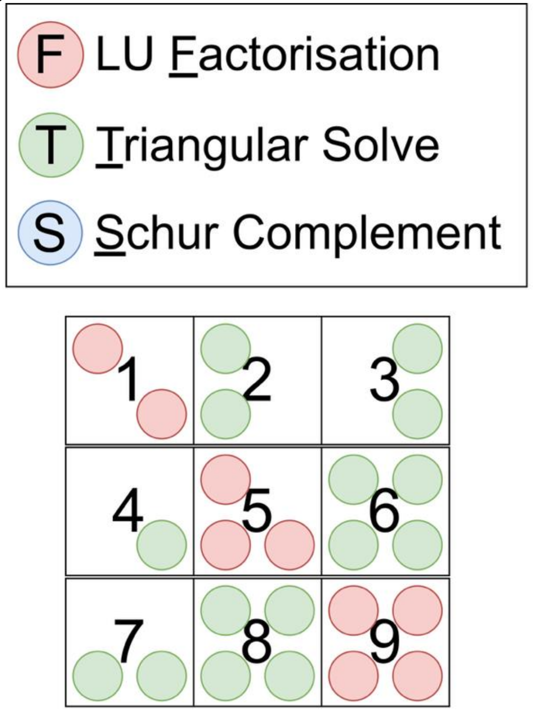
After this, four Schur complement tasks can be executed, depending on previous results.

The dependencies between tasks are complex in sparse LU factorisation, which restricts concurrency.

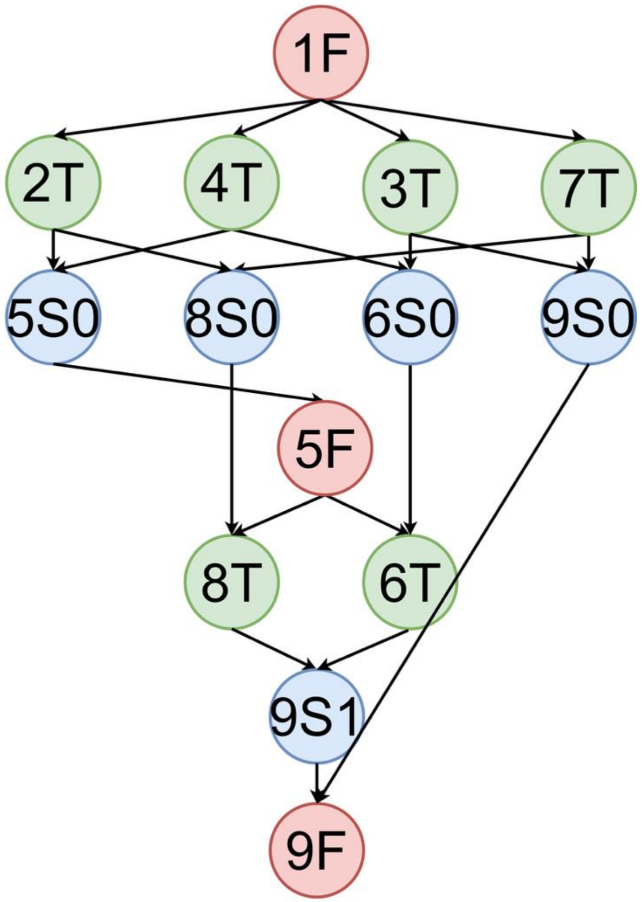
Task Dependencies Restrict Concurrency

Super Scientific Software Laboratory

There are three task types (LU factorisation, triangular solve and Schur complement) in sparse LU factorisation. This figure shows the task dependency of factorising a 6-by-6 blocked matrix.



Task Count Statistics
LU factorisation x3
Triangular solve x6
Schur complement x5



Fristly, task '1F' starts.

Then, four triangular solve tasks can be executed, depending on the result of '1F'.

After this, four Schur complement tasks can be executed, depending on previous results.

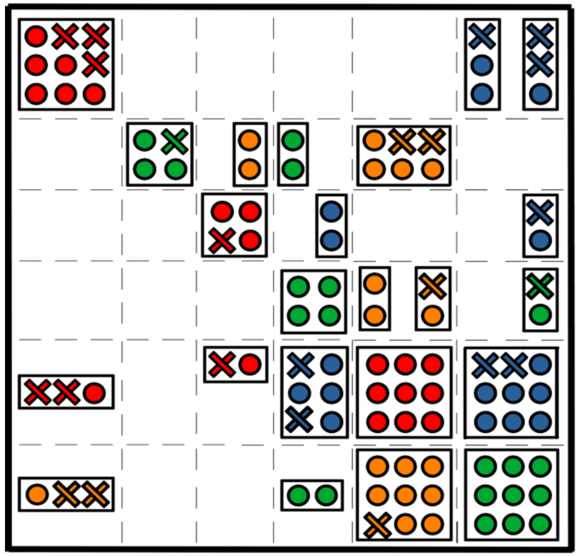
The dependencies between tasks are complex in sparse LU factorisation, which restricts concurrency.

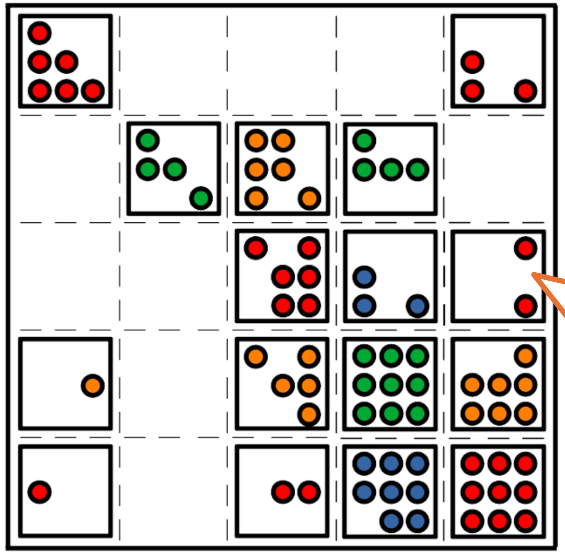
Outline

- Background
  - ① Sparse LU Factorisation
  - ② Task Dependencies Restrict Concurrency
  - ③ Single Task Is Too Small For a GPU
- Motivations
  - ① Aggregate: to Prepare More Tasks for a GPU
  - ② Batch: to Selectively Run the Tasks in Parallel
- Trojan Horse
  - ① Overview
  - ② An Example to Use the Trojan Horse
  - ③ Functional Modules of the Trojan Horse (Priortizer, Container, Collector & Executor)
- Experiments
  - ① Experimental Setup
  - ② Scale-Up Evaluation
  - ③ Scale-Out Evaluation
  - ④ Comparison with CPU solvers
- Conclusion

Single Task Is Too Small For a GPU

Existing methods break the matrix into small blocks and generate small tasks.





The small scale of individual tasks limits the effective utilisation of GPU parallelism.

Supernodal / Multifrontal Methods:
The input of each task is generally very small, typically on the order of 10 on average.

Sparse Blocking Methods:
The input of each task is generally bigger, typically on the order of 512, with a sparsity of approximately 0.05 on average.

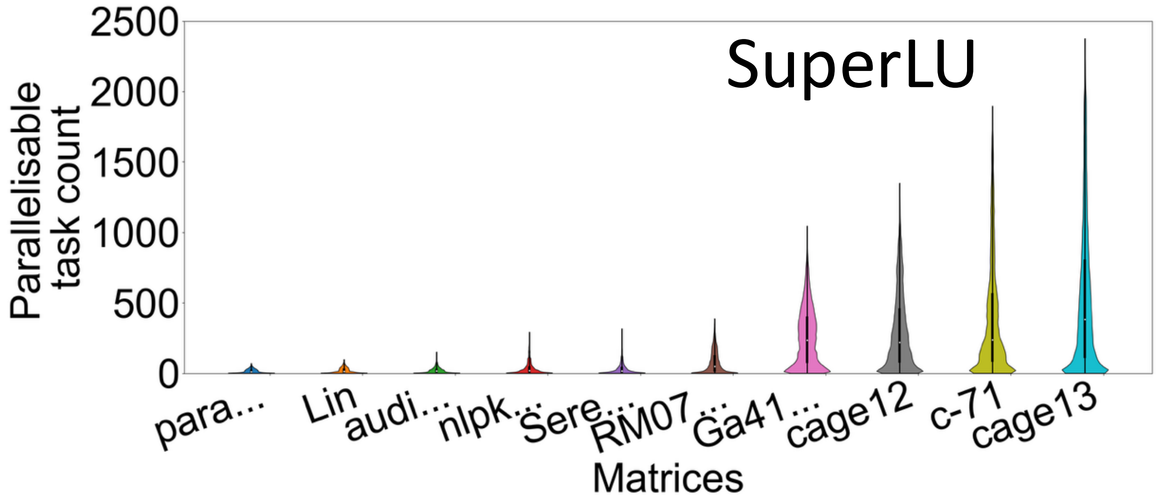
Outline

- Background
  - ① Sparse LU Factorisation
  - ② Task Dependencies Restrict Concurrency
  - ③ Single Task Is Too Small For a GPU
- Motivations
  - ① Aggregate: to Prepare More Tasks for a GPU
  - ② Batch: to Selectively Run the Tasks in Parallel
- Trojan Horse
  - ① Overview
  - ② An Example to Use the Trojan Horse
  - ③ Functional Modules of the Trojan Horse (Priortizer, Container, Collector & Executor)
- Experiments
  - ① Experimental Setup
  - ② Scale-Up Evaluation
  - ③ Scale-Out Evaluation
  - ④ Comparison with CPU solvers
- Conclusion

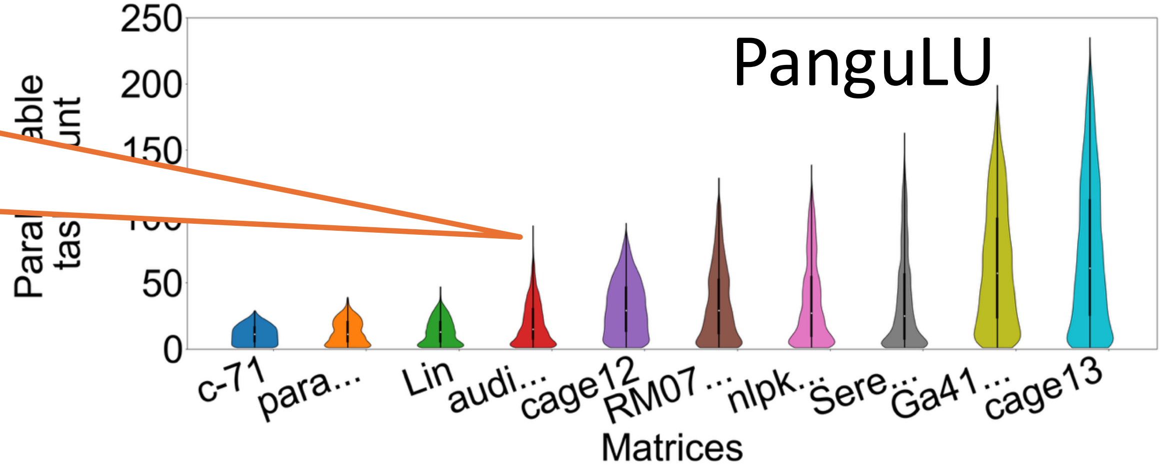
Aggregate: to Prepare More Tasks for a GPU

In the numeric factorisation stage, some tasks are mutually independent and they can be executed concurrently.

We conduct a static analysis on the task DAGs from SuperLU and PanguLU, recording the parallelisable task count.



In the violin plots, the width at each vertical position indicates the count of occurrences for a specific batch size. The height of each violin indicates the maximum parallelisable task count.

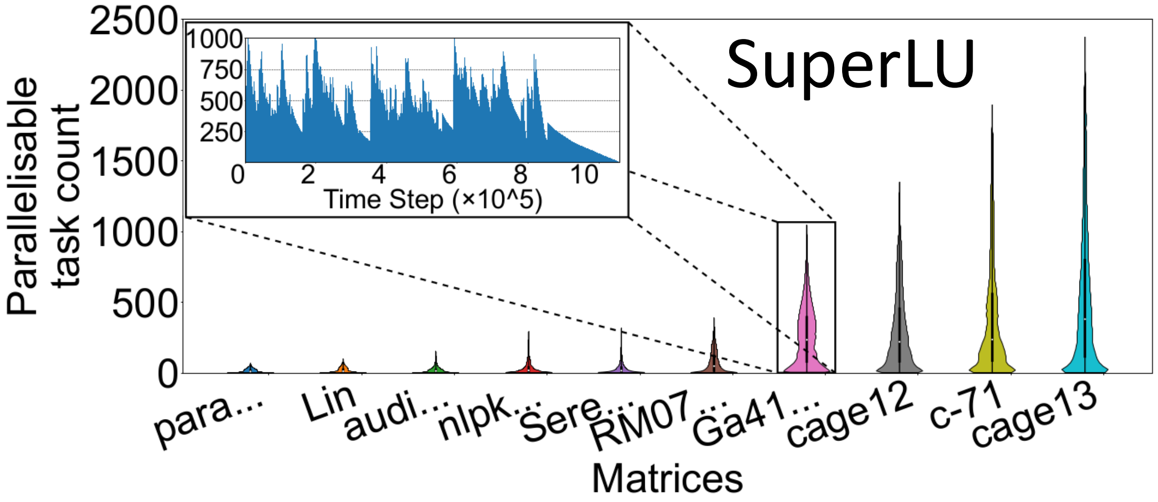


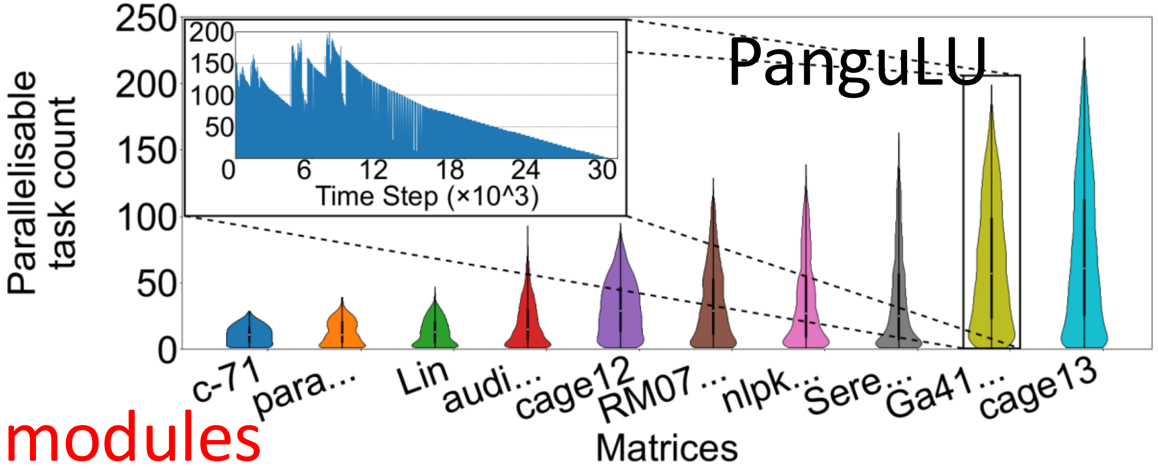
Aggregate: to Prepare More Tasks for a GPU

Taking the matrix 'Ga41As41H72' highlighted, the highest number of tasks can run in parallel are 1047 and 199 in SuperLU and PanguLU. The observation brings the potential to run the tasks in a batch mode.

Considerations

- Prepare adequate tasks
- Obey dependency constraints
- Consider task priority





We will design the Aggregate stage with two modules called Prioritizer and Container in Trojan Horse.

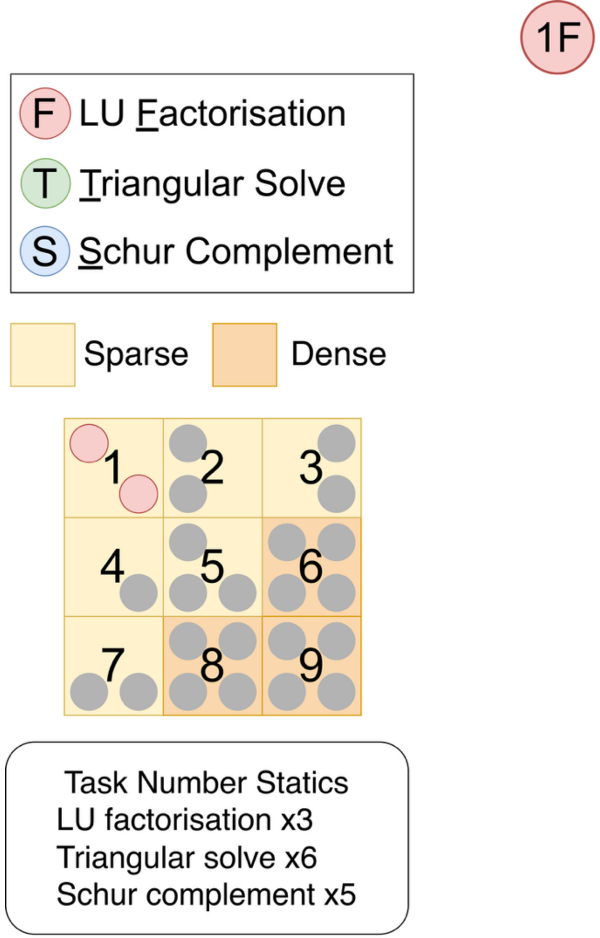
Outline

- Background
  ① Sparse LU Factorisation
  ② Task Dependencies Restrict Concurrency
  ③ Single Task Is Too Small For a GPU
- Motivations
  ① Aggregate: to Prepare More Tasks for a GPU
  ② Batch: to Selectively Run the Tasks in Parallel
- Trojan Horse
  ① Overview
  ② An Example to Use the Trojan Horse
  ③ Functional Modules of the Trojan Horse (Priortizer, Container, Collector & Executor)
- Experiments
  ① Experimental Setup
  ② Scale-Up Evaluation
  ③ Scale-Out Evaluation
  ④ Comparison with CPU solvers
- Conclusion

Batch: to Selectively Run the Tasks in Parallel

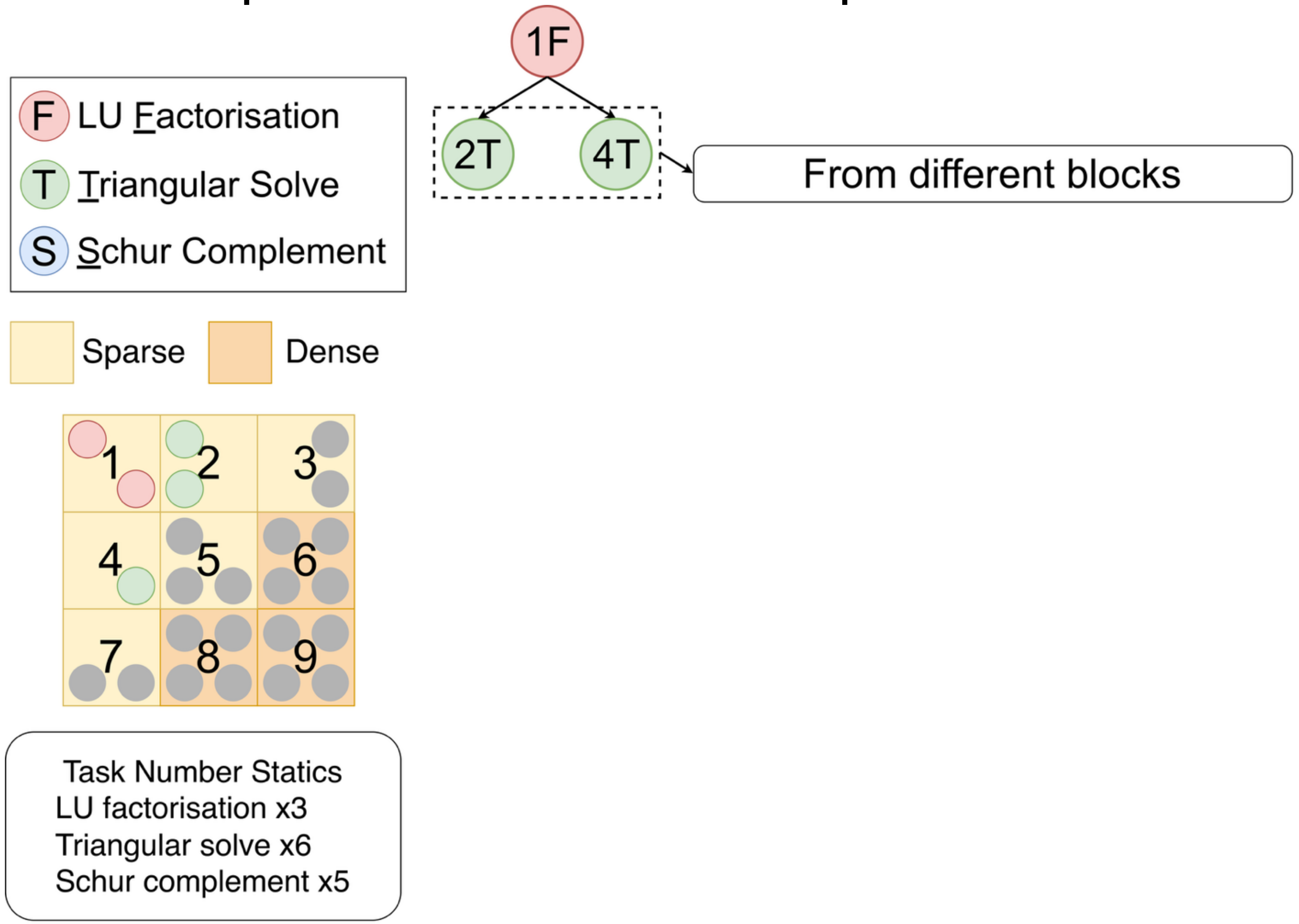
超级科学软件实验室
Super Scientific Software Laboratory

This figure shows the parallelisable tasks when factorising a 6-by-6 sparse matrix, assuming a GPU can process two tasks in parallel. It leads to a requirement to batch diverse tasks.



Batch: to Selectively Run the Tasks in Parallel

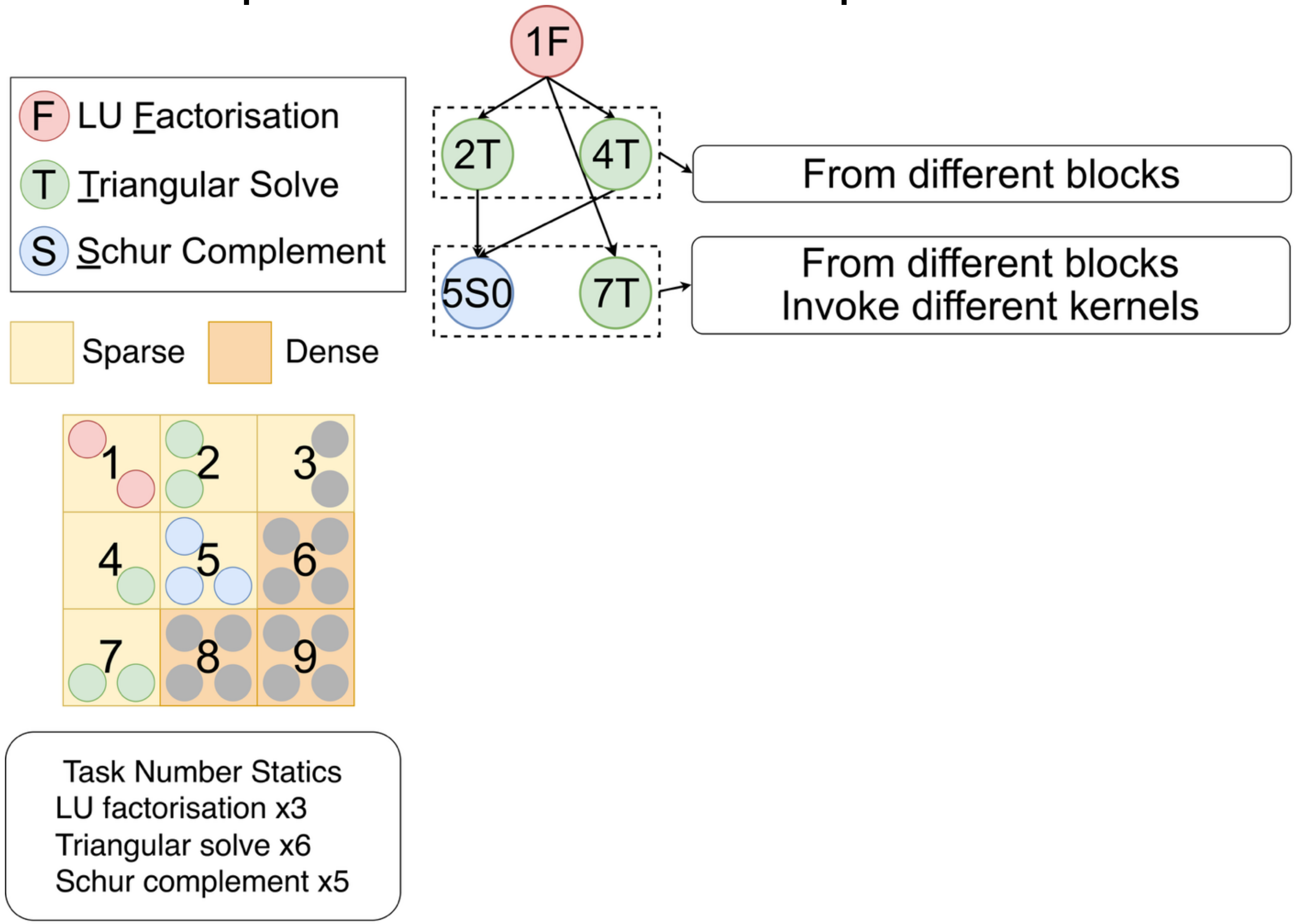
This figure shows the parallelisable tasks when factorising a 6-by-6 sparse matrix, assuming a GPU can process two tasks in parallel. It leads to a requirement to batch diverse tasks.



Batch: to Selectively Run the Tasks in Parallel

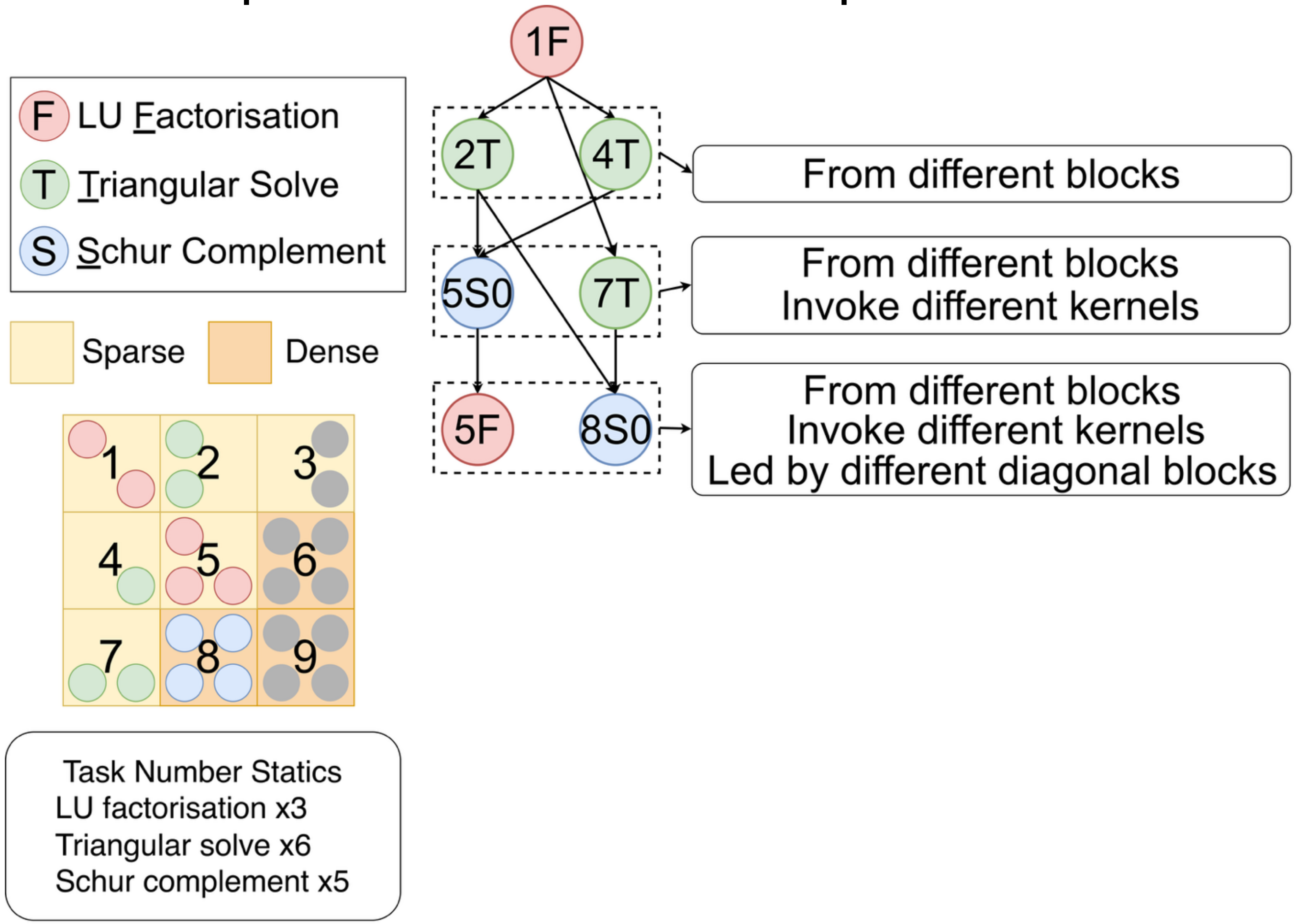
超级科学软件实验室
Super Scientific Software Laboratory

This figure shows the parallelisable tasks when factorising a 6-by-6 sparse matrix, assuming a GPU can process two tasks in parallel. It leads to a requirement to batch diverse tasks.



Batch: to Selectively Run the Tasks in Parallel

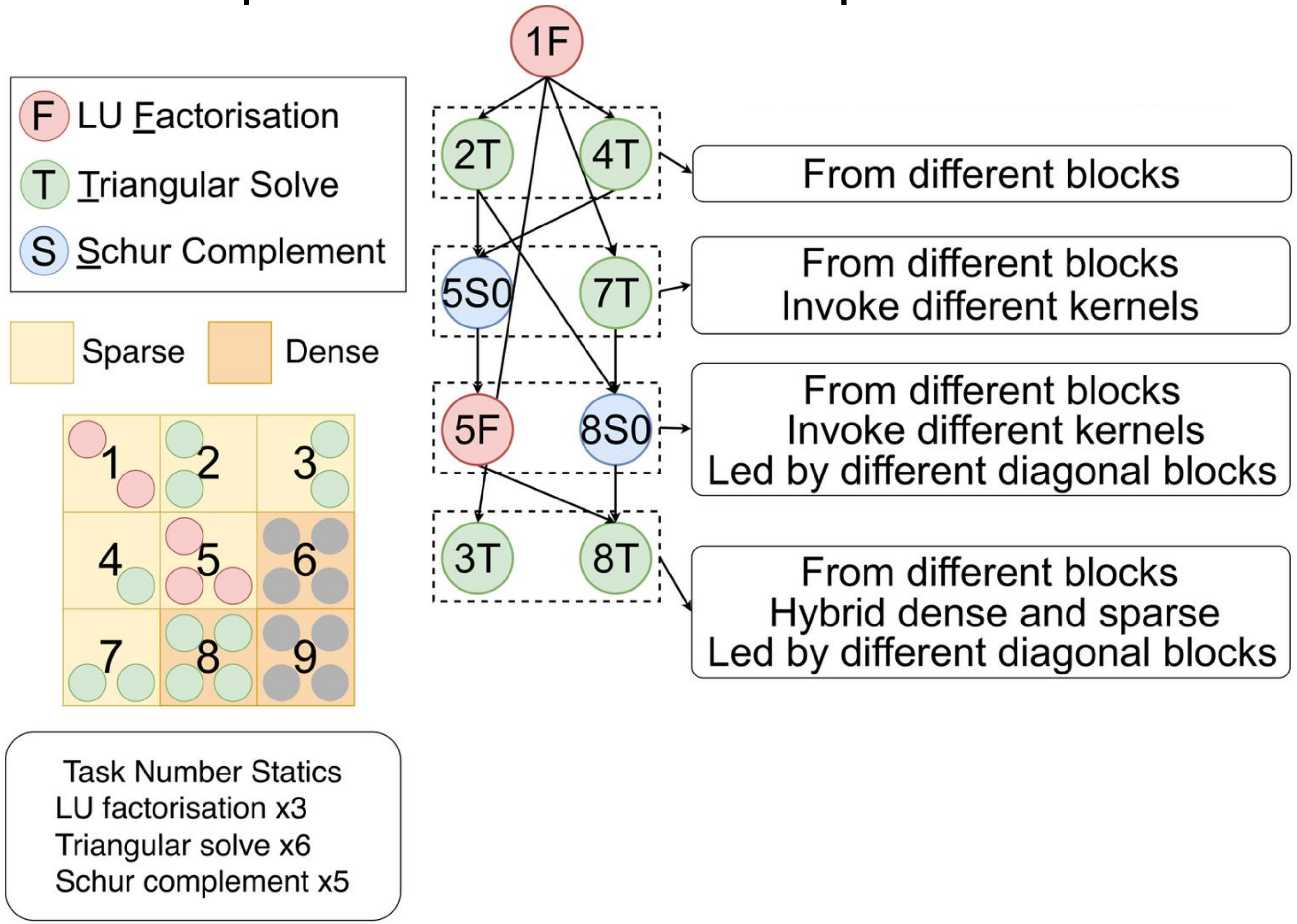
This figure shows the parallelisable tasks when factorising a 6-by-6 sparse matrix, assuming a GPU can process two tasks in parallel. It leads to a requirement to batch diverse tasks.



Batch: to Selectively Run the Tasks in Parallel

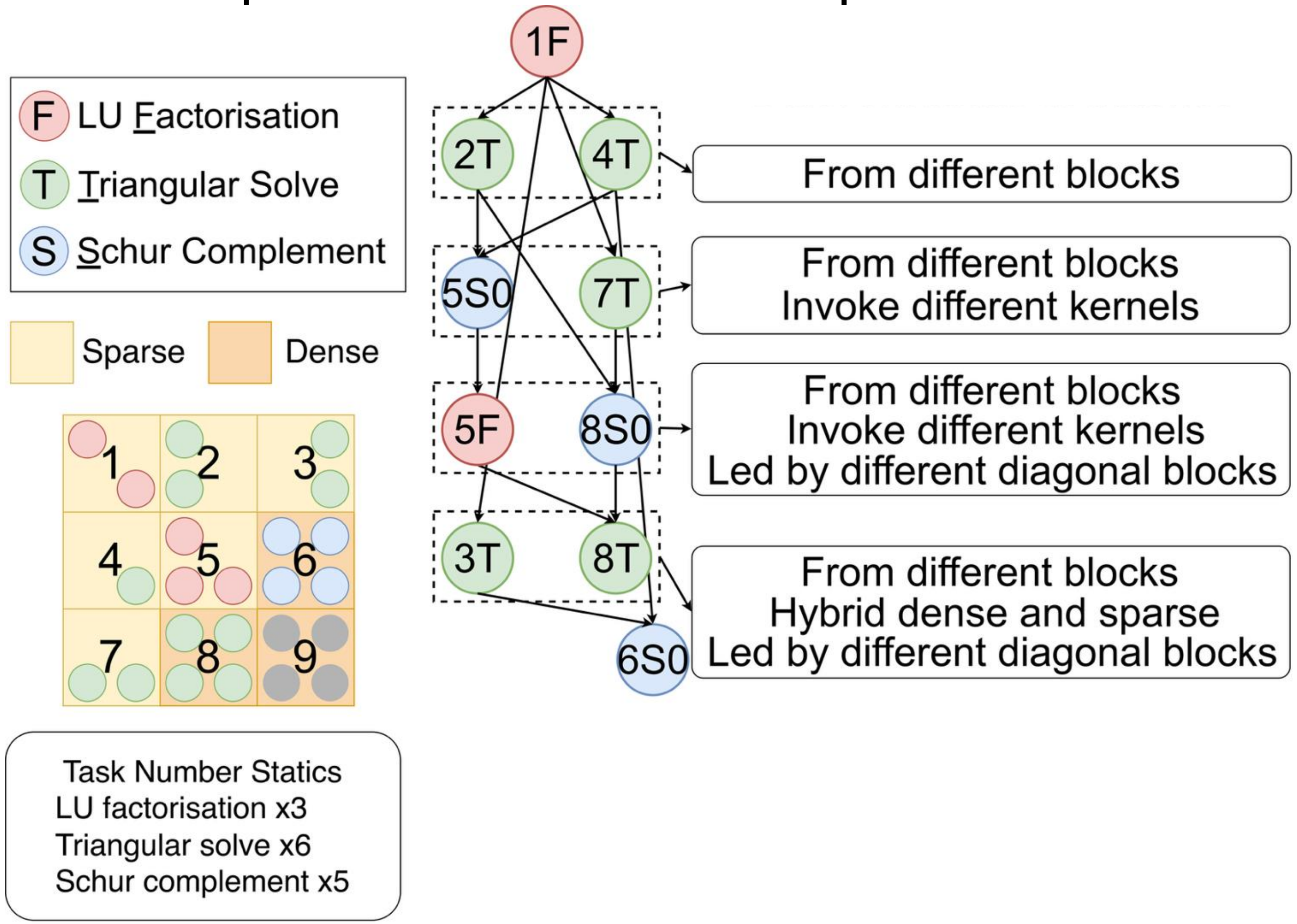
超级科学软件实验室
Super Scientific Software Laboratory

This figure shows the parallelisable tasks when factorising a 6-by-6 sparse matrix, assuming a GPU can process two tasks in parallel. It leads to a requirement to batch diverse tasks.



Batch: to Selectively Run the Tasks in Parallel

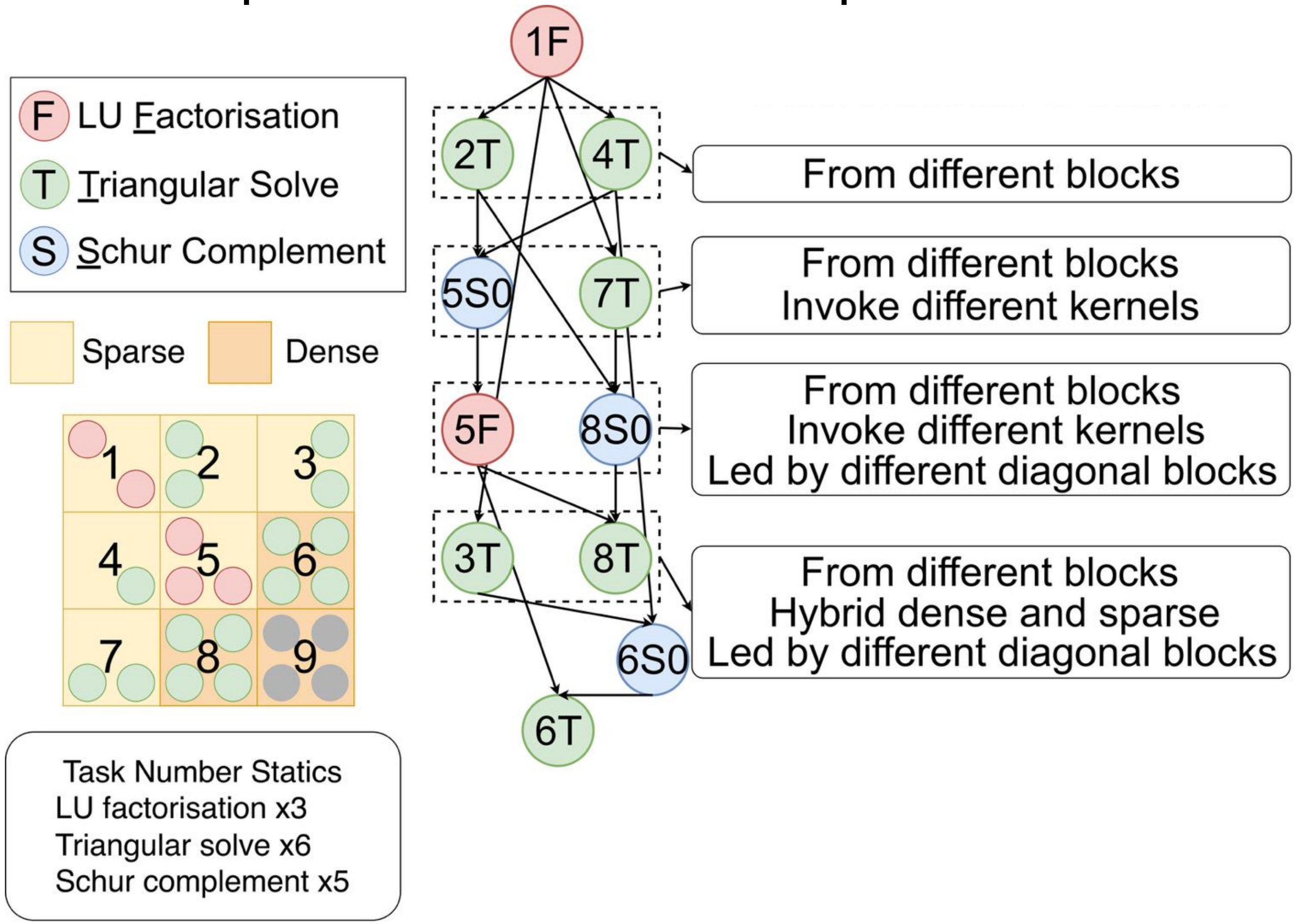
This figure shows the parallelisable tasks when factorising a 6-by-6 sparse matrix, assuming a GPU can process two tasks in parallel. It leads to a requirement to batch diverse tasks.



Batch: to Selectively Run the Tasks in Parallel

超级科学软件实验室
Super Scientific Software Laboratory

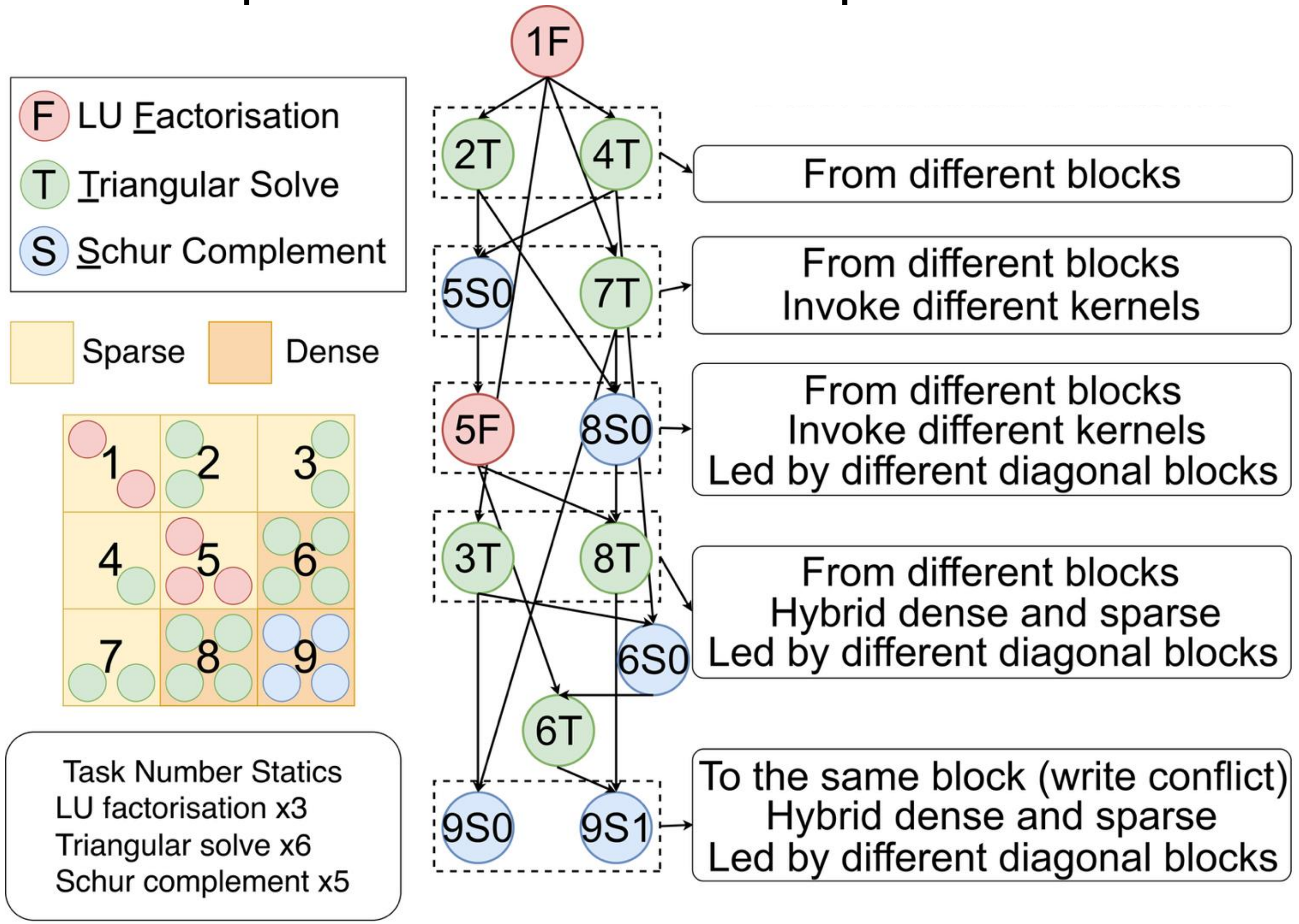
This figure shows the parallelisable tasks when factorising a 6-by-6 sparse matrix, assuming a GPU can process two tasks in parallel. It leads to a requirement to batch diverse tasks.



Batch: to Selectively Run the Tasks in Parallel

超级科学软件实验室
Super Scientific Software Laboratory

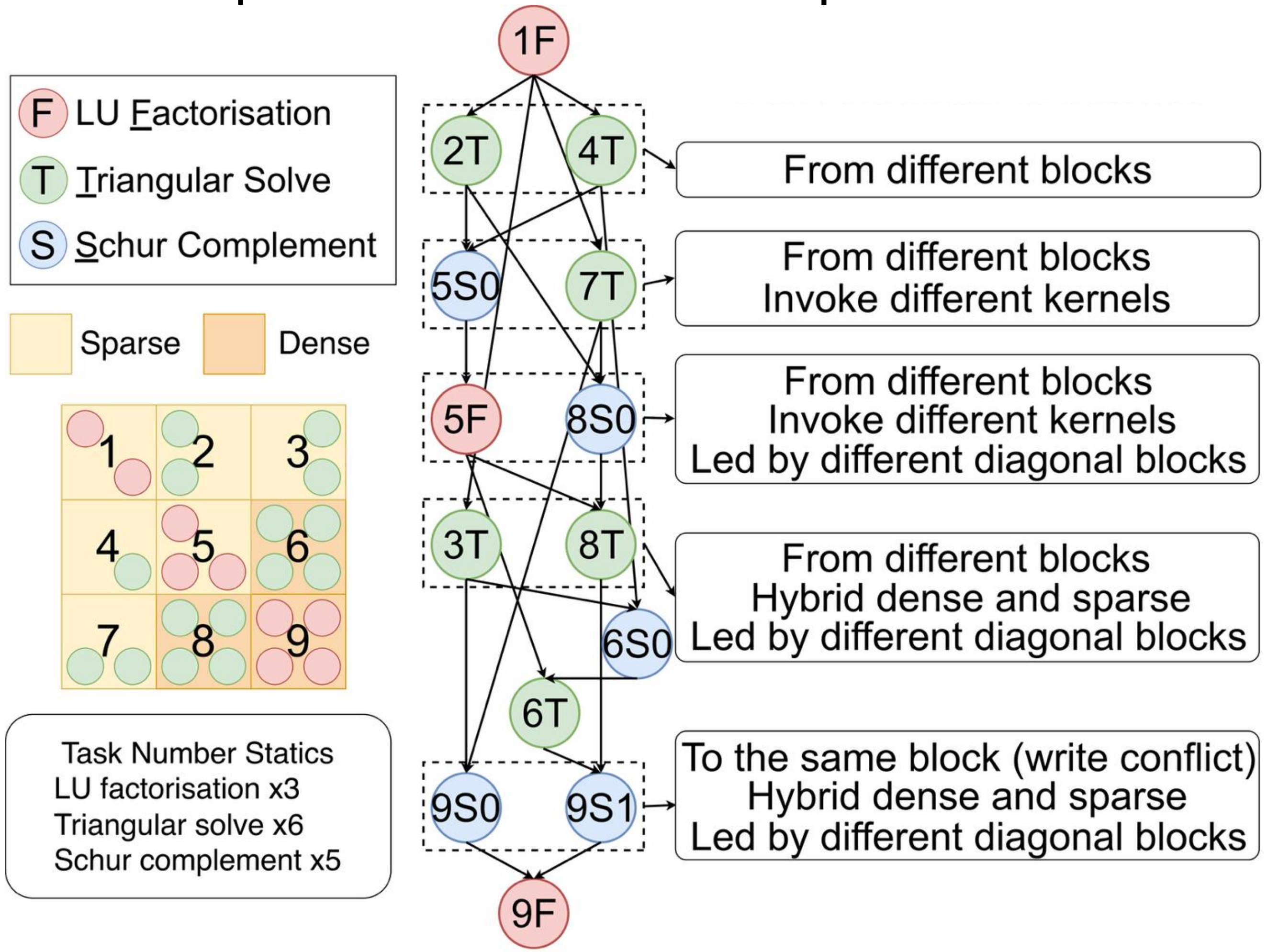
This figure shows the parallelisable tasks when factorising a 6-by-6 sparse matrix, assuming a GPU can process two tasks in parallel. It leads to a requirement to batch diverse tasks.



Batch: to Selectively Run the Tasks in Parallel

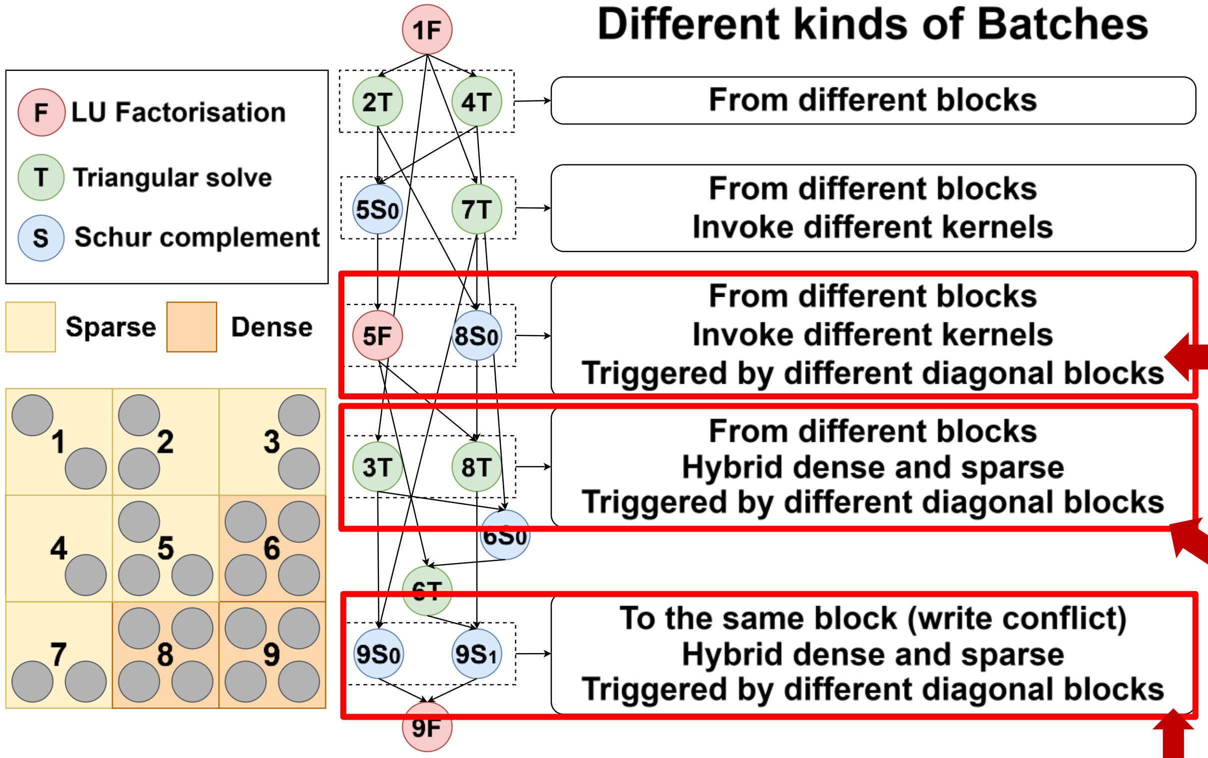
超级科学软件实验室
Super Scientific Software Laboratory

This figure shows the parallelisable tasks when factorising a 6-by-6 sparse matrix, assuming a GPU can process two tasks in parallel. It leads to a requirement to batch diverse tasks.



Batch: to Selectively Run the Tasks in Parallel

超级科学软件实验室
Super Scientific Software Laboratory



The Batch stage would receive

- tasks on different blocks
- tasks of different types,
- tasks triggered by differe

Trojan Horse: to **aggregate** and **batch** small tasks for saturating GPUs.

For different tasks in one batched execution,

- their kernel may be different,

Tasks '5F' (LU factorisation on block 5) and '8S_0' (Schur update on block 8) are triggered by different diagonal blocks (blocks 1 and 5) and can be batched, despite involving different kernels.

- their inputs may be dense or sparse blocks, and

Tasks '3T' (triangular solve on block 3, sparse) and '8T' (triangular solve on block 8, dense) can be batched, despite one is sparse and the other is dense.

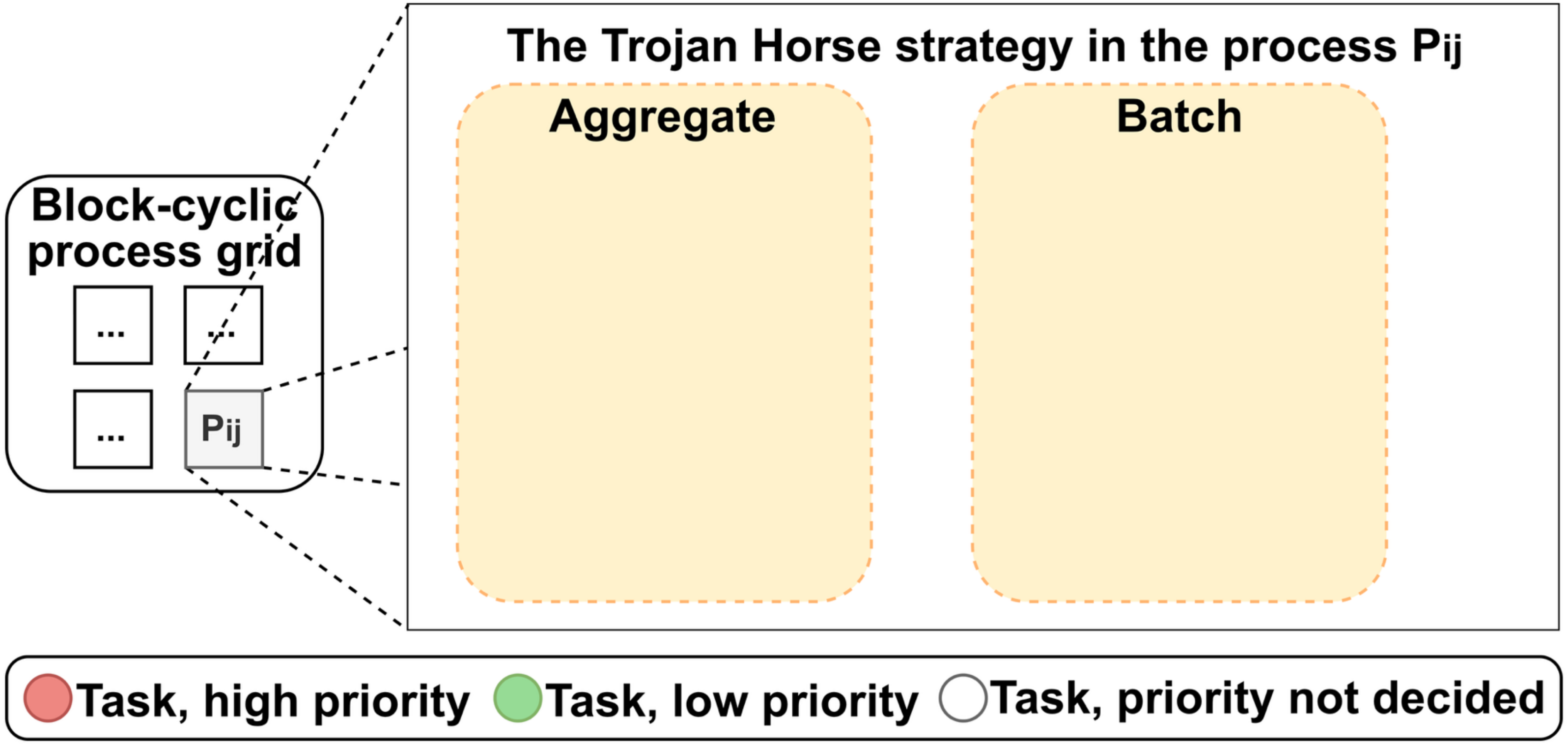
- they may write the same block.

Tasks '9S_0' (Schur update on block 9, triggered by the 0th diagonal block 1) and '9S_1' (Schur update on block 8 , triggered by the 1st diagonal block 5) can be batched. Both task compute Schur update on block 9. Batching them will bring write conflict, therefore needs atomic operations.

Outline

Overview

The Trojan Horse focuses on
(1) Scaling up the execution efficiency of a single GPU in a cluster, and
(2) Scaling out to multiple GPUs, and integrating to distributed solvers.



Two stages:

- Aggregate
- Batch

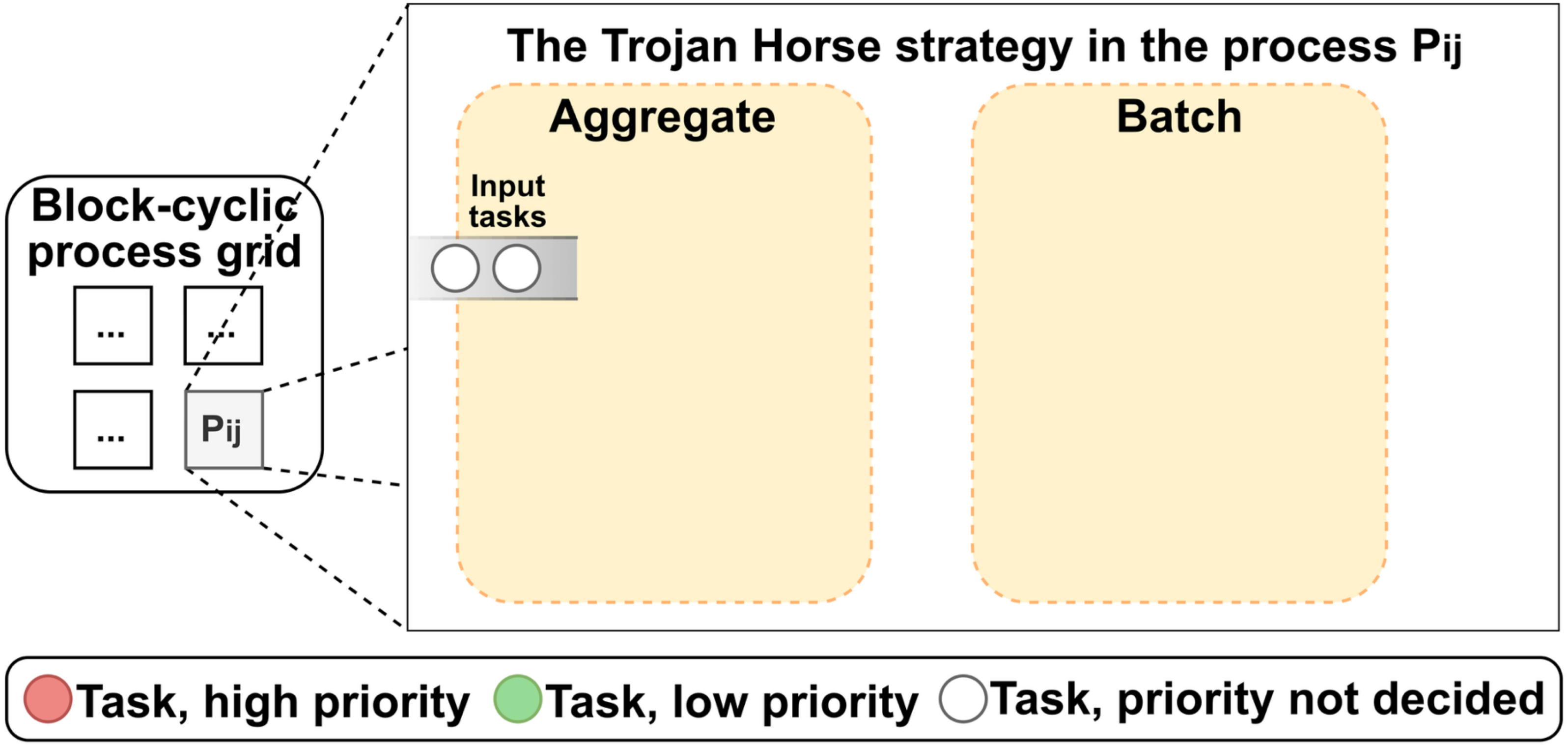
Four functional modules:

- Prioritizer
- Container
- Collector
- Executor

Overview

超级科学软件实验室
Super Scientific Software Laboratory

The Trojan Horse focuses on
(1) Scaling up the execution efficiency of a single GPU in a cluster, and
(2) Scaling out to multiple GPUs, and integrating to distributed solvers.



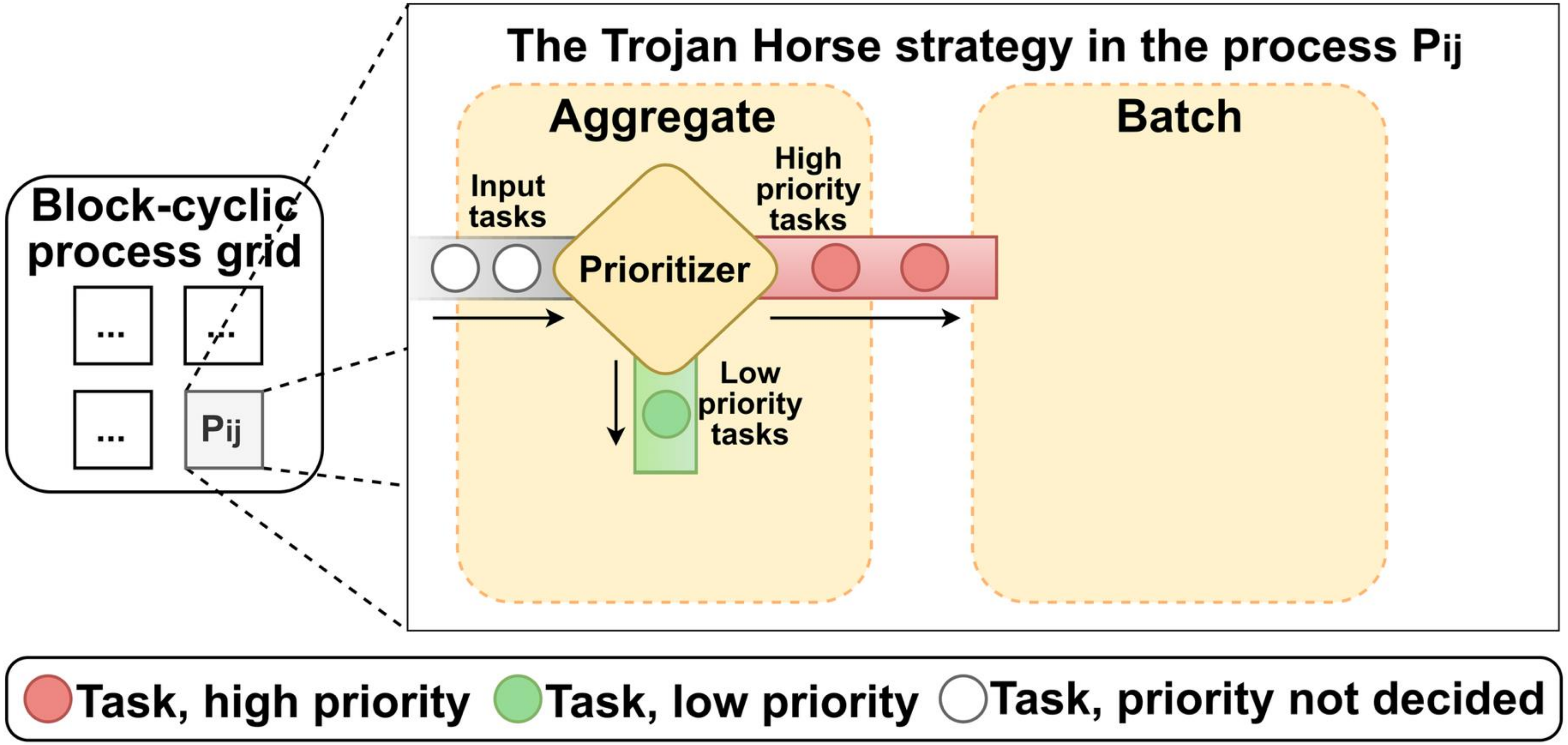
Two stages:
- Aggregate
- Batch

Four functional modules:
- Prioritizer
- Container
- Collector
- Executor

Overview

超级科学软件实验室
Super Scientific Software Laboratory

The Trojan Horse focuses on
(1) Scaling up the execution efficiency of a single GPU in a cluster, and
(2) Scaling out to multiple GPUs, and integrating to distributed solvers.



Two stages:

- Aggregate
- Batch

Four functional modules:

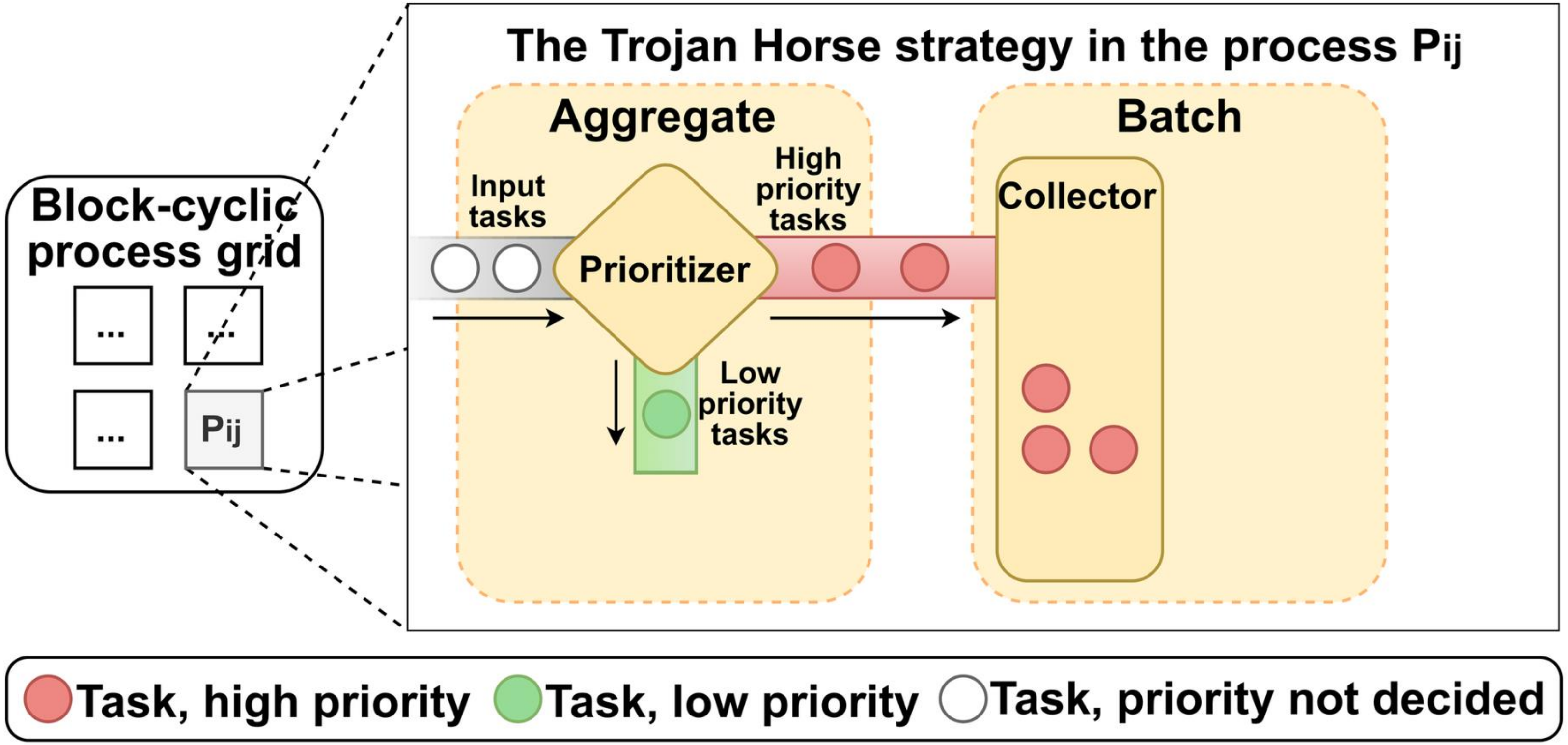
- Prioritizer
- Container
- Collector
- Executor

Overview

The Trojan Horse focuses on

(1) Scaling up the execution efficiency of a single GPU in a cluster, and

(2) Scaling out to multiple GPUs, and integrating to distributed solvers.



Two stages:

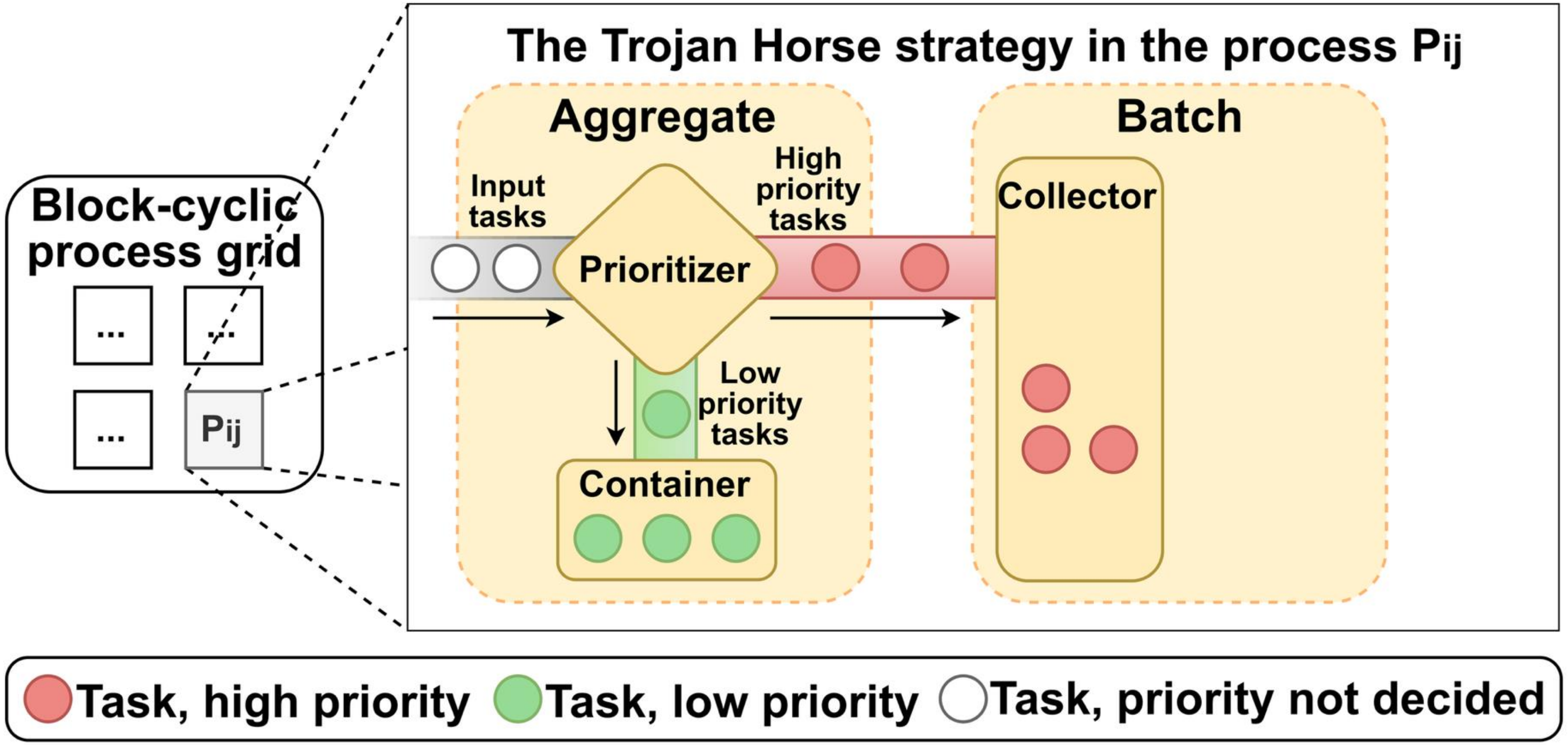
- Aggregate
- Batch

Four functional modules:

- Prioritizer
- Container
- Collector
- Executor

Overview

The Trojan Horse focuses on
(1) Scaling up the execution efficiency of a single GPU in a cluster, and
(2) Scaling out to multiple GPUs, and integrating to distributed solvers.



Two stages:

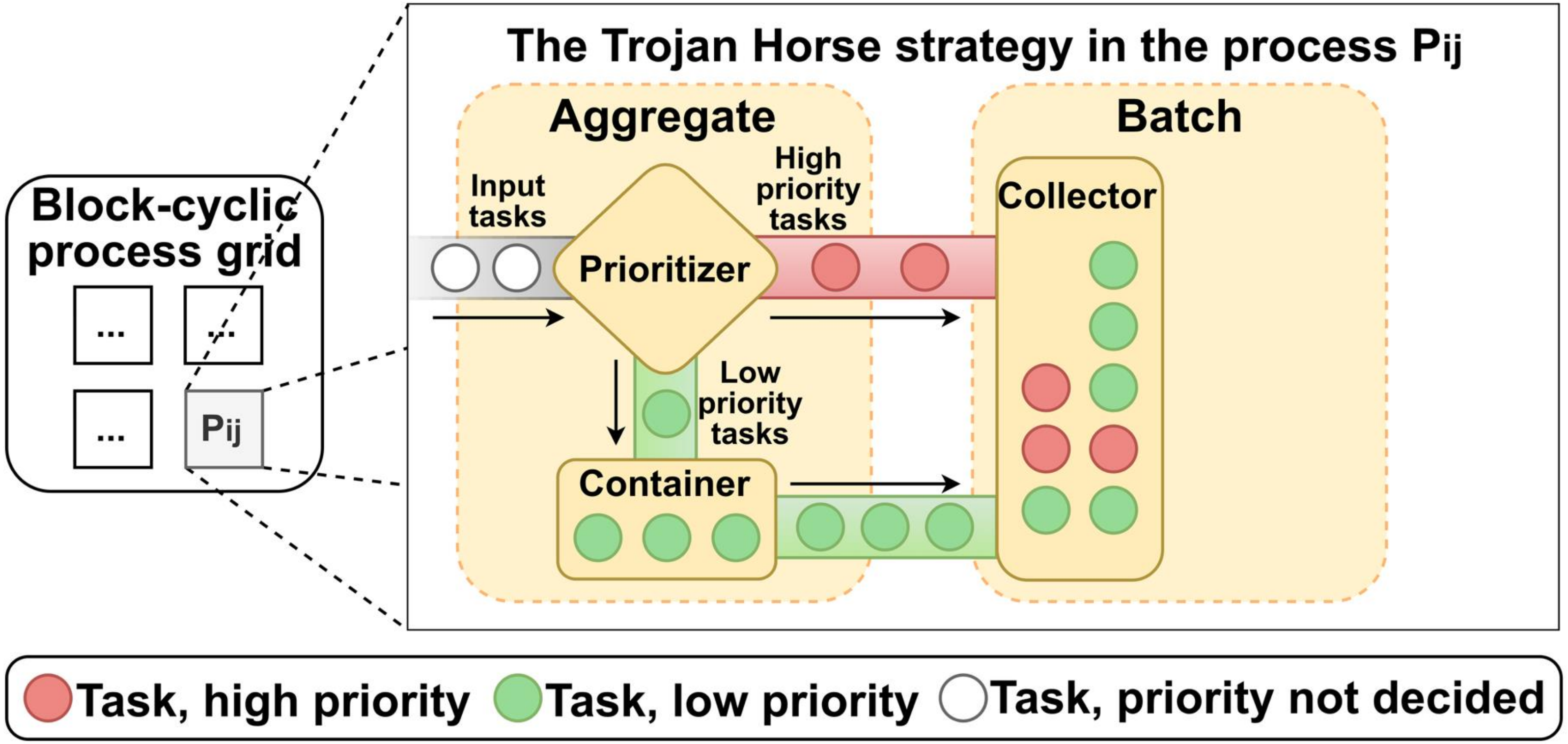
- Aggregate
- Batch

Four functional modules:

- Prioritizer
- Container
- Collector
- Executor

Overview

The Trojan Horse focuses on
(1) Scaling up the execution efficiency of a single GPU in a cluster, and
(2) Scaling out to multiple GPUs, and integrating to distributed solvers.



Two stages:

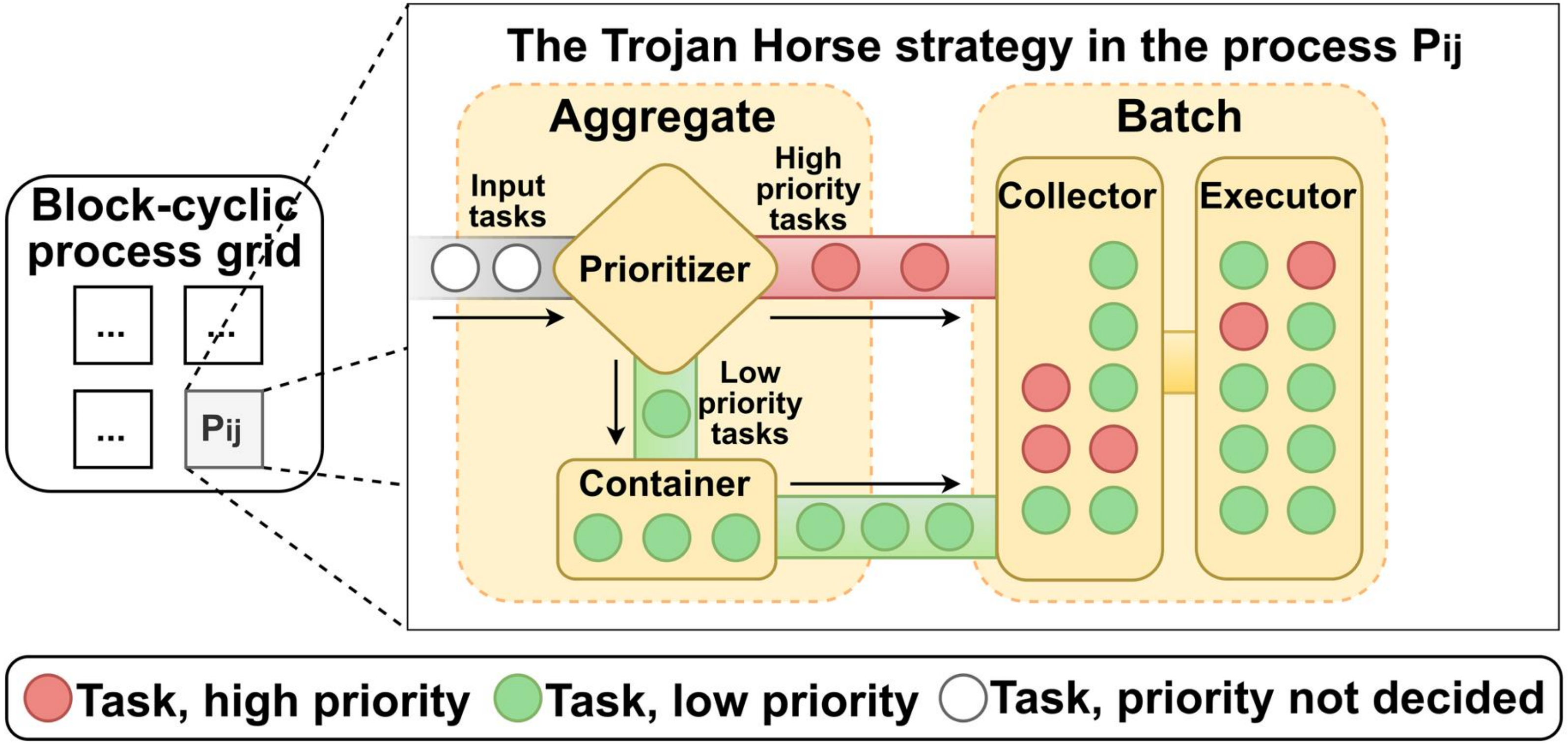
- Aggregate
- Batch

Four functional modules:

- Prioritizer
- Container
- Collector
- Executor

Overview

The Trojan Horse focuses on
(1) Scaling up the execution efficiency of a single GPU in a cluster, and
(2) Scaling out to multiple GPUs, and integrating to distributed solvers.



Two stages:

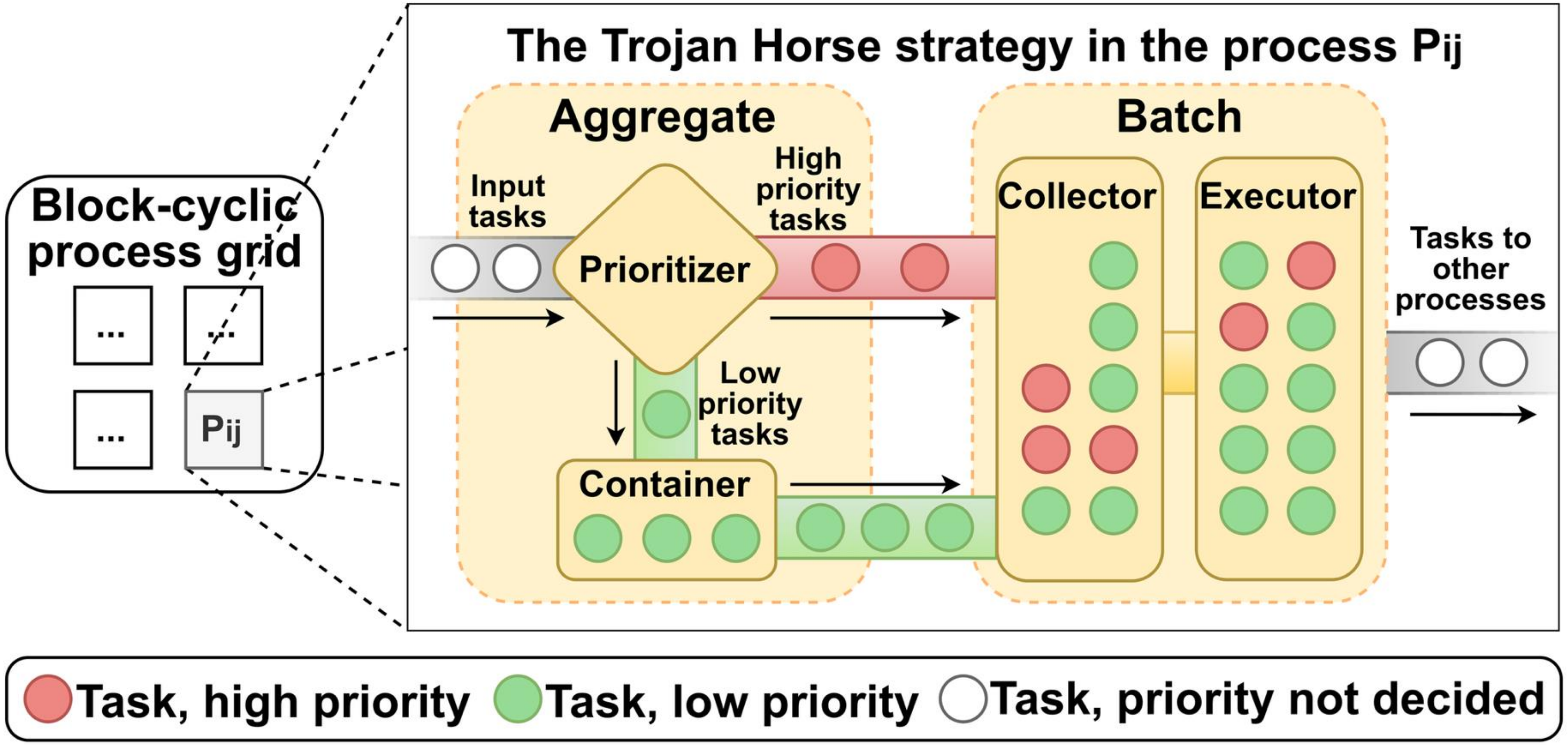
- Aggregate
- Batch

Four functional modules:

- Prioritizer
- Container
- Collector
- Executor

Overview

The Trojan Horse focuses on
(1) Scaling up the execution efficiency of a single GPU in a cluster, and
(2) Scaling out to multiple GPUs, and integrating to distributed solvers.



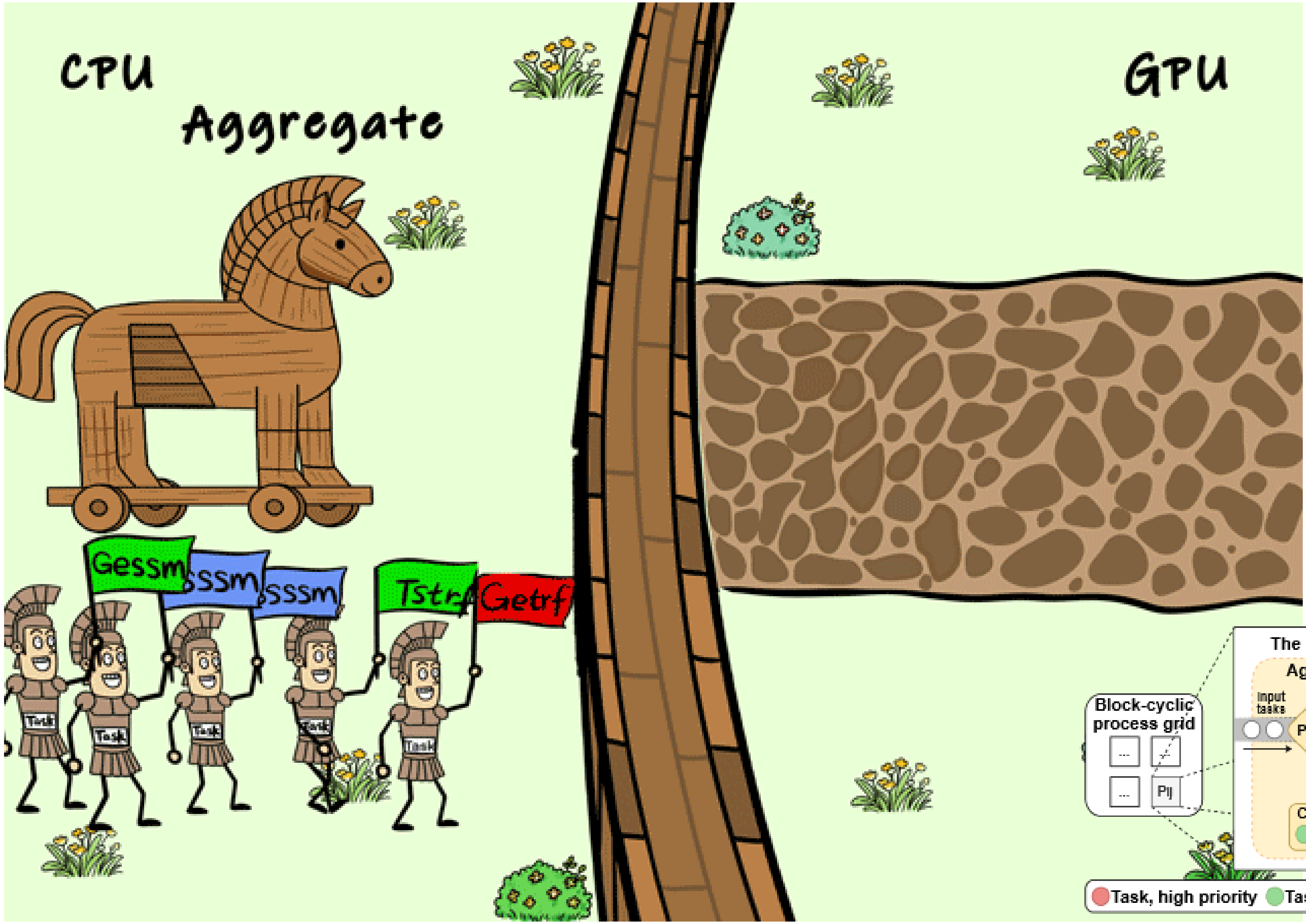
Two stages:

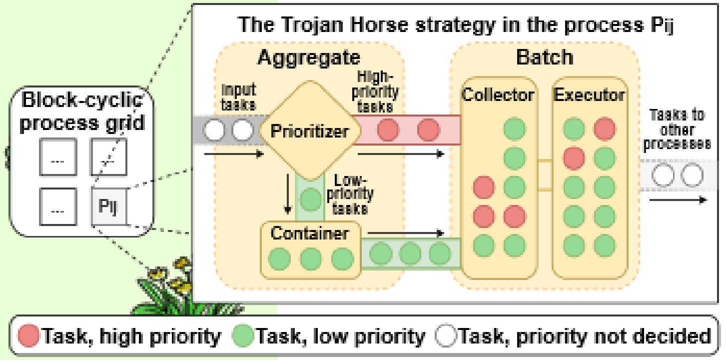
- Aggregate
- Batch

Four functional modules:

- Prioritizer
- Container
- Collector
- Executor

Overview


CPU
Aggregate
GPU
Gessm
Sssm
Sssm
Tstrf
Getrf


The Trojan Horse strategy in the process Pij
Aggregate
Batch
Block-cyclic process grid
Input tasks
High-priority tasks
Prioritizer
Low-priority tasks
Container
Collector
Executor
Tasks to other processes
Pij
Task, high priority
Task, low priority
Task, priority not decided

Outline

- Background
  ① Sparse LU Factorisation
  ② Task Dependencies Restrict Concurrency
  ③ Single Task Is Too Small For a GPU
- Motivations
  ① Aggregate: to Prepare More Tasks for a GPU
  ② Batch: to Selectively Run the Tasks in Parallel
- Trojan Horse
  ① Overview
  ② An Example to Use the Trojan Horse
  ③ Functional Modules of the Trojan Horse (Priortizer, Container, Collector & Executor)
- Experiments
  ① Experimental Setup
  ② Scale-Up Evaluation
  ③ Scale-Out Evaluation
  ④ Comparison with CPU solvers
- Conclusion

An Example to Use the Trojan Horse

We prepare an example of factorising a 6-order blocked matrix using a solver integrated with the Trojan Horse.

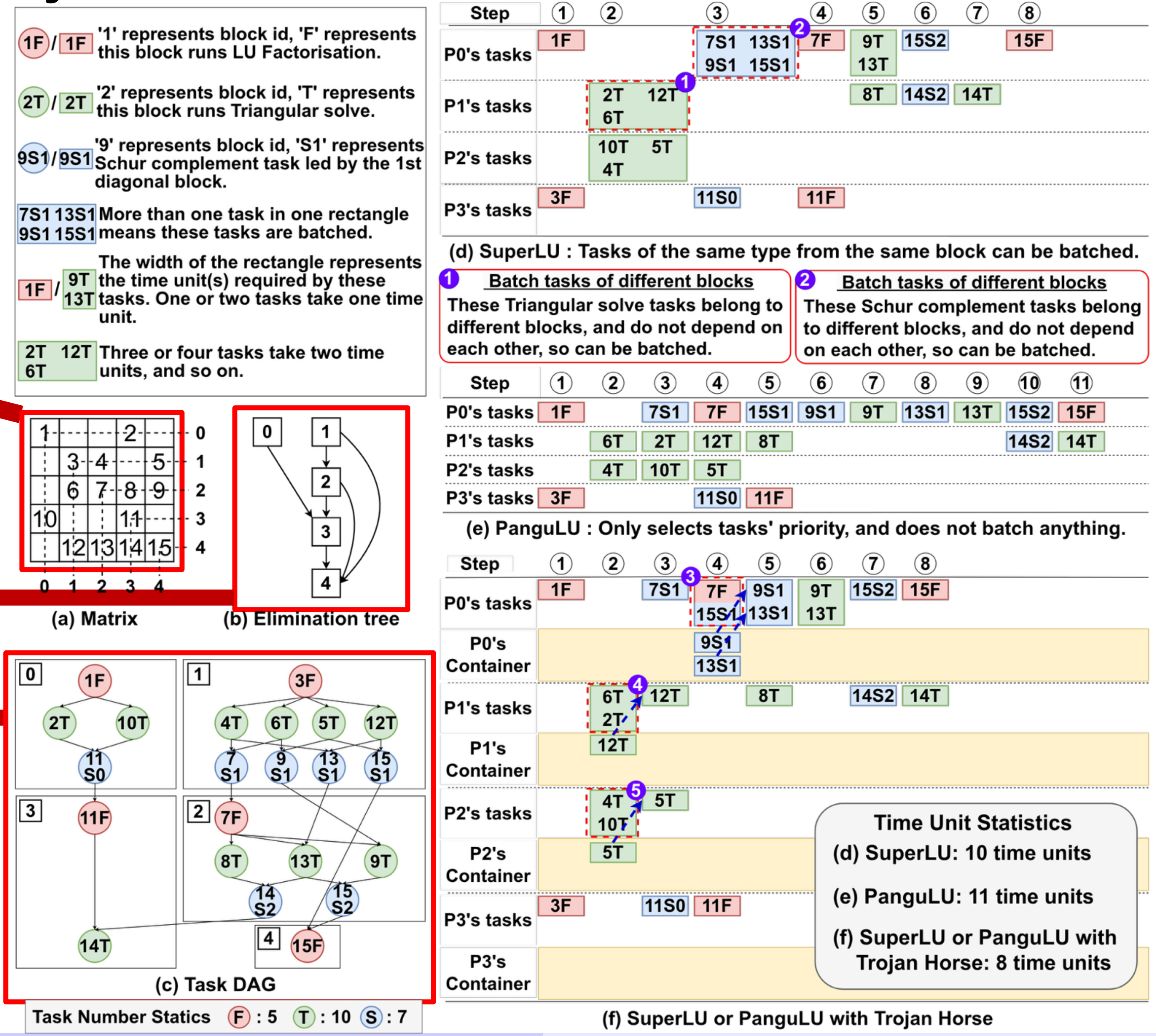
Each number on the blocked matrix labels a nonzero block.

An elimination tree, or a DAG, of the numeric factorisation phase.

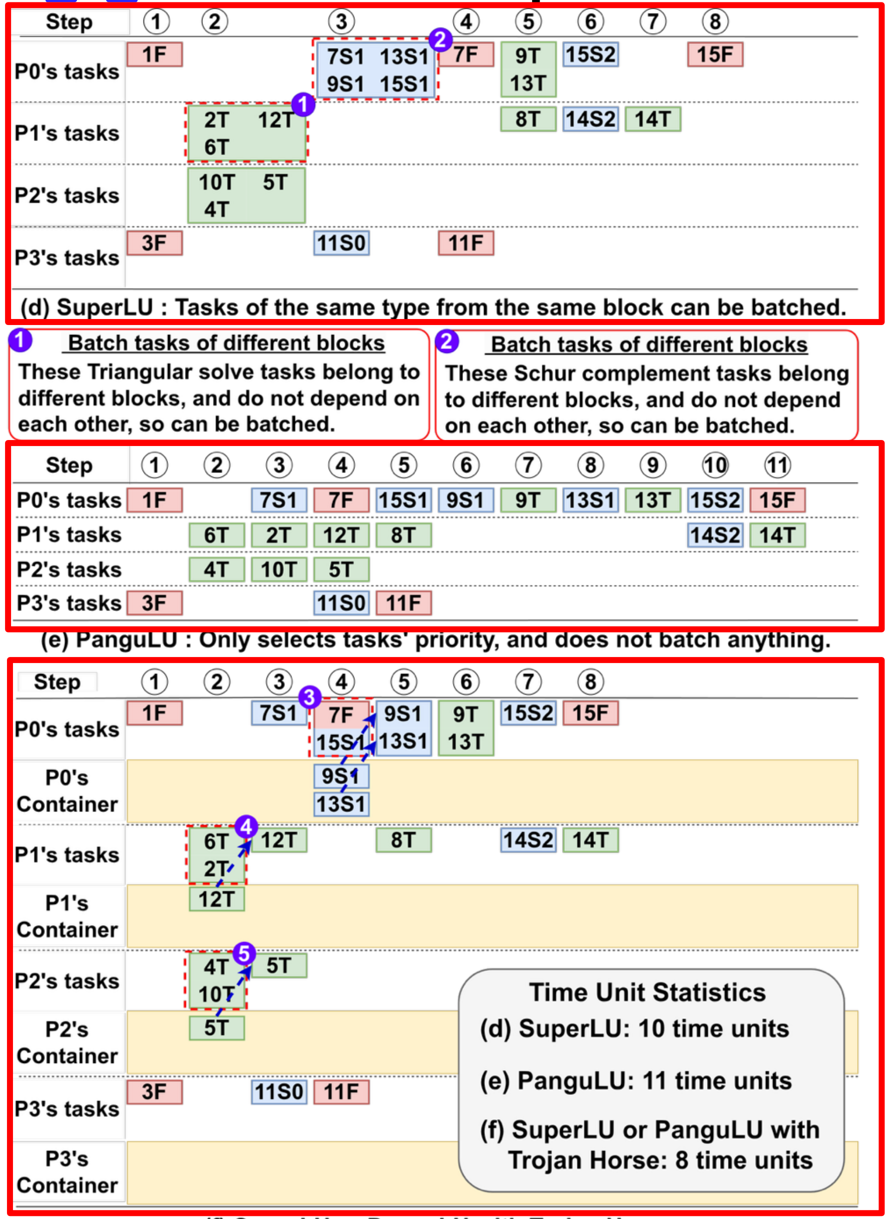
The complete dependencies of all 48 tasks:

- 5 diagonal LU factorisation
- 10 triangular solve
- 7 Schur complement

These tasks are divided into 5 parts by the diagonal blocks 0-4 triggering them.



An Example to Use the Trojan Horse



(f) SuperLU or PanguLU with Trojan Horse

The timeline (10 time units) of SuperLU using four processes.
SuperLU can batch:

- Tasks of the same type
- Tasks from the same elimination tree level.

The timeline (11 time units) of PanguLU using four processes.
PanguLU executes tasks based on priority and without batching.

The timeline (only 8 time units) of SuperLU or PanguLU with the Trojan Horse using four processes.
Solver with Trojan Horse can batch:

- Tasks of different blocks
- Tasks of different types
- Tasks triggered by different diagonal blocks

Assume the GPU can execute two tasks simultaneously.

Outline

- Background
  - ① Sparse LU Factorisation
  - ② Task Dependencies Restrict Concurrency
  - ③ Single Task Is Too Small For a GPU
- Motivations
  - ① Aggregate: to Prepare More Tasks for a GPU
  - ② Batch: to Selectively Run the Tasks in Parallel
- Trojan Horse
  - ① Overview
  - ② An Example to Use the Trojan Horse
  - ③ Functional Modules of the Trojan Horse (Priortizer, Container, Collector & Executor)
- Experiments
  - ① Experimental Setup
  - ② Scale-Up Evaluation
  - ③ Scale-Out Evaluation
  - ④ Comparison with CPU solvers
- Conclusion

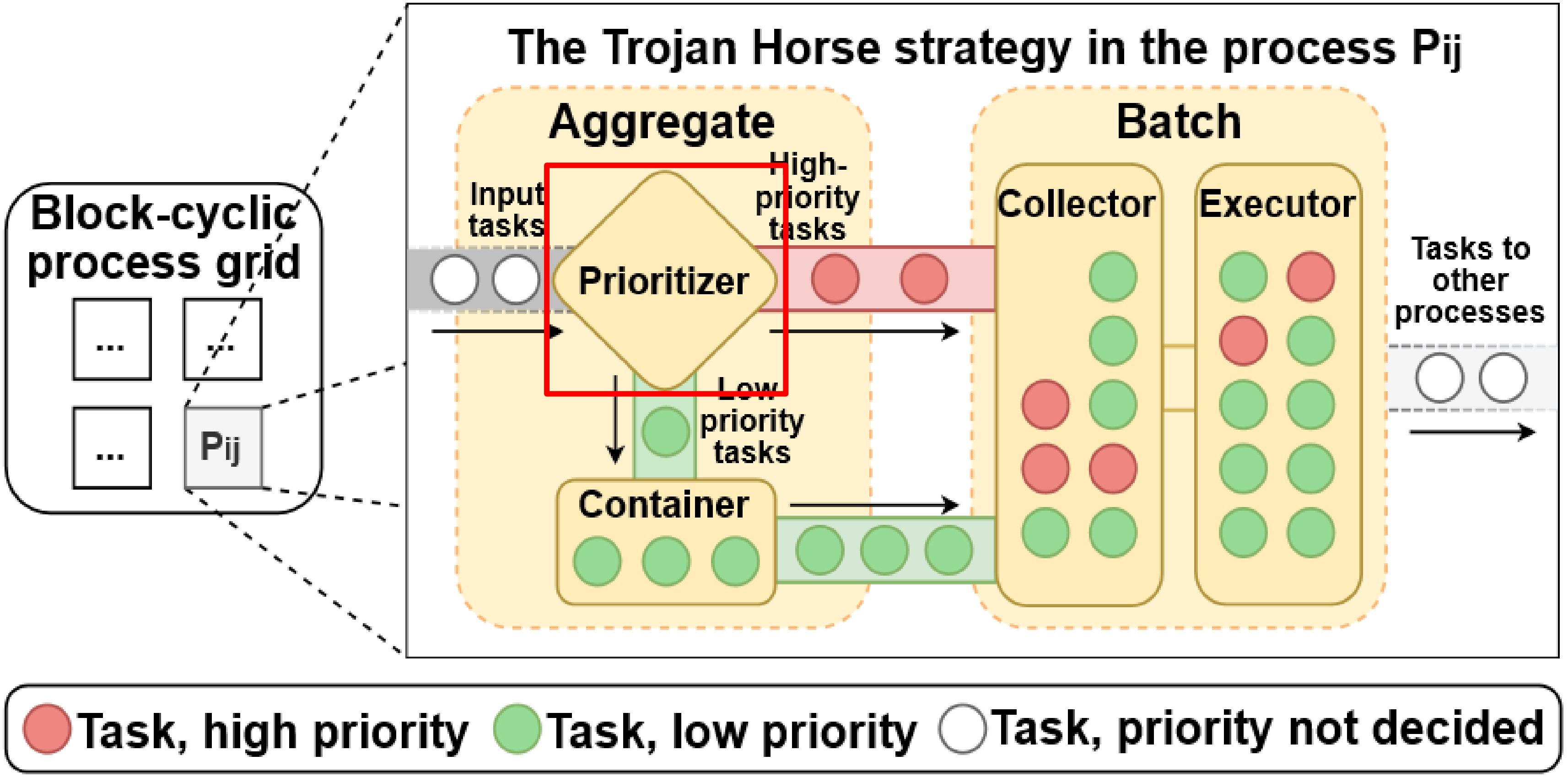
超级科学软件实验室
Super Scientific Software Laboratory

Aggregate Stage: Module 1: Prioritizer

The Prioritizer is designed to

Tag executable tasks and separate them into high- and low-priority tasks.

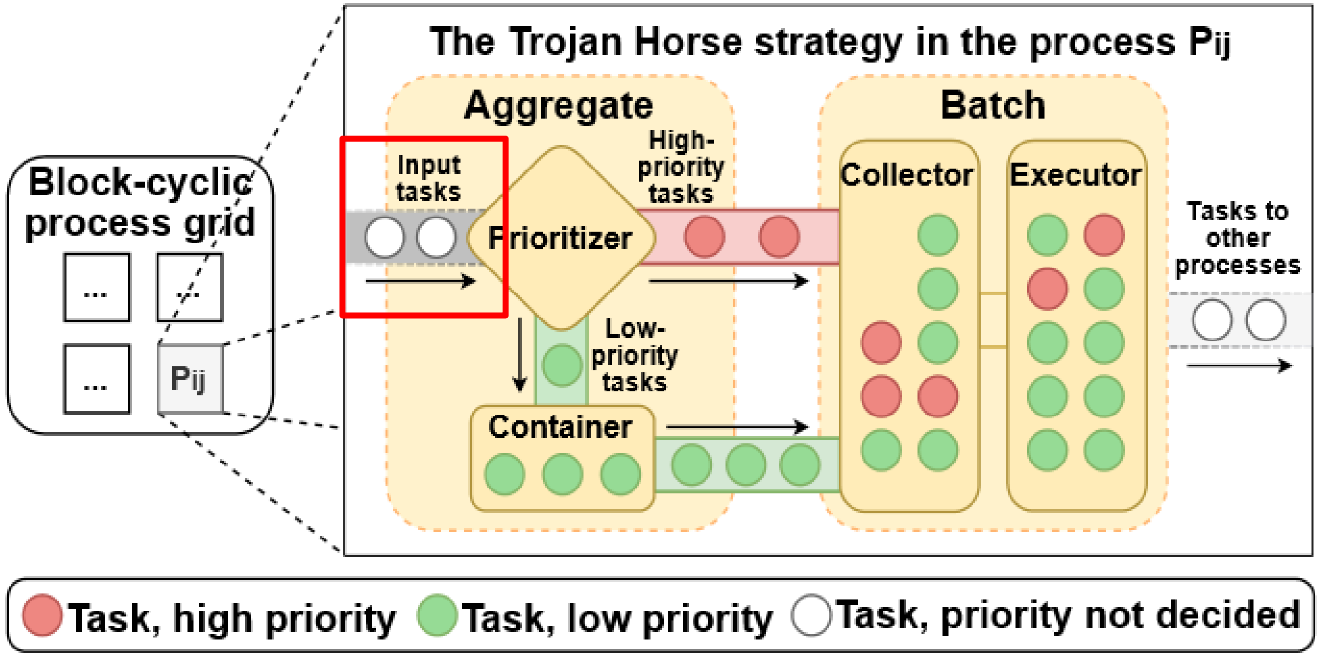
→ Ensure the high-priority tasks be executed earlier.

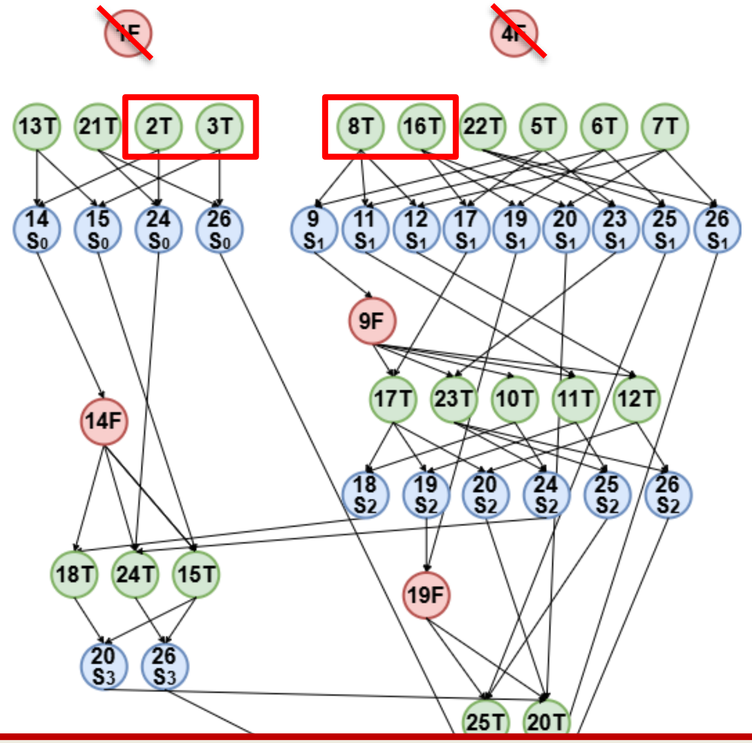


Aggregate Stage: Module 1: Prioritizer

① Receive executable tasks → ② Prioritize executable tasks → ③Provide executable tasks

① The Prioritizer needs to receive executable tasks.





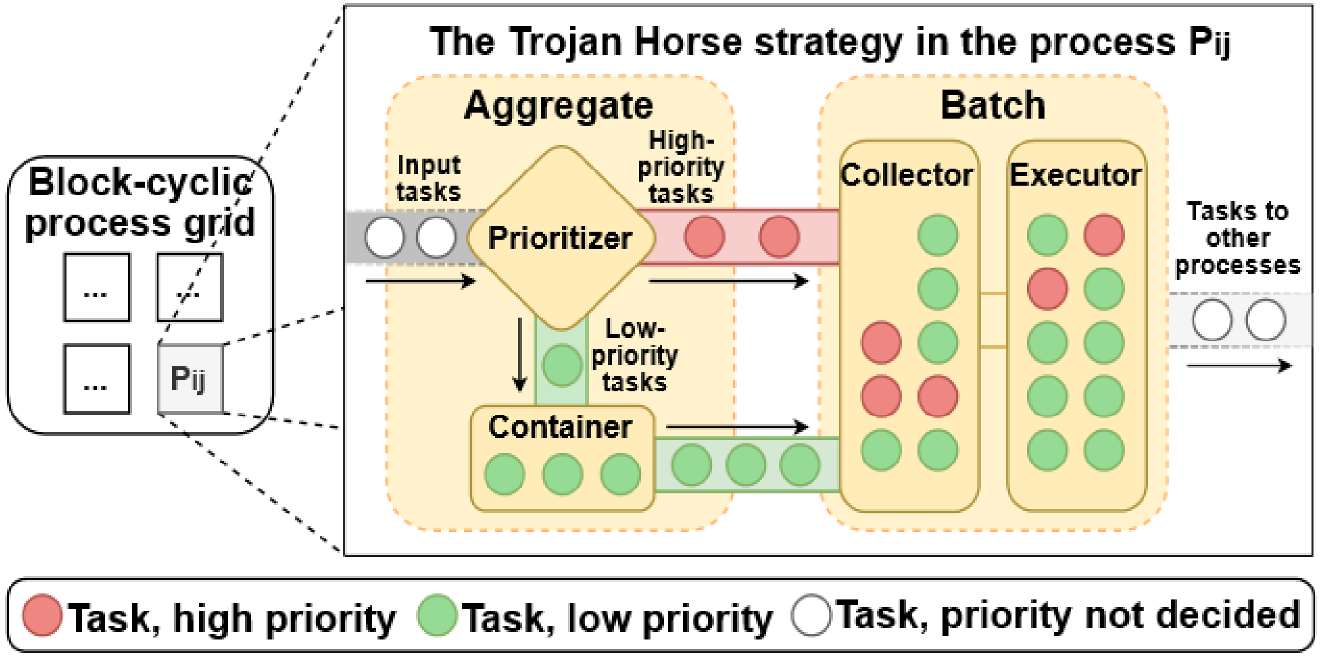
Example:

After the execution of ‘1F’ and ‘4F’, '2T', '3T', '8T', and '16T' are the executable tasks.

Aggregate Stage: Module 1: Prioritizer

① Receive executable tasks → **② Prioritize executable tasks** → **③Provide executable tasks**

② The Prioritizer determines the urgency of each task.



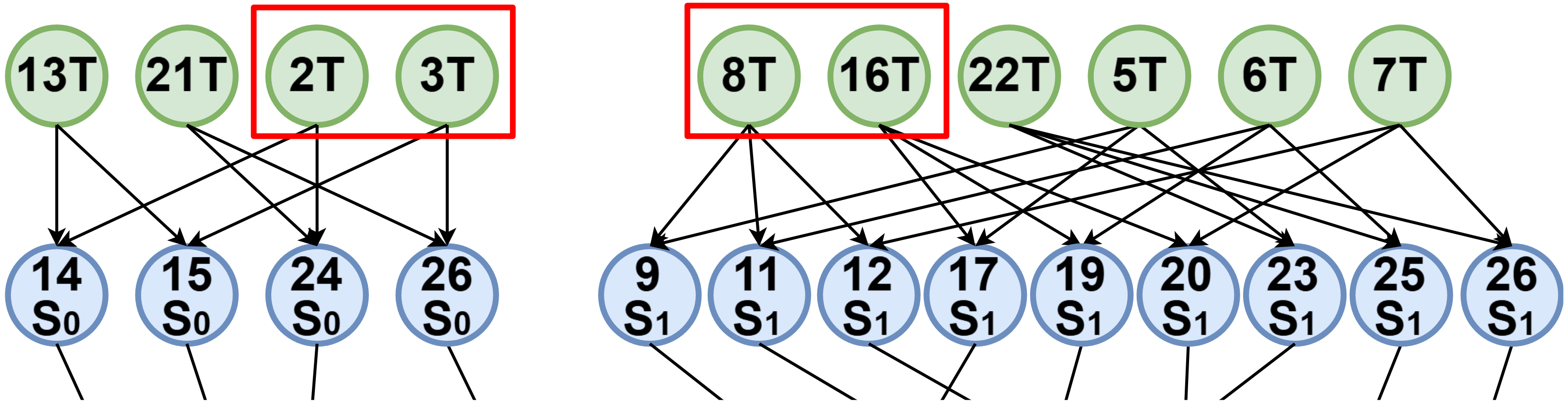
1			2		3
	4	5		6	7
	8	9	10	11	12
13			14		15
	16	17	18	19	20
21	22	23	24	25	26

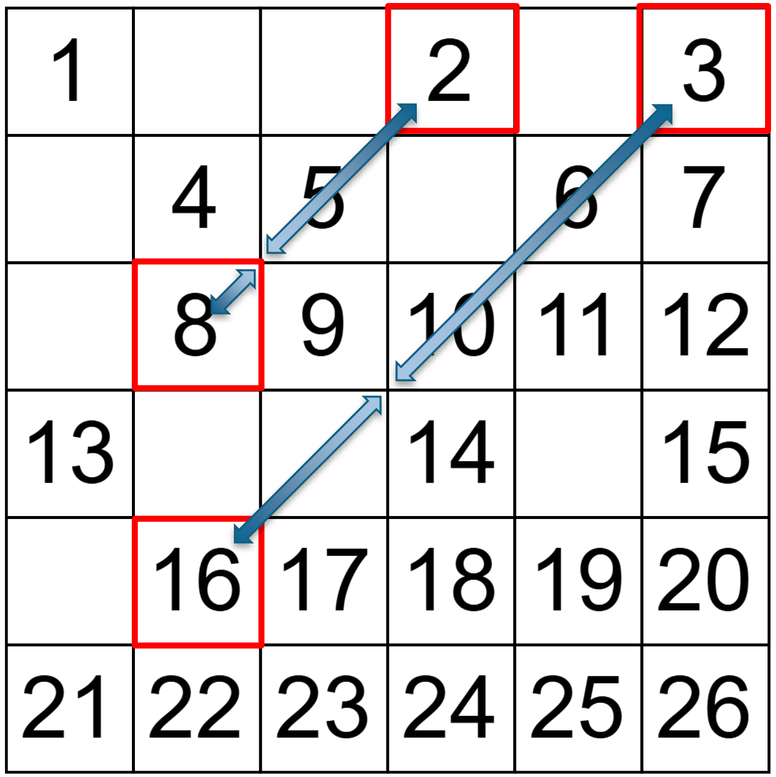
The **closer** a block is to the diagonal block, the **higher** its urgency.

Aggregate Stage: Module 1: Prioritizer

① Receive executable tasks → ② Prioritize executable tasks → ③Provide executable tasks

For example,
'8T' is the closest to the diagonal block (the most urgent), and
'3T' is the farthest from the diagonal block (the least urgent).



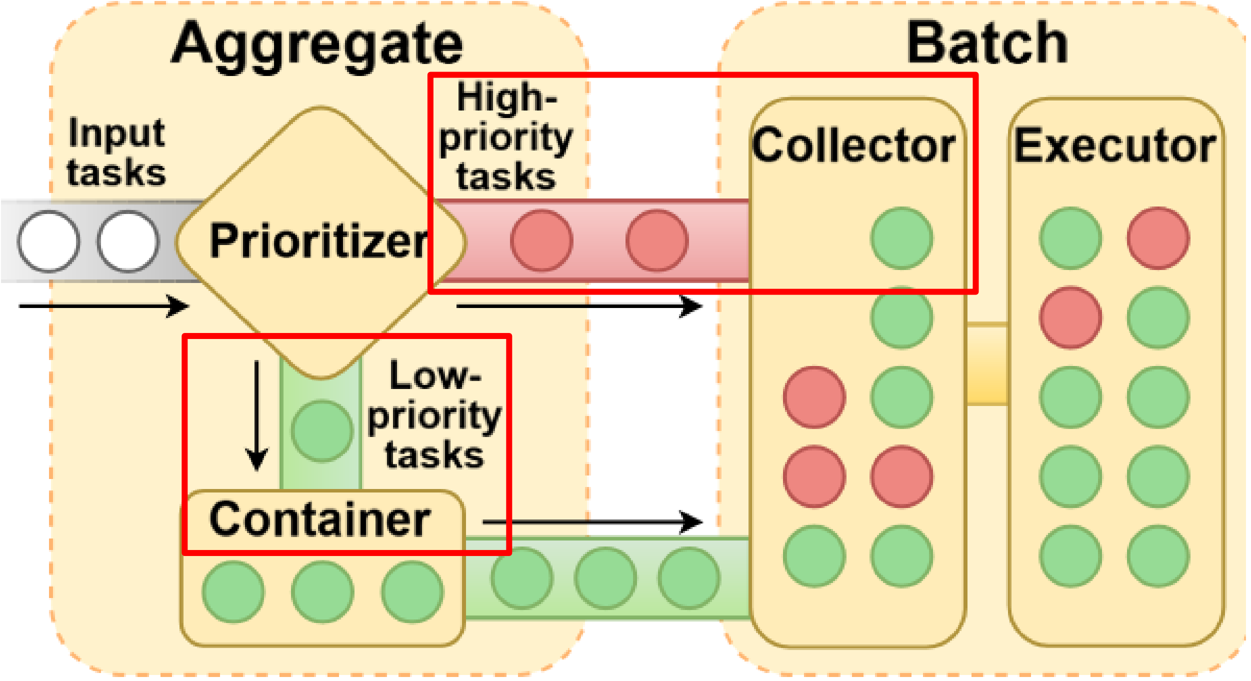


Aggregate Stage: Module 1: Prioritizer

超级科学软件实验室
Super Scientific Software Laboratory

① Receive executable tasks → ② Prioritize executable tasks → ③Provide executable tasks

③ The Prioritizer will provide executable tasks to Collector and Container according to the task priorities.



Step	1	2
P0's tasks	1F	
P0's Container		
P1's tasks		8T 2T
P1's Container		16T 3T
P2's tasks		5T 13T
P2's Container		6T 21T
P3's tasks	4F	7T 22T
P3's Container		

Example:
2nd Time Step
- 8T → Collector
- 2T, 16T→ Container
- 3T → Container

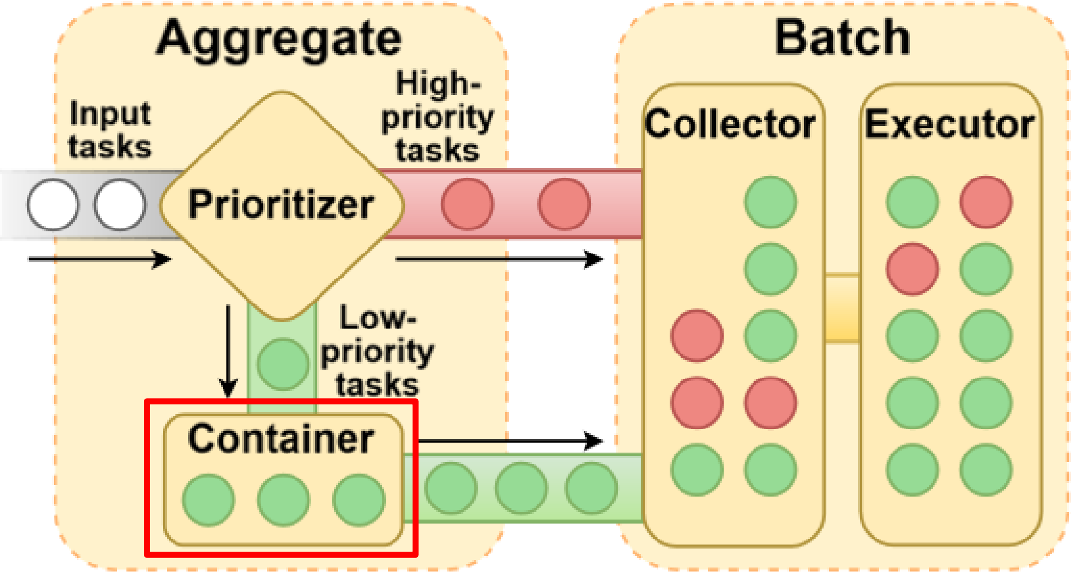
Aggregate Stage: Module 2: Container

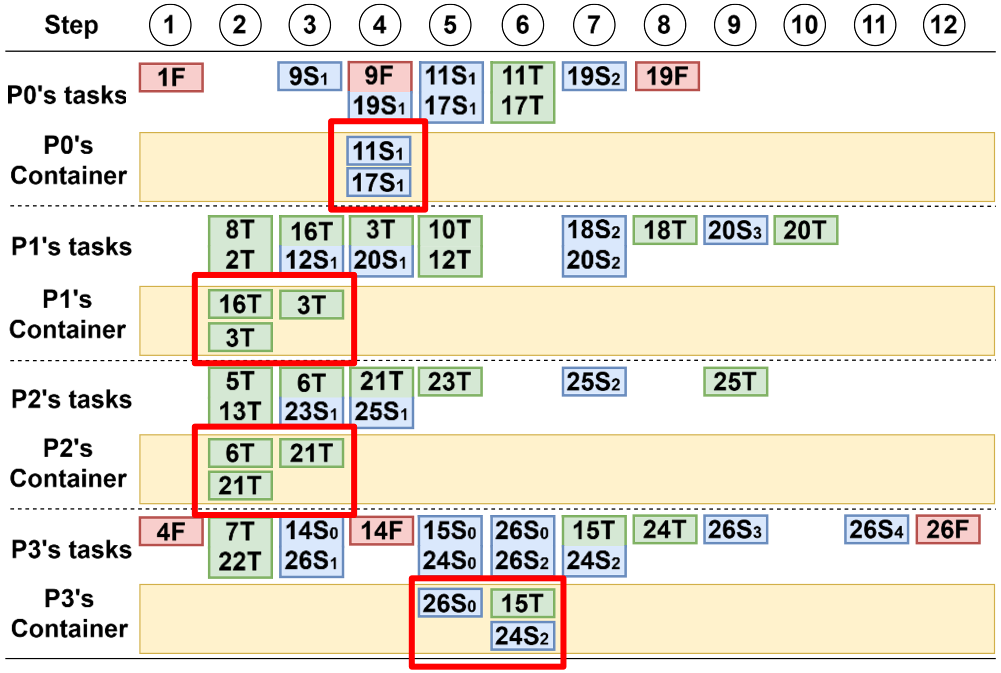
超级科学软件实验室
Super Scientific Software Laboratory

The Container is designed to:

Serve as a temporary buffer for tasks with relatively lower priority.

→ Give some tasks for batched run when GPU is not that saturated.





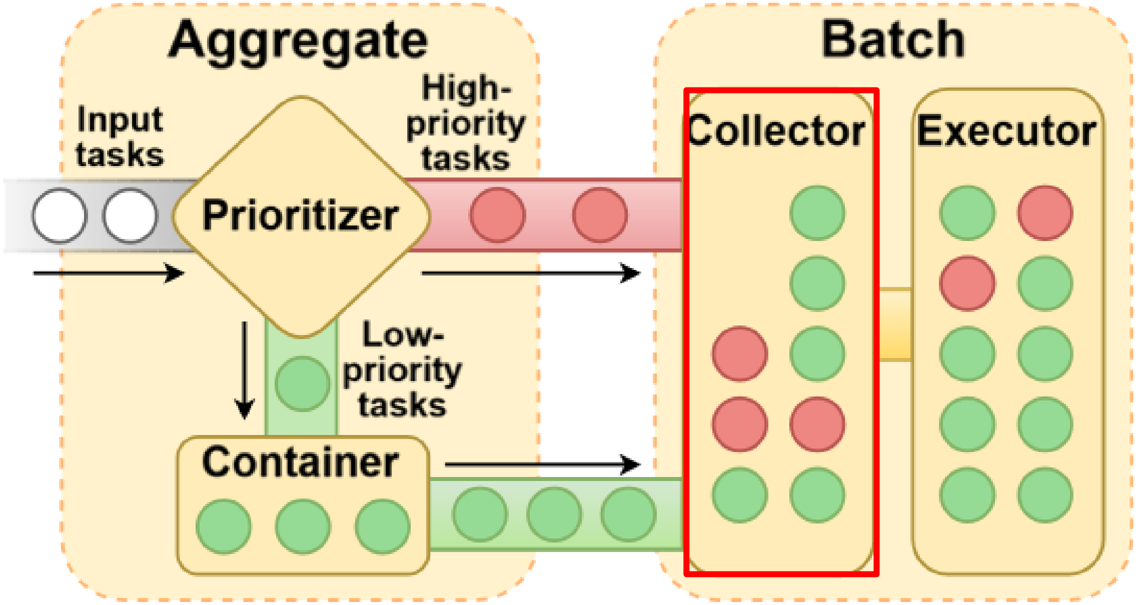
Tasks in the Container do not need to be executed immediately, therefore can be deferred to saturate the GPU when high-priority tasks are not enough in later timesteps.

Batch Stage: Module 3: Collector

The Collector is designed to :

Assemble a group of tasks and dispatch them to the GPU.

→ Schedule workload to saturate the GPU's resources.



The count of the tasks collected by the Collector is dynamically determined:

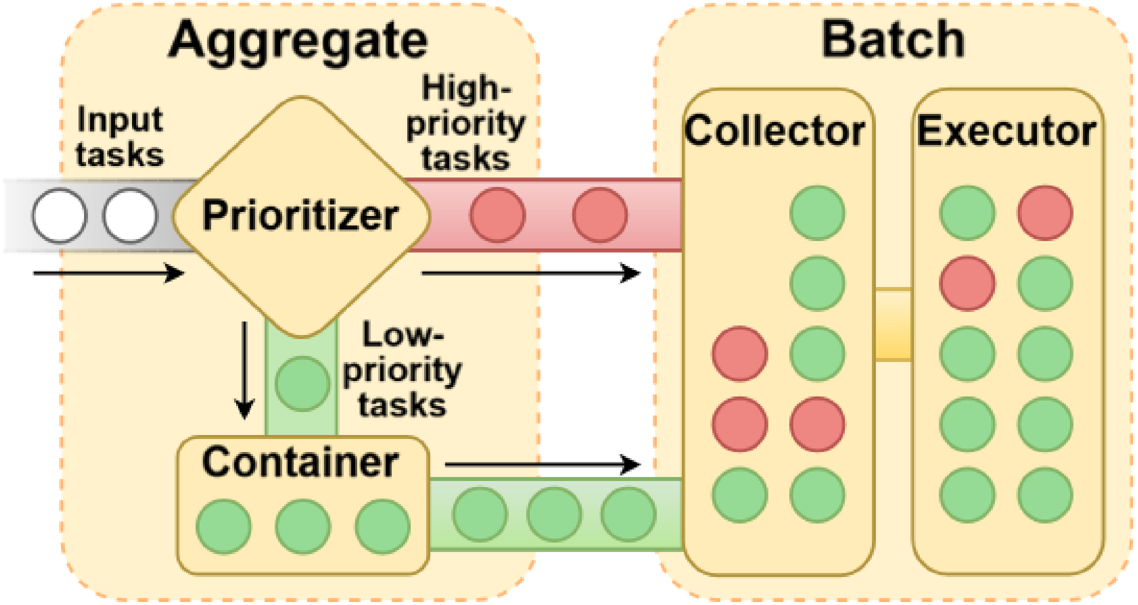
- While the Executor runs tasks, the Collector will collect one more group of tasks.
- The Collector will collect more tasks on faster GPUs, and less tasks for slower GPUs.

Batch Stage: Module 3: Collector

The Collector is designed to :

Assemble a group of tasks and dispatch them to the GPU.

→ Schedule workload to saturate the GPU's resources.



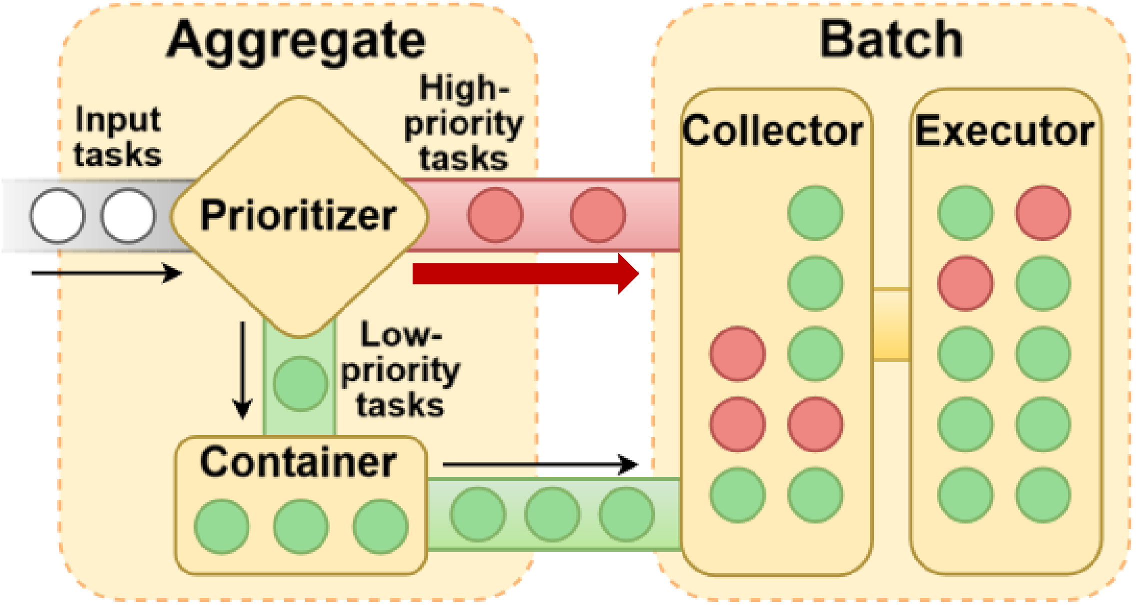
Two phases of task collecting

Batch Stage: Module 3: Collector

The Collector is designed to :

Assemble a group of tasks and dispatch them to the GPU.

→ Schedule workload to saturate the GPU's resources.



Two phases of task collecting

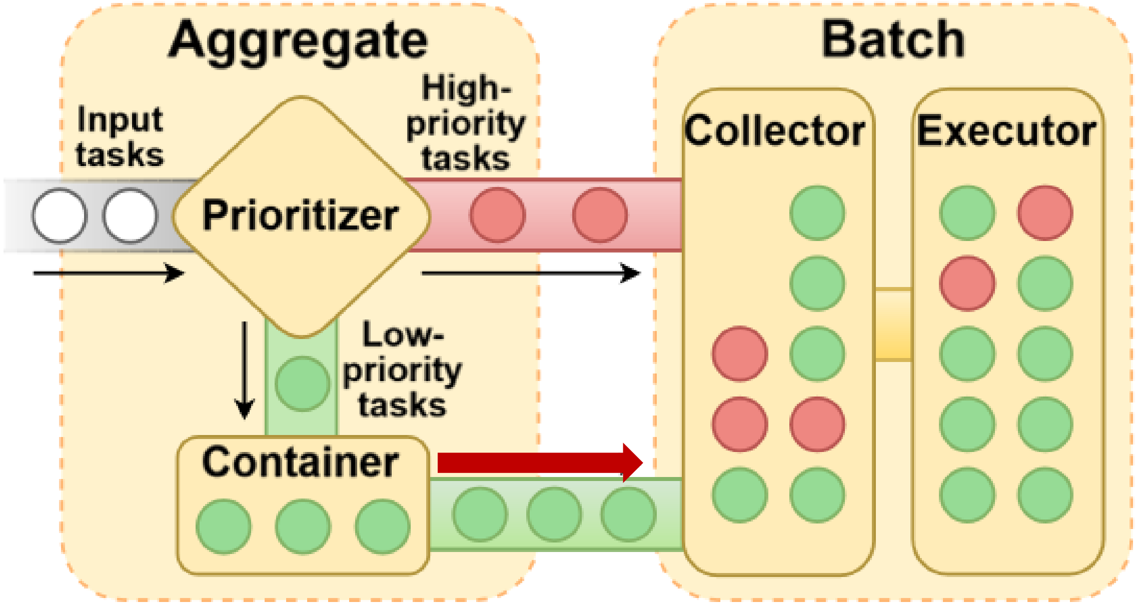
Receive tasks from Prioritizer

Batch Stage: Module 3: Collector

The Collector is designed to :

Assemble a group of tasks and dispatch them to the GPU.

→ Schedule workload to saturate the GPU's resources.



Two phases of task collecting

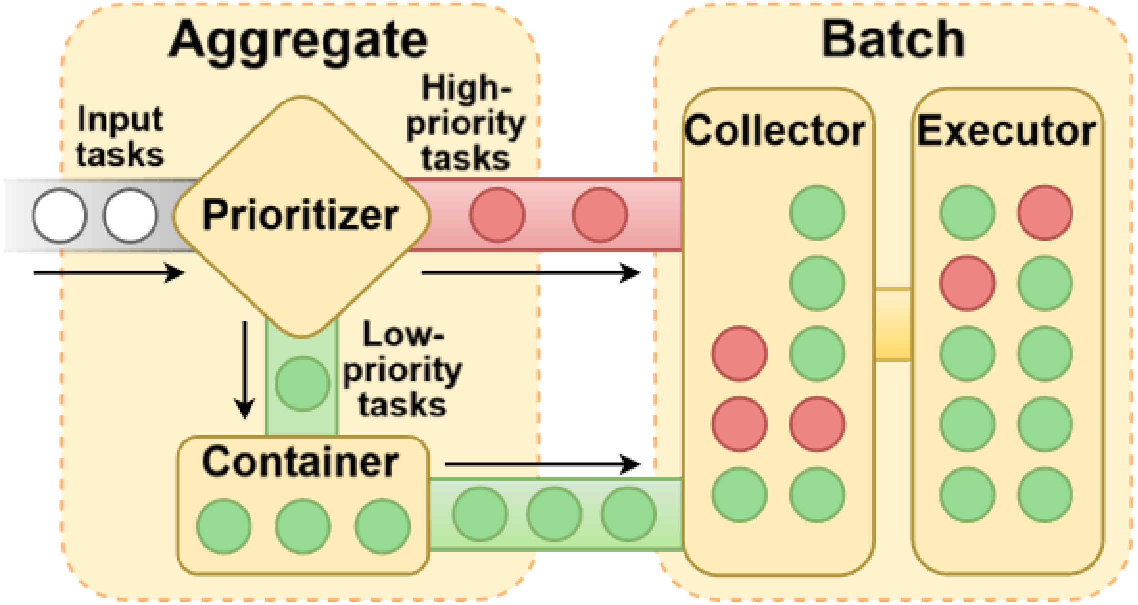
Receive tasks from Prioritizer	Fetch tasks from Container

Batch Stage: Module 3: Collector

The Collector is designed to :

Assemble a group of tasks and dispatch them to the GPU.

→ Schedule workload to saturate the GPU's resources.



Two phases of task collecting

Receive tasks from Prioritizer

Fetch tasks from Container

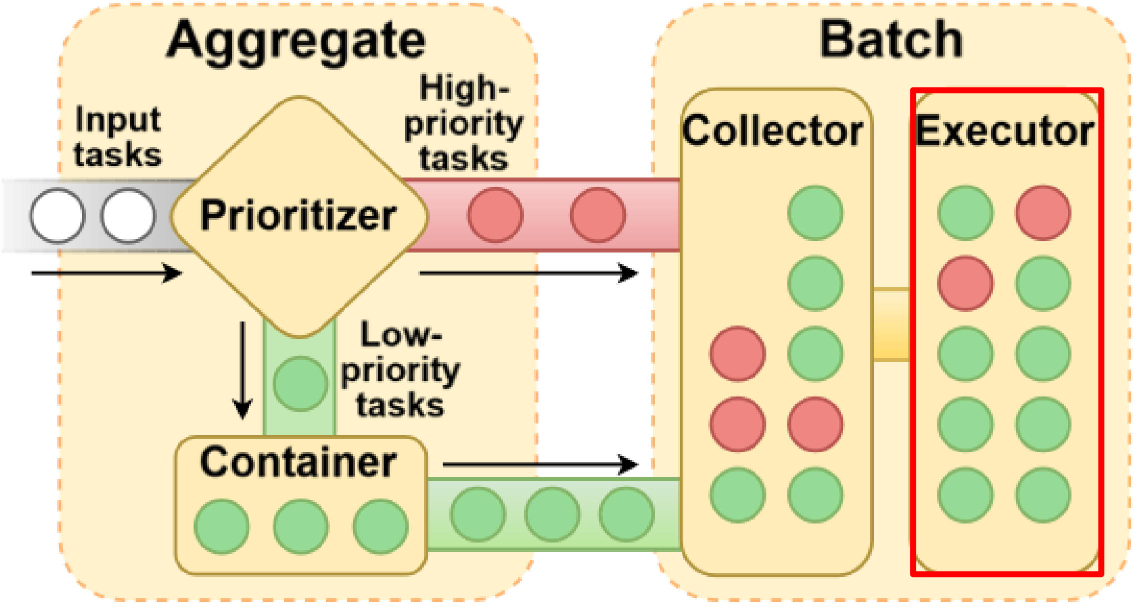
To ensure the most urgent tasks are considered first.

Batch Stage: Module 4: Executor

The Executor is designed to:

Provide a flexible batched kernel.

→ Let heterogeneous tasks run simultaneously.



The heterogeneity of the tasks is reflected in:

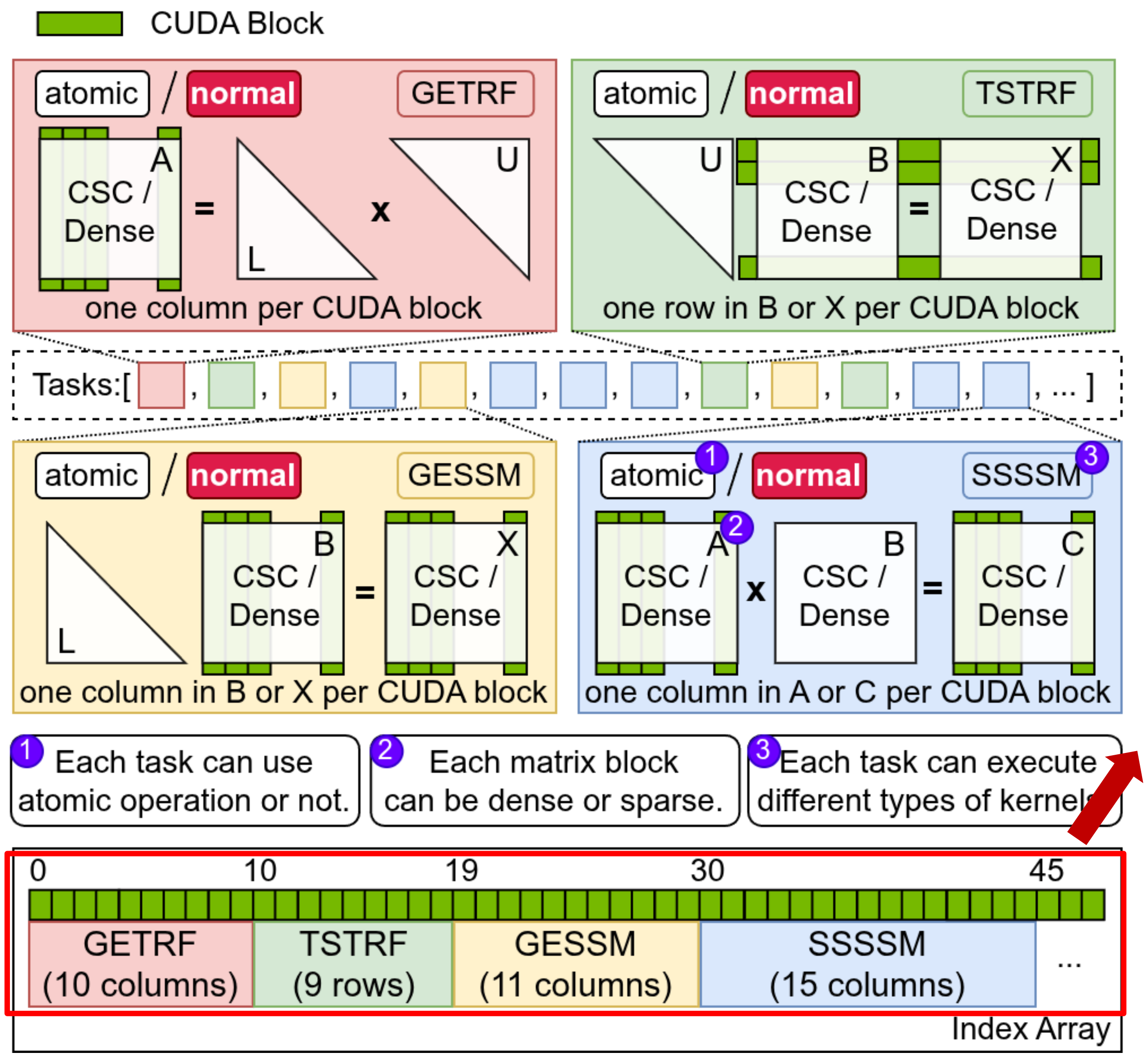
① Each task can execute different types of kernels:

- GETRF (LU factorisation)
- TSTRF/GESSM (triangular solve)
- SSSSM (Schur complement matrix multiplication)

② Each matrix block can be dense or sparse.

③ Whether or not need to handle write conflicts.

Batch Stage: Module 4: Executor



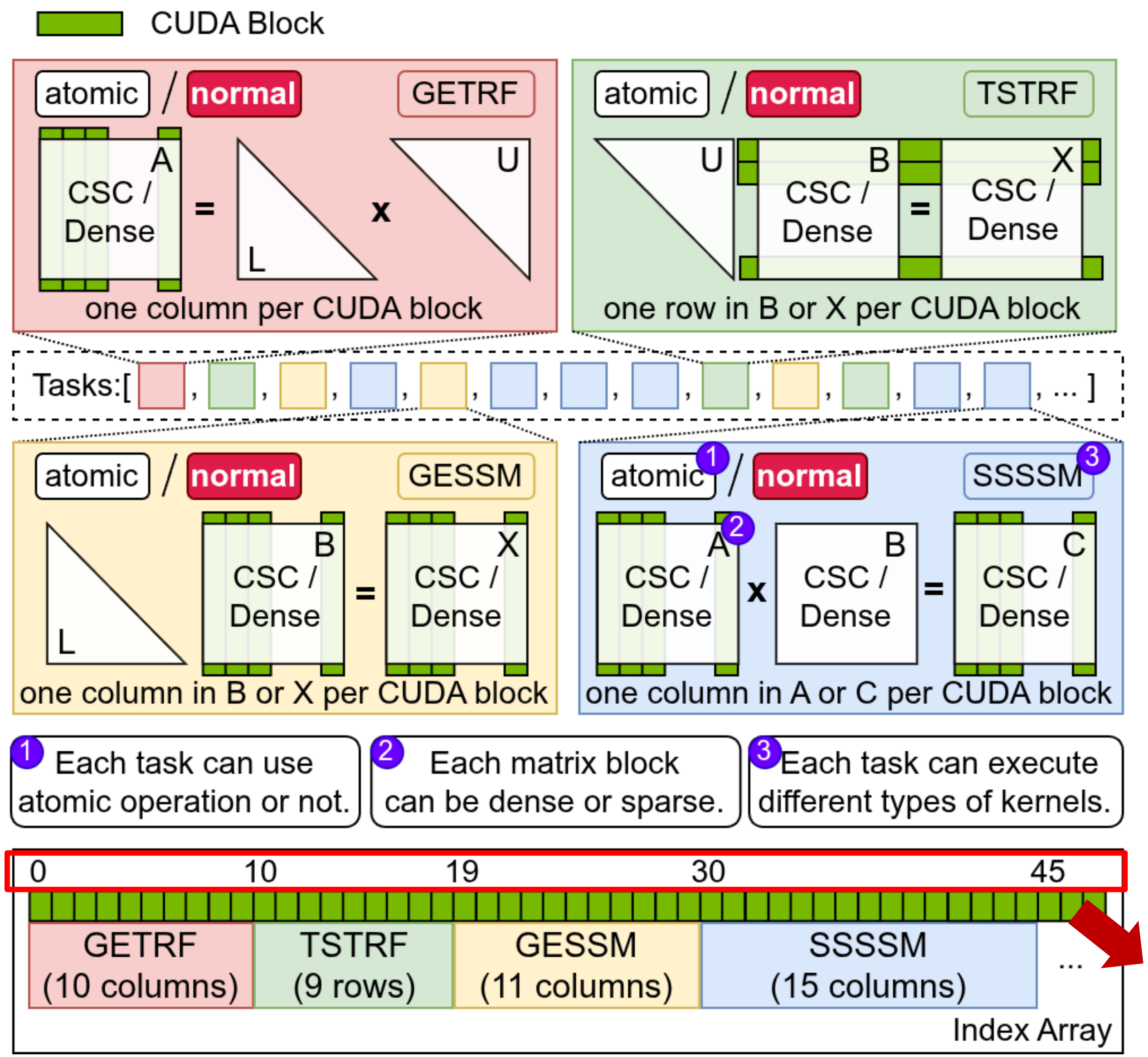
① Each task can execute different types of kernels: GETRF, TSTRF, GESSM, SSSSM

② Each block can be dense or sparse.

③ Configurable whether to enable atomic operations to resolve write conflicts.

To execute multiple tasks in one CUDA kernel, we map tasks to CUDA blocks.
Before launching the kernel, we prepare an array to store each task's starting block index.

Batch Stage: Module 4: Executor



① Each task can execute different types of kernels: GETRF, TSTRF, GESSM, SSSSM

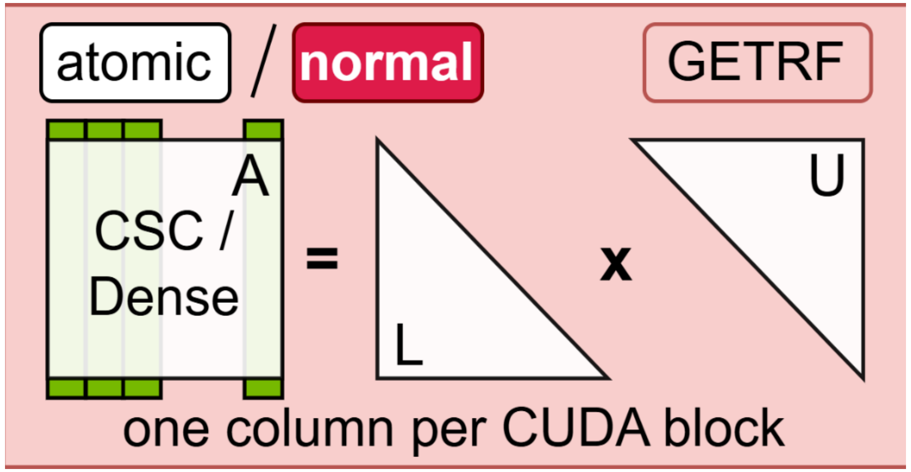
② Each block can be dense or sparse.

③ Configurable whether to enable atomic operations to resolve write conflicts.

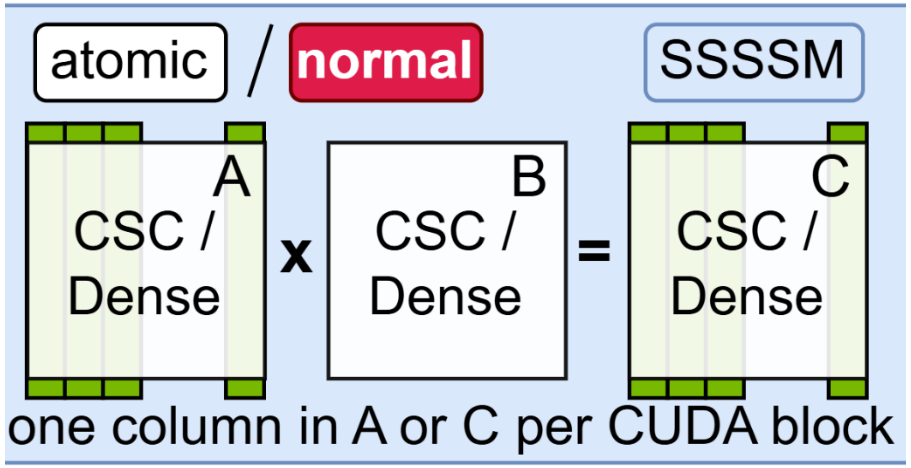
To execute multiple tasks in one CUDA kernel, we map tasks to CUDA blocks.
Before launching the kernel, we prepare an array to store each task's starting block index.

During execution, CUDA blocks use their block indices to identify their tasks.

Batch Stage: Module 4: Executor

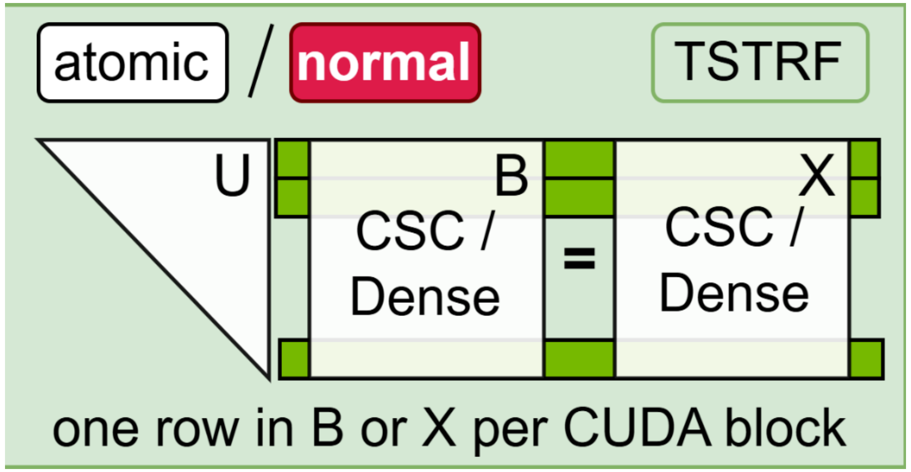


The GETRF tasks assign one CUDA block per column, adopting a synchronisation-free left-looking approach for LU factorisation.

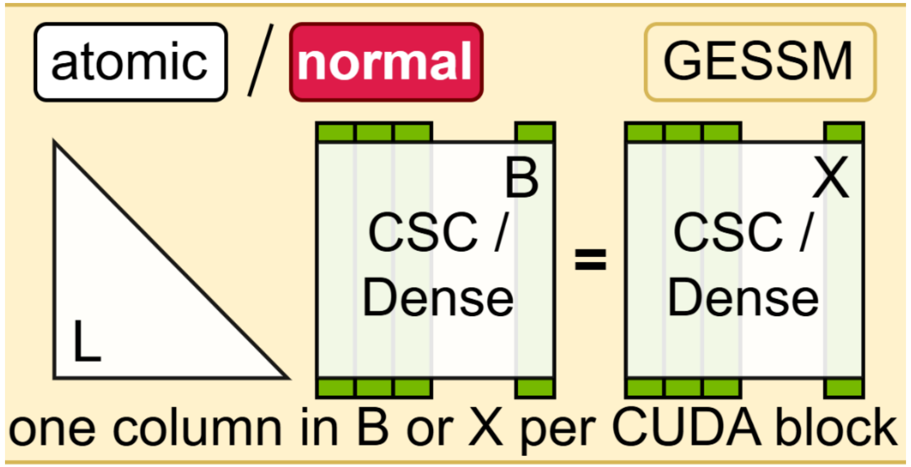


The SSSSM tasks employ a column-column multiplication method, where each element of matrix *B* independently multiplies an entire column of matrix *A*.

Batch Stage: Module 4: Executor



The TSTRF tasks assign each CUDA block to a row of *B*. Each thread holds an element in column of *U* and multiply it with corresponding row element of *B*.



The GESSM tasks assign each CUDA block to a column of *B*. Each thread holds an element in row of *L* and multiply it with corresponding column element of *B*.

Outline

- Background
  - ① Sparse LU Factorisation
  - ② Task Dependencies Restrict Concurrency
  - ③ Single Task Is Too Small For a GPU
- Motivations
  - ① Aggregate: to Prepare More Tasks for a GPU
  - ② Batch: to Selectively Run the Tasks in Parallel
- Trojan Horse
  - ① Overview
  - ② An Example to Use the Trojan Horse
  - ③ Functional Modules of the Trojan Horse (Priortizer, Container, Collector & Executor)
- Experiments
  - ① Experimental Setup
  - ② Scale-Up Evaluation
  - ③ Scale-Out Evaluation
  - ④ Comparison with CPU solvers
- Conclusion

Experimental Setup

We evaluate these solver variants (in double precision):

(1) SuperLU_DIST 9.1.0 without/with Trojan Horse (max supernode size : 256)
(2) PanguLU 5.0.0 without/with Trojan Horse (block size: 512)
(3) PanguLU 5.0.0 with multiple CUDA streams (block size: 512)
(4) PaStiX 6.4.0 with StarPU 1.4.8
(5) MUMPS 5.6.0

We compiled above solvers with three major libraries:

(1) CUDA 12.8 (on NVIDIA GPUs) / ROCm 4.3 (on AMD GPUs)
(2) Intel MPI 2021.1
(3) OpenBLAS 0.3.29

The evaluation is conducted in three parts:

(1) Scale-up, on a single GPU (NVIDIA RTX 5060Ti, RTX 5090 and A100)
(2) Scale-out, on distributed multiple GPUs (16-card NVIDIA H100 and 16-card AMD MI50)
(3) Comparison with modern CPU (32-core Intel Xeon Gold 6462C)

Outline

超级科学软件实验室
Super Scientific Software Laboratory

- Background
  - ① Sparse LU Factorisation
  - ② Task Dependencies Restrict Concurrency
  - ③ Single Task Is Too Small For a GPU
- Motivations
  - ① Aggregate: to Prepare More Tasks for a GPU
  - ② Batch: to Selectively Run the Tasks in Parallel
- Trojan Horse
  - ① Overview
  - ② An Example to Use the Trojan Horse
  - ③ Functional Modules of the Trojan Horse (Priortizer, Container, Collector & Executor)
- Experiments
  - ① Experimental Setup
  - ② Scale-Up Evaluation
  - ③ Scale-Out Evaluation
  - ④ Comparison with CPU solvers
- Conclusion

Scale-Up: Experimental Setup

超级科学软件实验室
Super Scientific Software Laboratory

GPU platforms: RTX 5060Ti and RTX 5090 GPUs share the same architecture but differ in theoretical performance (~4x FP64 peak performance difference, and ~4x memory bandwidth difference), enabling an effective scale-up evaluation.

GPU	#CUDA 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

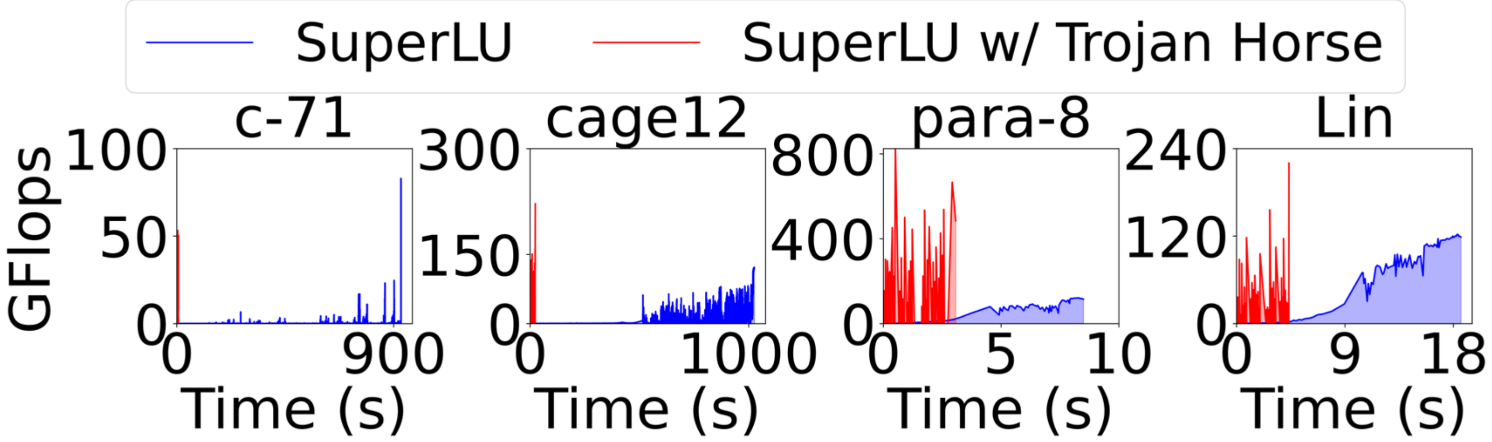
Scale-Up: Experimental Setup

The four matrices used are from SuiteSparse, and were selected in prior SuperLU and PanguLU benchmarks. These matrices are of moderate size, sufficient for GPU parallelism, yet small enough to be tested on a single GPU.

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

Performance Evaluation of Kernels

We trace the numeric factorisation phase of SuperLU and PanguLU without (blue lines) and with (red lines) the Trojan Horse. The performance of each kernel execution are shown in these figures.

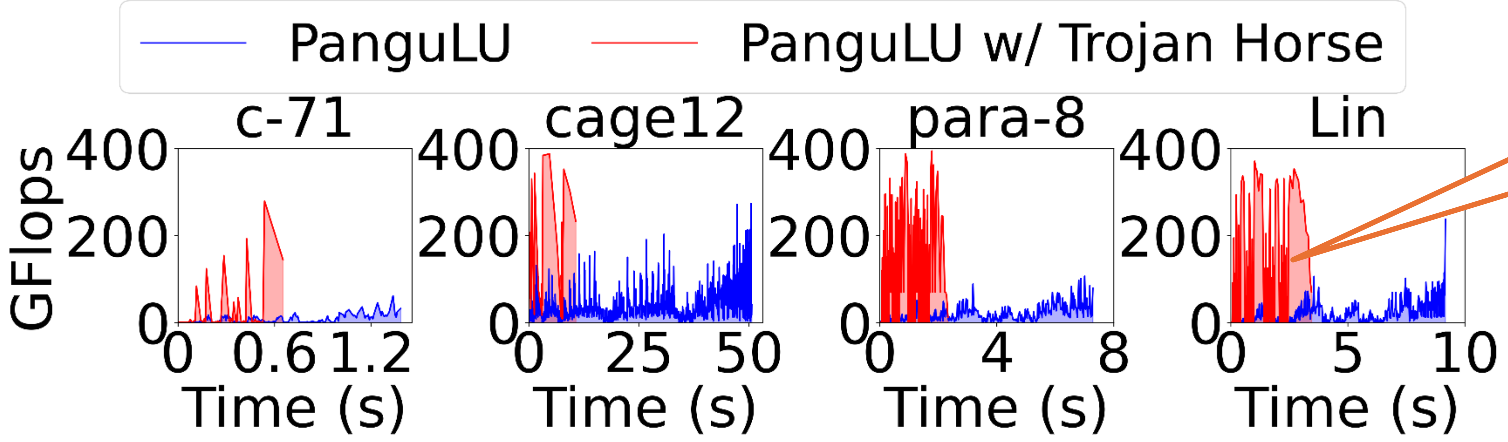


Kernel Speedup

Average: 1.2x

Maximum: 2.0x

The solvers integrated with Trojan Horse has higher average performance (y-axis) and completes factorisation faster (x-axis).



Kernel Speedup

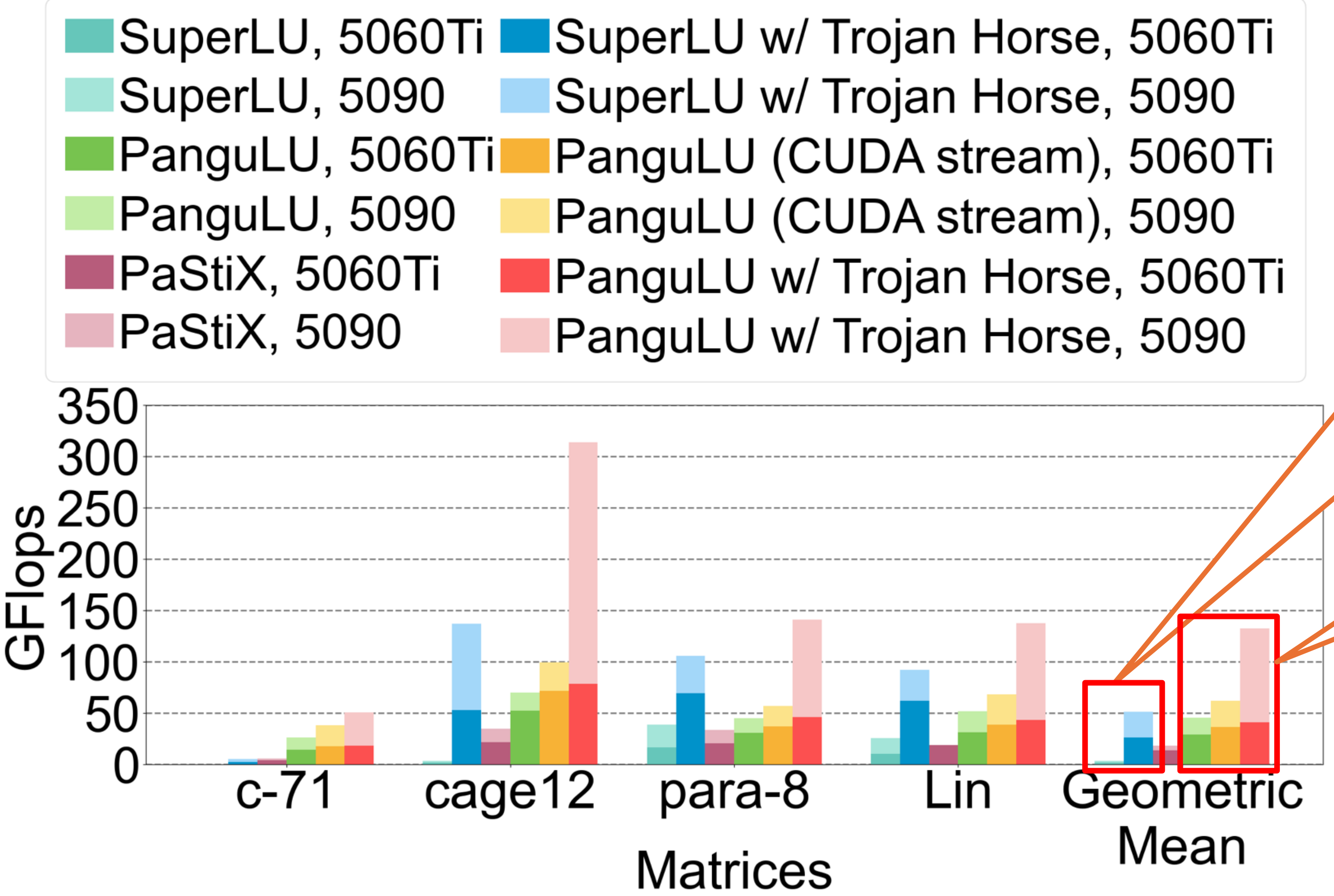
Average: 2.5x

Maximum: 2.8x

GPU: RTX 5090

Scale-Up Evaluation on RTX 5060Ti and 5090

The following figure illustrates the performance of different solvers on four example matrices.



SuperLU Without Trojan Horse: 1.09x,
SuperLU With Trojan Horse: 1.94x.

PanguLU Without Trojan Horse: 1.57x,
PanguLU With Trojan Horse: 3.25x.

Trojan Horse improve the performance gain attained from faster GPUs.

Reduction in the Count of Kernel Executions

超级科学软件实验室
Super Scientific Software Laboratory

Our aggregate stage reduces the kernel execution count, providing substantial tasks to batch.

Matrix	Kernel count w/o Trojan Horse	Kernel count 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%

SuperLU

Matrix	Kernel count w/o Trojan Horse	Kernel count 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%

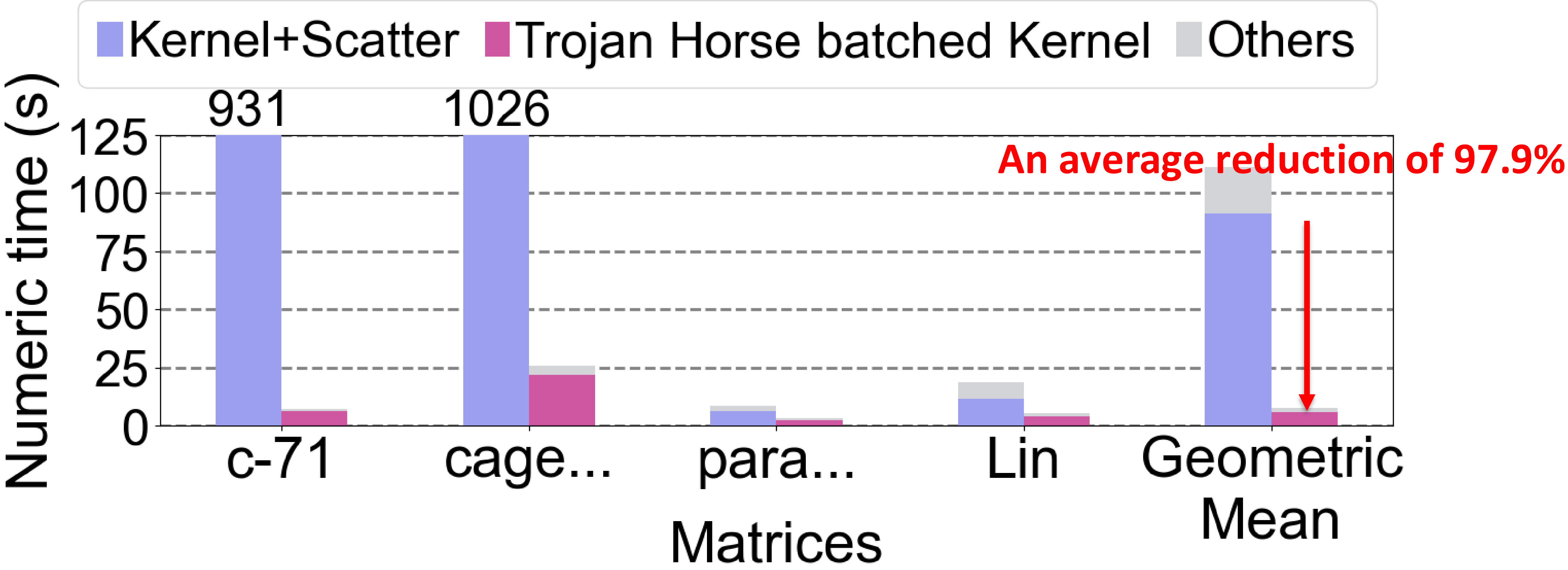
PanguLU

For both SuperLU and PanguLU, the number of kernel execution decrease by about two orders of magnitude.

Reduction in Kernel Runtime of SuperLU

超级科学软件实验室
Super Scientific Software Laboratory

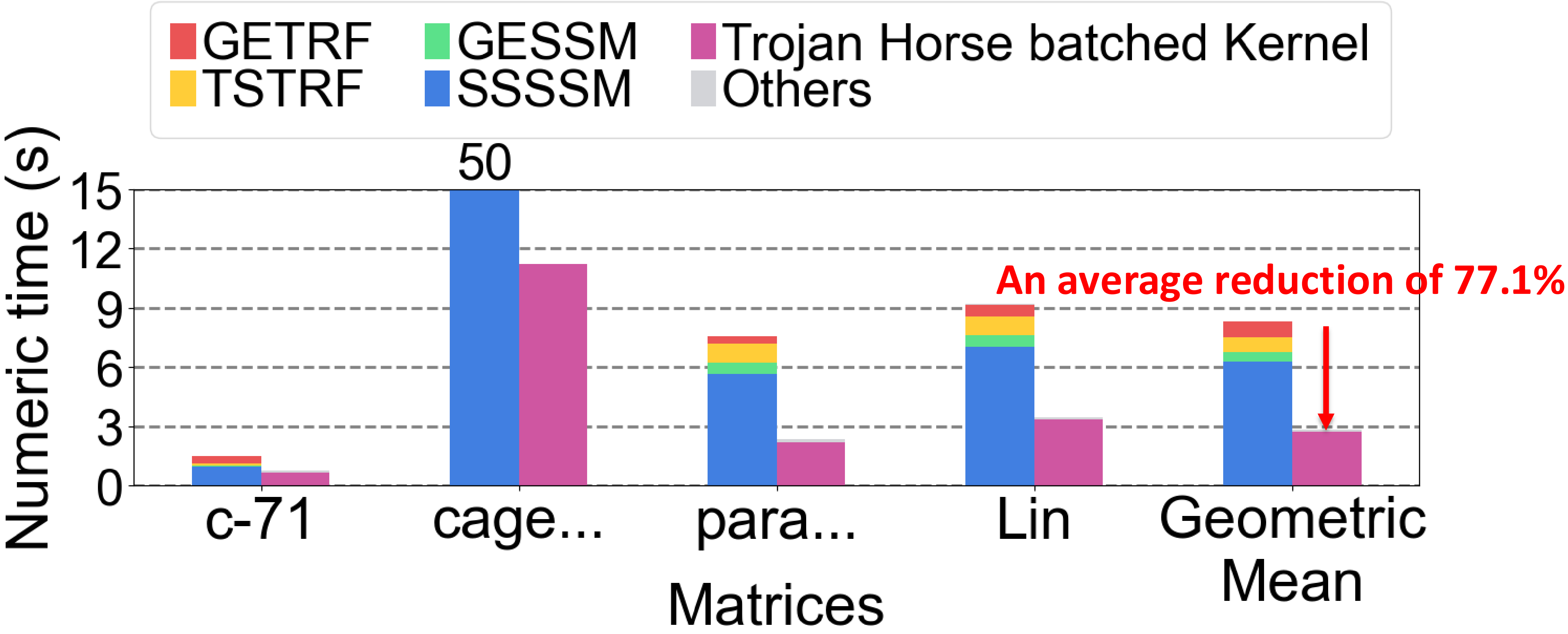
The Trojan Horse batched kernel reduces the kernel execution time. The comparison of kernel execution times between SuperLU without and with Trojan Horse is shown below.



Reduction in Kernel Runtime of PanguLU

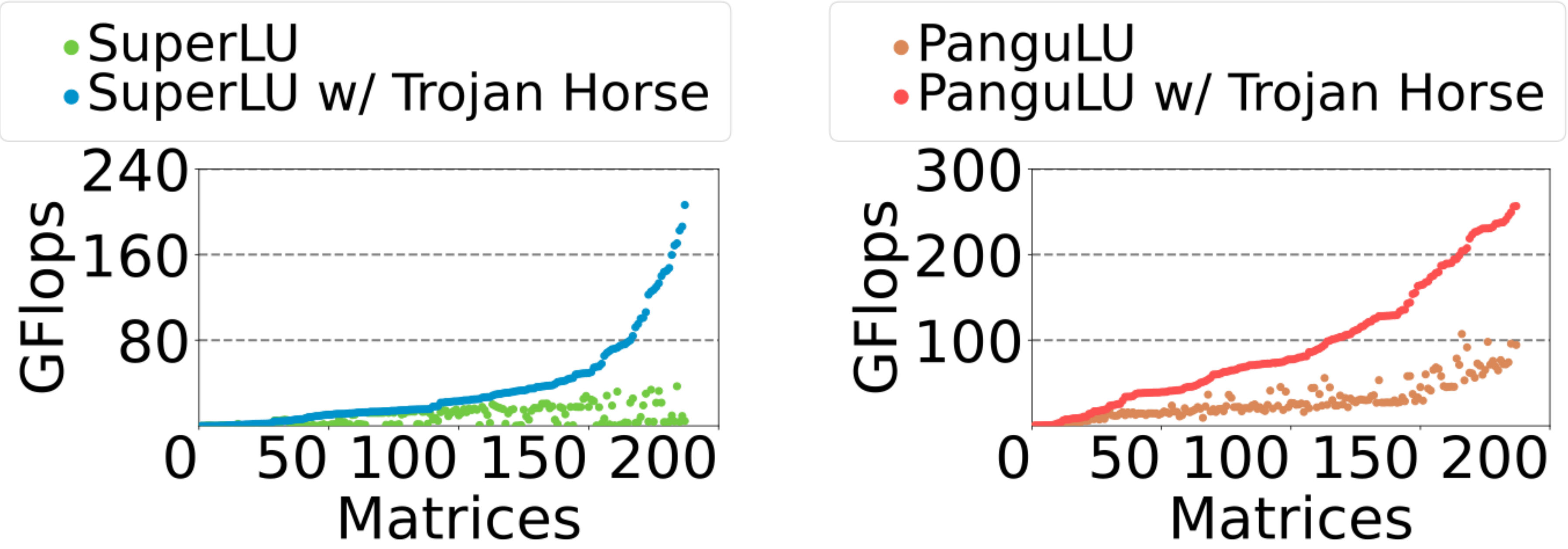
超级科学软件实验室
Super Scientific Software Laboratory

The Trojan Horse batched kernel reduces the kernel execution time. The comparison of kernel execution times between PanguLU without and with Trojan Horse is shown below.



Scale-Up Evaluation on 200 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. 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.



Performance evaluation of Trojan Horse on 200 square matrices from SuiteSparse (GPU: an NVIDIA A100)

Outline

- Background
  - ① Sparse LU Factorisation
  - ② Task Dependencies Restrict Concurrency
  - ③ Single Task Is Too Small For a GPU
- Motivations
  - ① Aggregate: to Prepare More Tasks for a GPU
  - ② Batch: to Selectively Run the Tasks in Parallel
- Trojan Horse
  - ① Overview
  - ② An Example to Use the Trojan Horse
  - ③ Functional Modules of the Trojan Horse (Priortizer, Container, Collector & Executor)
- Experiments
  - ① Experimental Setup
  - ② Scale-Up Evaluation
  - ③ Scale-Out Evaluation
  - ④ Comparison with CPU solvers
- Conclusion

Scale-Out: Experimental Setup

The hardware configuration is shown below. The 16-card H100 GPUs are evenly distributed on two nodes, and the 16-card MI50 GPUs are evenly distributed on four nodes.

16 GPUs	#CUDA Cores	FP64 peak	Memory	B/W
H100 SXM	14592	25.61 TFlops	80 GB	2.04 TB/s
MI50 PCIe	3840	6.71 TFlops	16 GB	1.02 TB/s

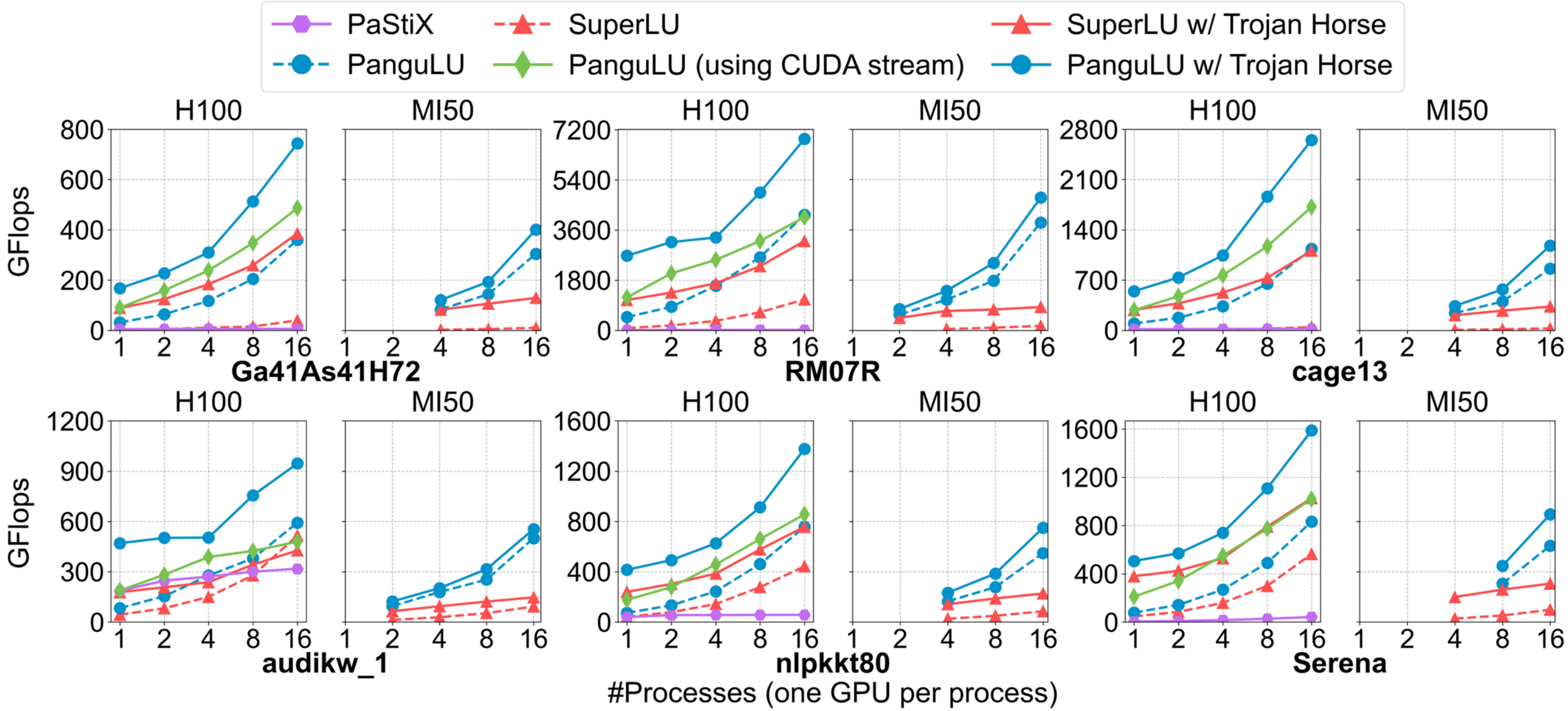
The six matrices used are from SuiteSparse and have been widely employed in existing works on SuperLU and PanguLU.

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

Scale-Out Evaluation

超级科学软件实验室
Super Scientific Software Laboratory

Both SuperLU and PanguLU with Trojan Horse continue to deliver strong performance gains as the number of GPUs increases, despite the workload per GPU is reduced.



Speedup on 16-card H100	
Average	Maximum
SuperLU w/ Trojan Horse	
3.5x	24.6x
PanguLU w/ Trojan Horse	
1.9x	2.3x

Speedup on 16-card MI50	
Average	Maximum
SuperLU w/ Trojan Horse	
4.7x	12.8x
PanguLU w/ Trojan Horse	
1.3x	1.4x

Outline

- Background
  - ① Sparse LU Factorisation
  - ② Task Dependencies Restrict Concurrency
  - ③ Single Task Is Too Small For a GPU
- Motivations
  - ① Aggregate: to Prepare More Tasks for a GPU
  - ② Batch: to Selectively Run the Tasks in Parallel
- Trojan Horse
  - ① Overview
  - ② An Example to Use the Trojan Horse
  - ③ Functional Modules of the Trojan Horse (Priortizer, Container, Collector & Executor)
- Experiments
  - ① Experimental Setup
  - ② Scale-Up Evaluation
  - ③ Scale-Out Evaluation
  - ④ Comparison with CPU solvers
- Conclusion

Comparison with CPU solvers

超级科学软件实验室
Super Scientific Software Laboratory

The memory B/W and peak FP64 performance of a GPU are higher than a CPU by about an order of magnitude.

GPU

Peak FP64 Performance: ~25 Tflops
Memory B/W: ~2 TB/s

CPU

Peak FP64 Performance: ~3 Tflops
Memory B/W: ~0.2 TB/s

Comparison with CPU solvers

超级科学软件实验室
Super Scientific Software Laboratory

As shown in the red box, over the past twenty years, sparse direct methods are far from saturating modern GPUs because of highly independent small tasks. Sparse direct methods on GPU has been slower than CPU methods.

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

For each matrix, **underlined and bold** represents the fastest result, and **bold** represents the second-fastest result.
CPU: Intel Xeon Gold 6462C (32 cores) GPU: NVIDIA H100

Comparison with CPU solvers

As shown in the red box, today, SuperLU_DIST and PanguLU on GPU with Trojan Horse are comparable to or faster than CPU methods.

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

For each matrix, **underlined and bold** represents the fastest result, and **bold** represents the second-fastest result.
CPU: Intel Xeon Gold 6462C (32 cores) GPU: NVIDIA H100

Outline

- Background
  - ① Sparse LU Factorisation
  - ② Task Dependencies Restrict Concurrency
  - ③ Single Task Is Too Small For a GPU
- Motivations
  - ① Aggregate: to Prepare More Tasks for a GPU
  - ② Batch: to Selectively Run the Tasks in Parallel
- Trojan Horse
  - ① Overview
  - ② An Example to Use the Trojan Horse
  - ③ Functional Modules of the Trojan Horse (Priortizer, Container, Collector & Executor)
- Experiments
  - ① Experimental Setup
  - ② Scale-Up Evaluation
  - ③ Scale-Out Evaluation
  - ④ Comparison with CPU solvers
- Conclusion

Conclusion

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

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.

PPoPP 2026

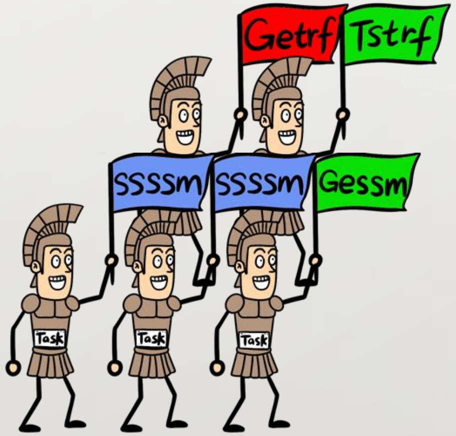
Trojan Horse： Aggregate-and-Batch for Scaling Up Sparse Direct Solvers on GPU Clusters

Open-sourced on Github: https://github.com/SuperScientificSoftwareLaboratory/TrojanHorse

Yida Li, Siwei Zhang, Yiduo Niu, Yang Du, Qingxiao Sun, Zhou Jin, Weifeng Liu

Super Scientific Software Laboratory (SSSLab)

China University of Petroleum-Beijing



THANKS