EURO-PAR
CONFERENCE 2026

SET: Stream-Event-Triggered Scheduling for Efficient CUDA Graph Pipelines

Presenter: Tsung-Wei Huang

Authors: Zhengxiong Li, Tsung-Wei Huang, Umit Ogras

International European Conference on Parallel and Distributed Computing (Euro-Par), 2026

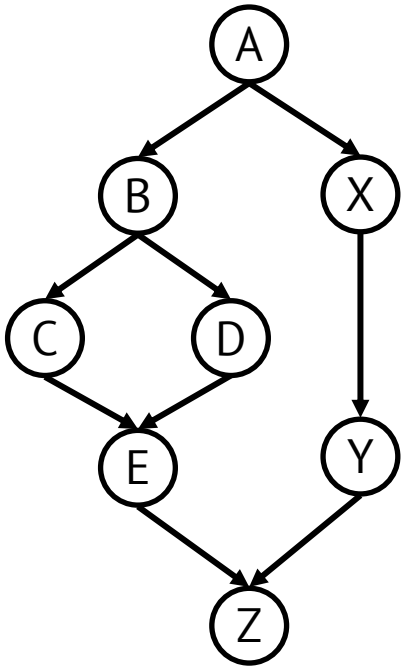
Aug. 28, Pisa, Italy

Outline

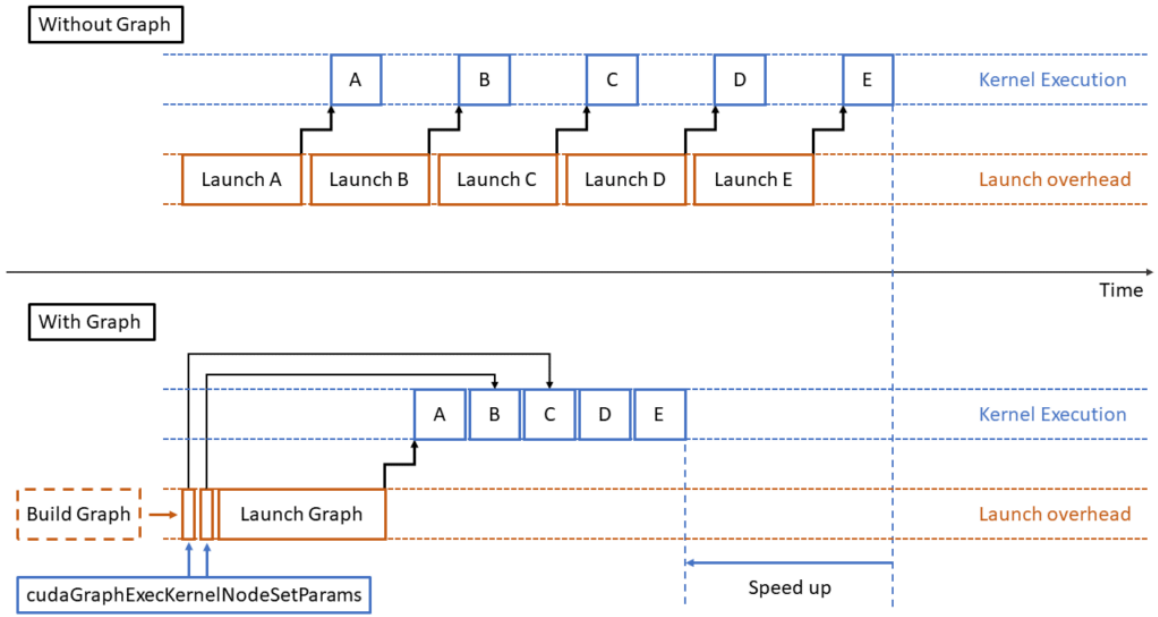
- **Introduction**
  - What is CUDA Graph
  - Performance gaps in execution flow
- **Analytical Model**
  - Bottleneck breakdown
  - Time in different cases
- **Our Proposal: SET**
  - Overview
  - Key components
- **Evaluations**
- **Takeaways**

Introduction

- **What is a CUDA Graph**
  - CUDA Graph captures kernel launch dependencies and replays them
    - It greatly reduces kernel launch overhead, but GPU may still sit idle



Kernel and Dependency



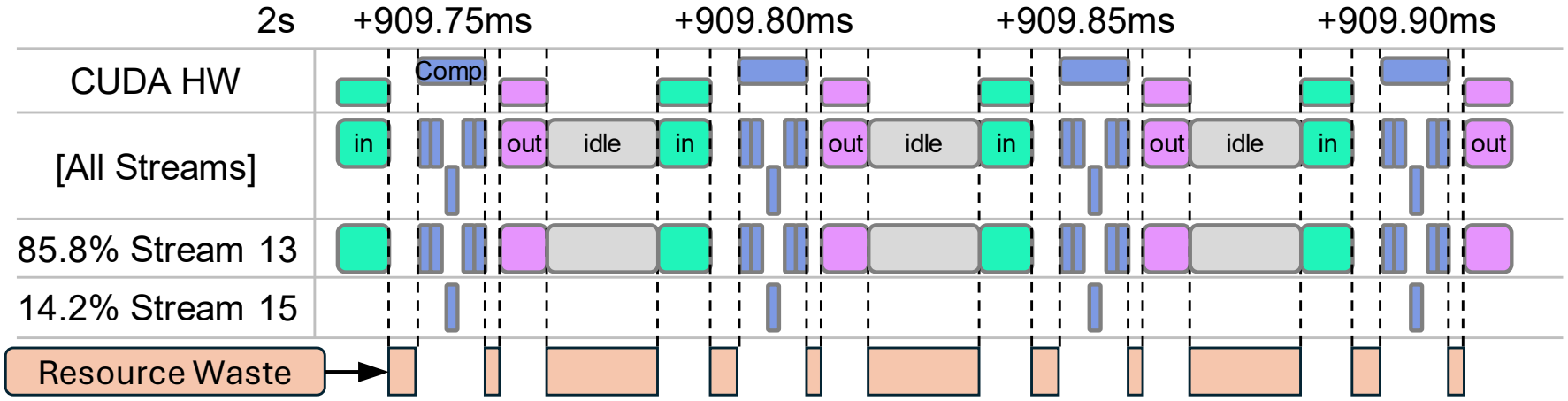
Example of how CUDA graph works

Steps:

1. Capture a directed acyclic graph (DAG) of kernels and memory operations
2. Instantiate the executable based on the captured graph
3. Replay efficiently to reduce launch overhead

Introduction

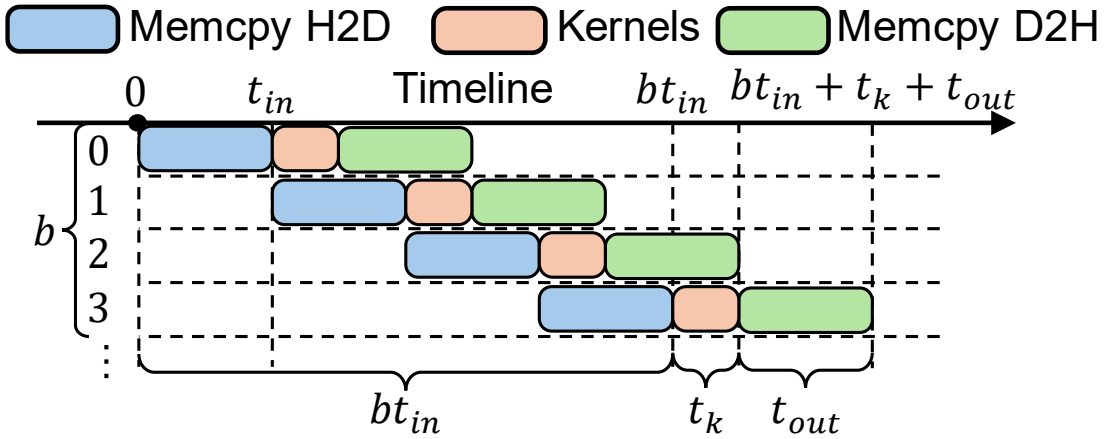
- **Performance gaps**
  - CUDA Graph does not guarantee a busy GPU
  - There are other gaps, arising from synchronization, and host-side operations, including argument updates, stream selection
  - Minimizing these gaps is crucial to realizing the potential of CUDA graph pipelines



Gaps between operations in Nsight profiler

Bottleneck Breakdown

- **Analytical model of performance gaps in ideal cases**
  - In ideal cases, there are no gaps between memory operations and kernels
  - It has two copy engines, where it can only copy one in at a time
  - The overall execution time for one job:
    - $t_{job,ideal} = t_{in} + t_k + t_{out}$
  - The overall execution time for one batch:
    - $T_{ideal} = bt_{in} + t_k + t_{out}$

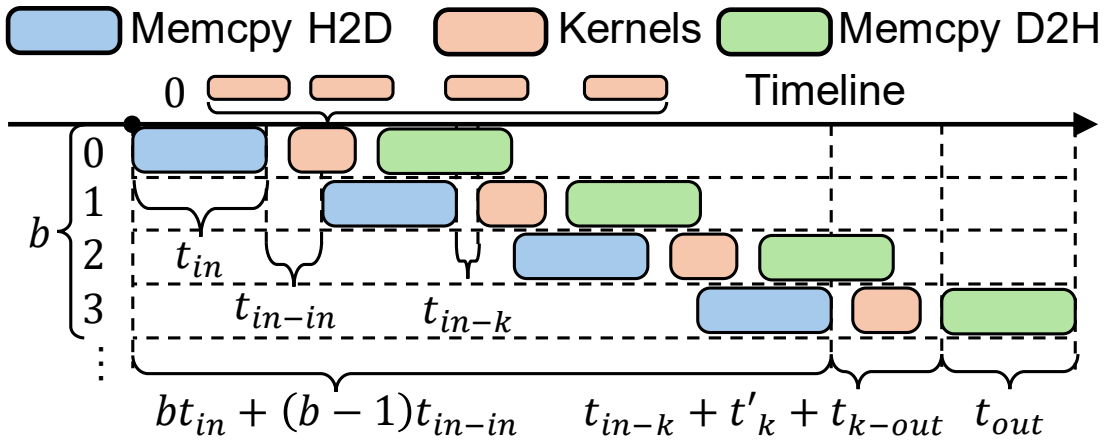


Ideal execution flow of a batch

Symbol	Definition
b	Batch size
t_{in}	Time spent on memcpy H2D
t_k	Time spent on computation
t_{out}	Time spend on memcpy D2H

Bottleneck Breakdown

- **Analytical model of performance gaps with intra-batch overheads**
  - With intra-batch overheads, there are gaps between memory operations and kernels
  - The overall execution time for one job:
    - $t_{job,intra} = t_{in} + t_{in-k} + t'_k + t_{k-out} + t_{out}$
  - The overall execution time for one batch:
    - $T_{intra} = bt_{in} + (b-1)t_{in-in} + t_{in-k} + t'_k + t_{k-out} + t_{out}$

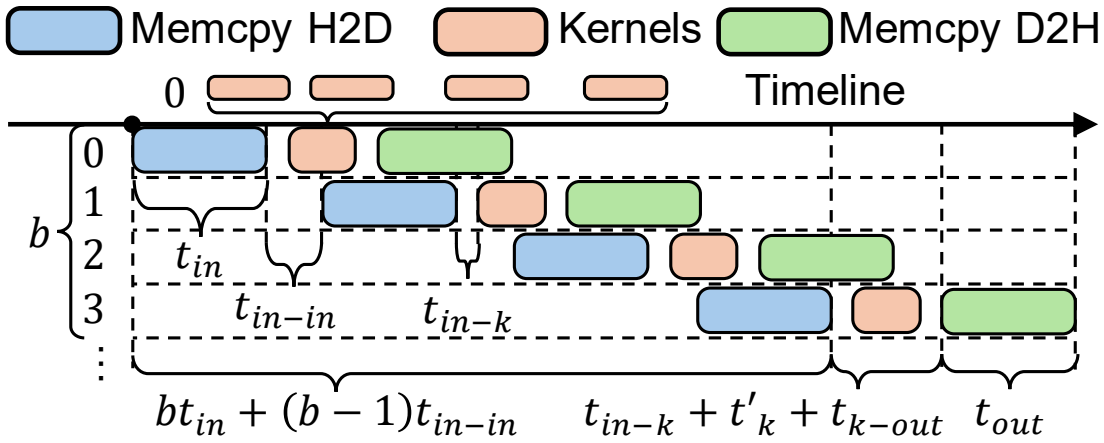


Execution flow with intra-batch overheads

Symbol	Definition
b	Batch size
t_{in}	Time spent on memcpy H2D
t'_k	Time spent on computation with gaps
t_{out}	Time spend on memcpy D2H
t_{in-in}	Time between two memcpy H2D
t_{in-k}	Time between memcpy H2D and kernels
t_{k-out}	Time between kernels and memcpy D2H

Bottleneck Breakdown

- **Analytical model of performance gaps with intra-batch overheads**
  - $T_{intra} = bt_{in} + (b-1)t_{in-in} + t_{in-k} + t'_k + t_{k-out} + t_{out}$
  - $T_{ideal} = bt_{in} + t_k + t_{out}$
  - Suppose the execution time for kernels increases by $\Delta t_k = t'_k - t_k$
  - Intra-batch overheads:
    - $t_{intra} = (b-1)t_{in-in} + t_{in-k} + \Delta t_k + t_{k-out}$

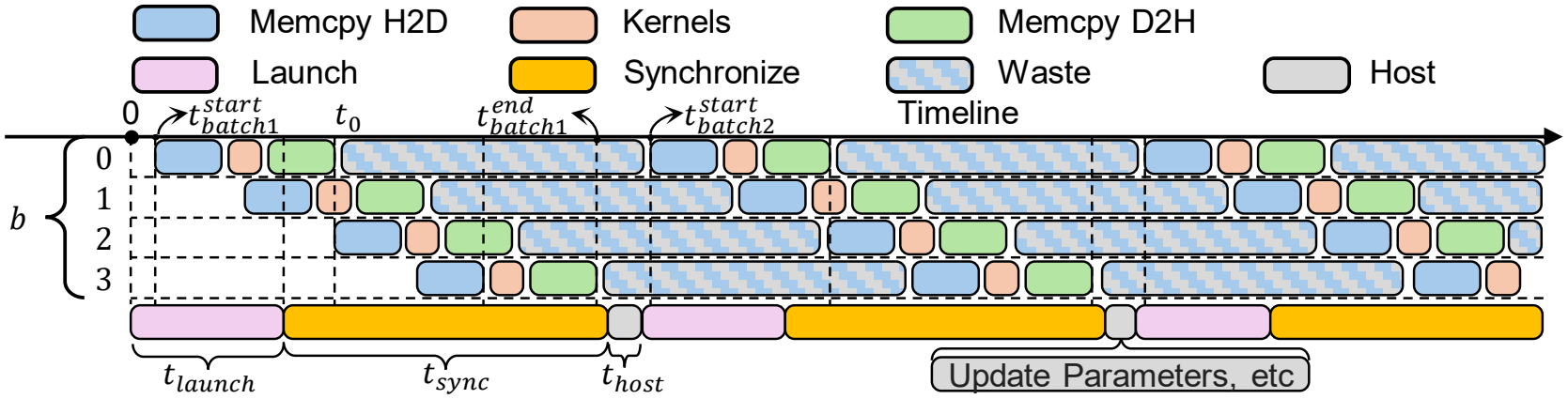


Execution flow with intra-batch overheads

Symbol	Definition
b	Batch size
t_{in}	Time spent on memcpy H2D
t'_k	Time spent on computation with gaps
t_{out}	Time spend on memcpy D2H
t_{in-in}	Time between two memcpy H2D
t_{in-k}	Time between memcpy H2D and kernels
t_{k-out}	Time between kernels and memcpy D2H

Bottleneck Breakdown

- **Analytical model of performance gaps with inter-batch overheads**
  - With inter-batch overheads, there are gaps between two batches
  - Considering both CPU and GPU, API costs are included
  - Due to the determinism of CUDA graph, the batch is treated as a whole
  - The next batch won't start unless the previous batch is done

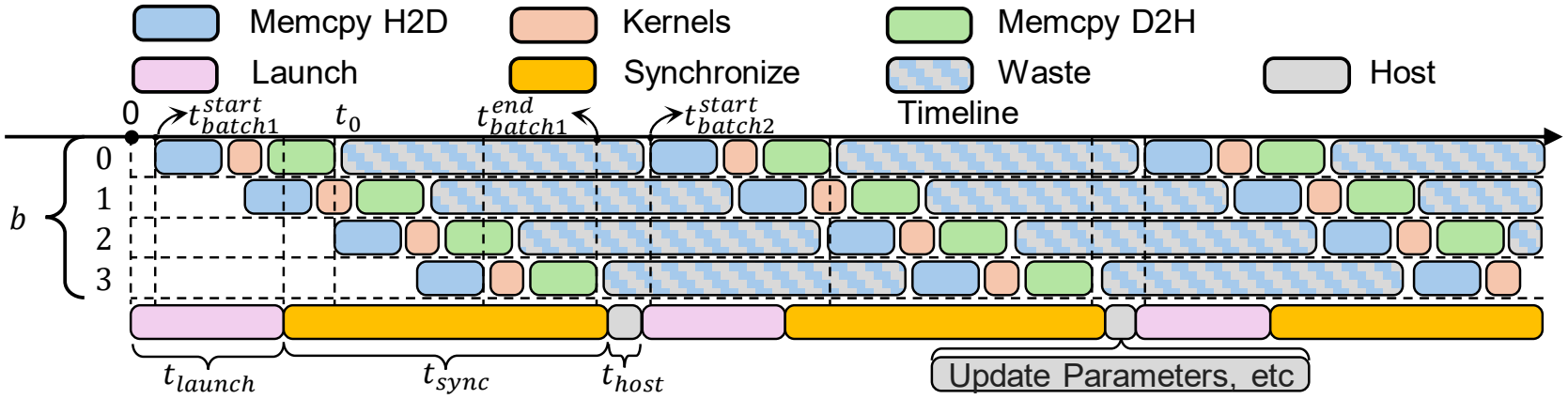


Execution flow with inter-batch overheads

Symbol	Definition
t_{launch}	Time on CUDA graph launch
t_{sync}	Time spent on synchronization
t_{host}	Time spent on host

Bottleneck Breakdown

- **Analytical model of performance gaps with inter-batch overheads**
  - First job in first batch finishes at t_0. The first batch finishes at t_{batch1}^{end}
  - However, the second batch won't start until t_{batch2}^{start}
  - Hardware resources are occupied and wasted between t_0 and t_{batch2}^{start}
  - Inter-batch overheads:
    - $t_{inter} = t_{batch2}^{start} - t_{batch1}^{end}$



Execution flow with inter-batch overheads

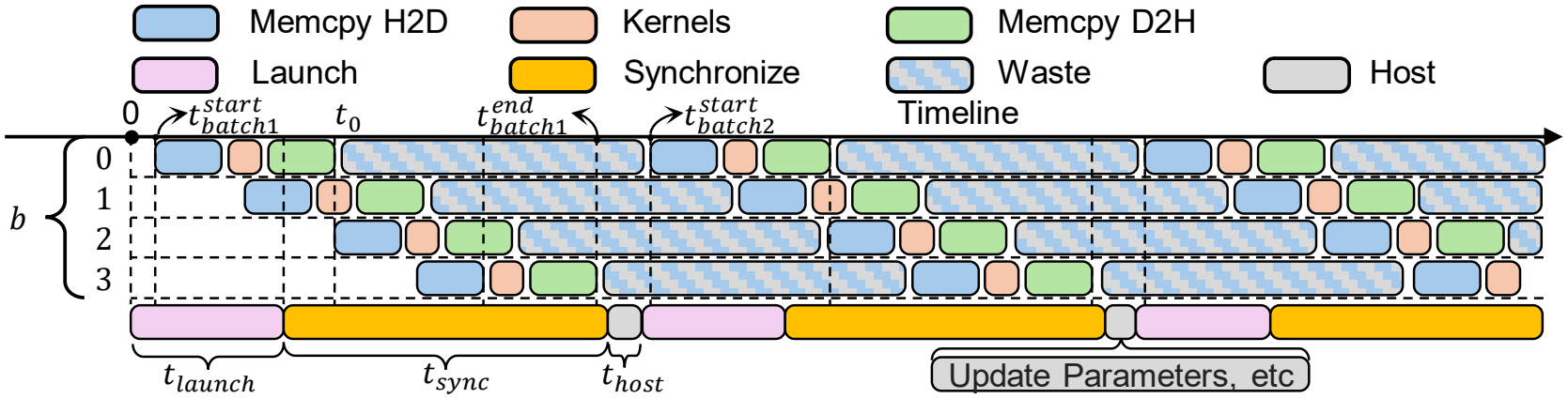
Symbol	Definition
t_{launch}	Time on CUDA graph launch
t_{sync}	Time spent on synchronization
t_{host}	Time spent on host

Bottleneck Breakdown

- **Analytical model of performance gaps with inter-batch overheads**
  - Wall-clock time for a batch:
    - $T_{measured} = t_{launch} + t_{sync} + t_{host}$

      $= T_{ideal} + t_{intra} + t_{inter}$

      $= T_{ideal} + t_{schedule}$
    - $t_{schedule}$ is the scheduling overheads, which can be quantized with profiling data



Execution flow with inter-batch overheads

Symbol	Definition
t_{launch}	Time on CUDA graph launch
t_{sync}	Time spent on synchronization
t_{host}	Time spent on host

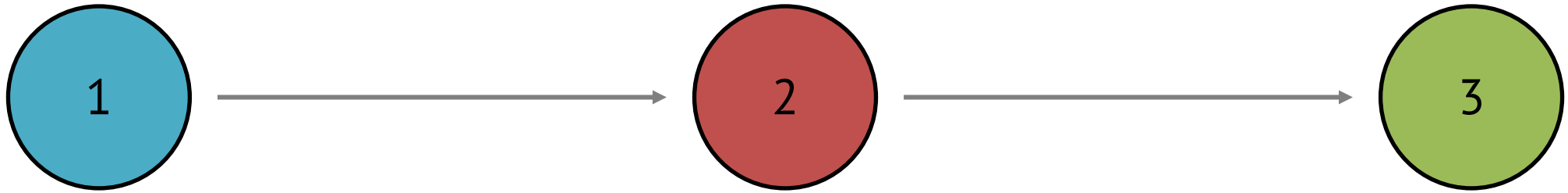
Existing Approaches

Approach	What it helps	What remains
Synchronous model	Baseline	No scheduling mechanism
CUDA Graph	Lower kernel launch overheads	Parameter updates and job-to-job scheduling still run on the host
Static Batching	More concurrent work in a batch	Batch-level synchronization creates inter-batch gaps
Queue Model	Dynamic load balancing	Shared/global queue can become a contention point
SET	Event-driven, adaptive dispatch	Per-worker queues + callbacks + work stealing

SET targets the scheduling overheads while preserving ordinary CUDA graph semantics

Our Proposal: SET

Instead of synchronizing a whole batch or polling every stream, SET reacts when a worker actually becomes free



Per-worker queues

Prepare job close to the worker that will execute it

Completion callbacks

Return a worker and notify scheduler only when its graph is done

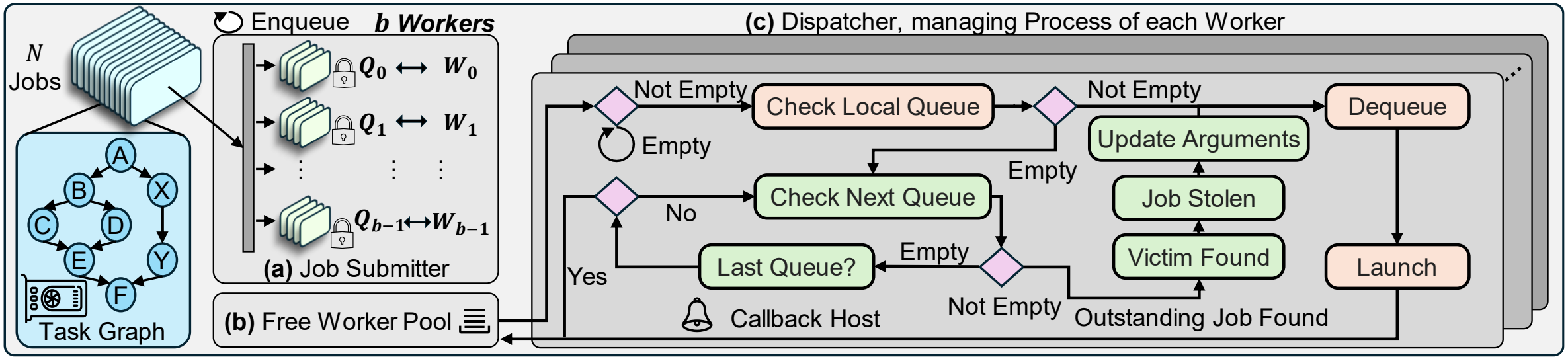
Work stealing

Move ready jobs to idle workers when needed

Goal: keep useful work in flight while making completion and resource reuse explicit

Our Proposal: SET

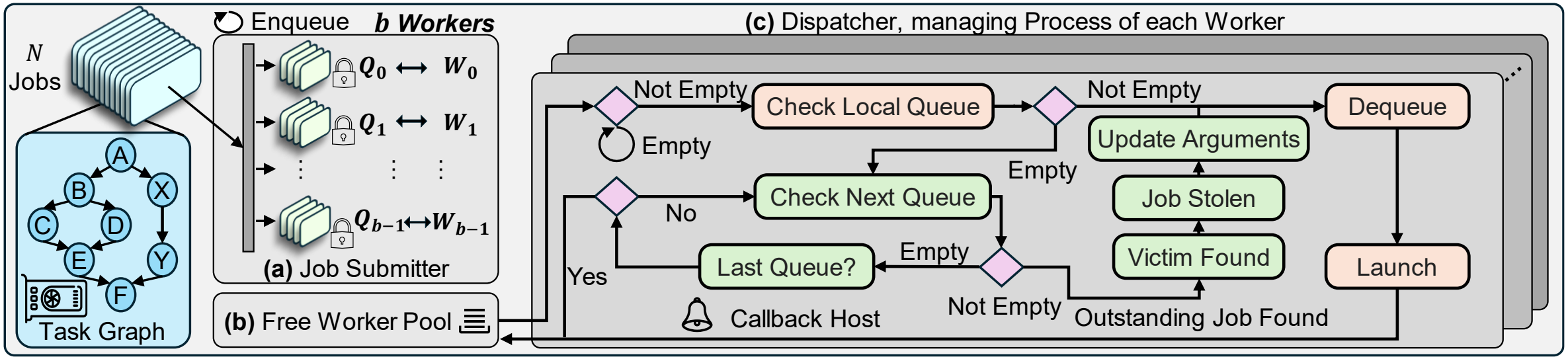
- **Overview of SET framework**
  - Key components:
    - Workers W_i: device configured with b workers, matching the batch size. Each worker W_i contains a dedicated CUDA stream S_i, a unique graph executable G_i, a set of device buffers M_i
    - Per-worker job queue Q_i: host-side queue, stores fully prepared graph executables rather than simple task indices



Overview of our runtime framework execution flow

Our Proposal: SET

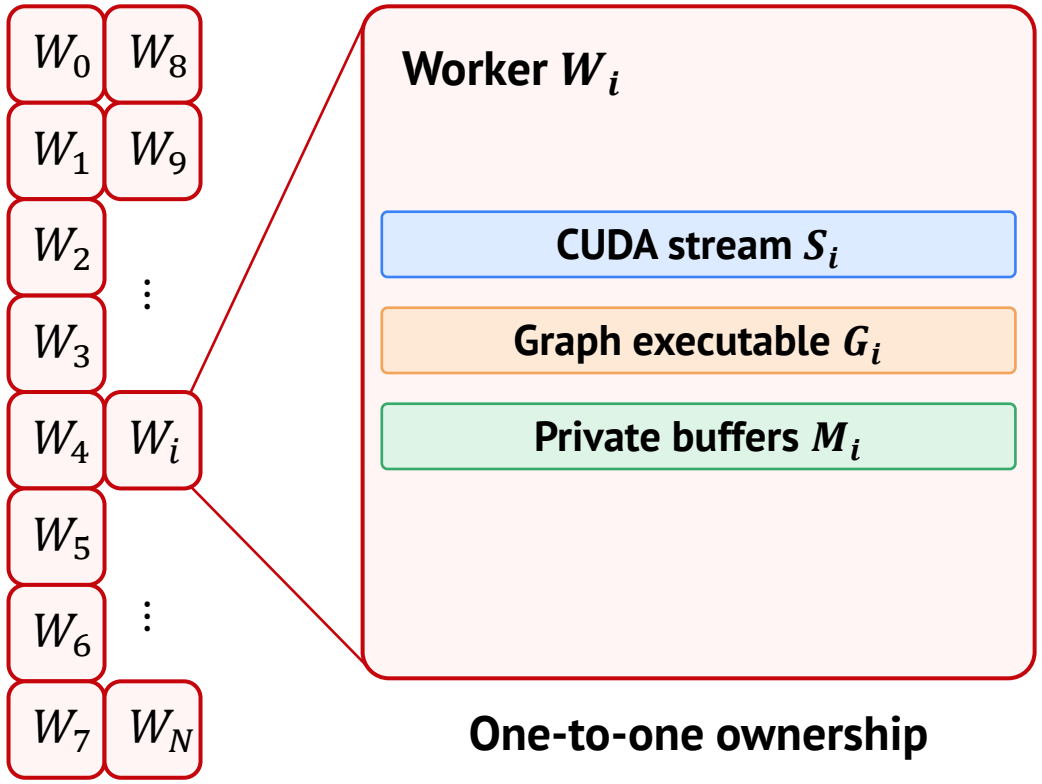
- **Overview of SET framework**
  - Key components:
    - Free worker pool W_{pool}: maintains the indices of currently available workers, updated only upon task completion
    - Submitter and dispatcher: monitors the job queues of all workers, updates the graph executables, fetches a job upon there is an available worker



Overview of our runtime framework execution flow

Our Proposal: SET

- **A worker owns everything needed to launch safely**



The graph executable G_i is bounded to the worker's private memory

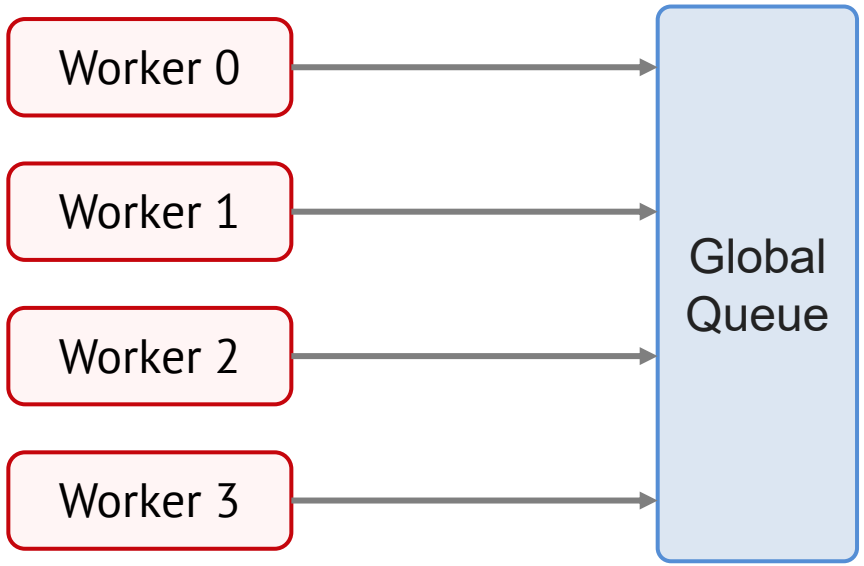
- Stream S_i handles the job
- Graph G_i is pre-instantiated for low launch overhead
- Buffers M_i isolate in-flight jobs from one another

Memory isolation is what makes work stealing safe

Our Proposal: SET

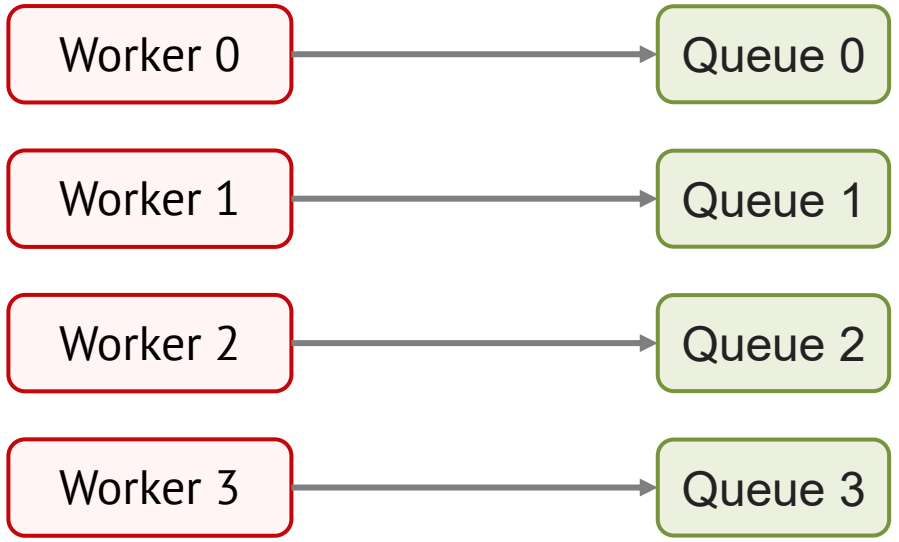
- **Per-worker queues avoid a hot global contention**

Global Queue



Many workers contend for one shared structure

SET

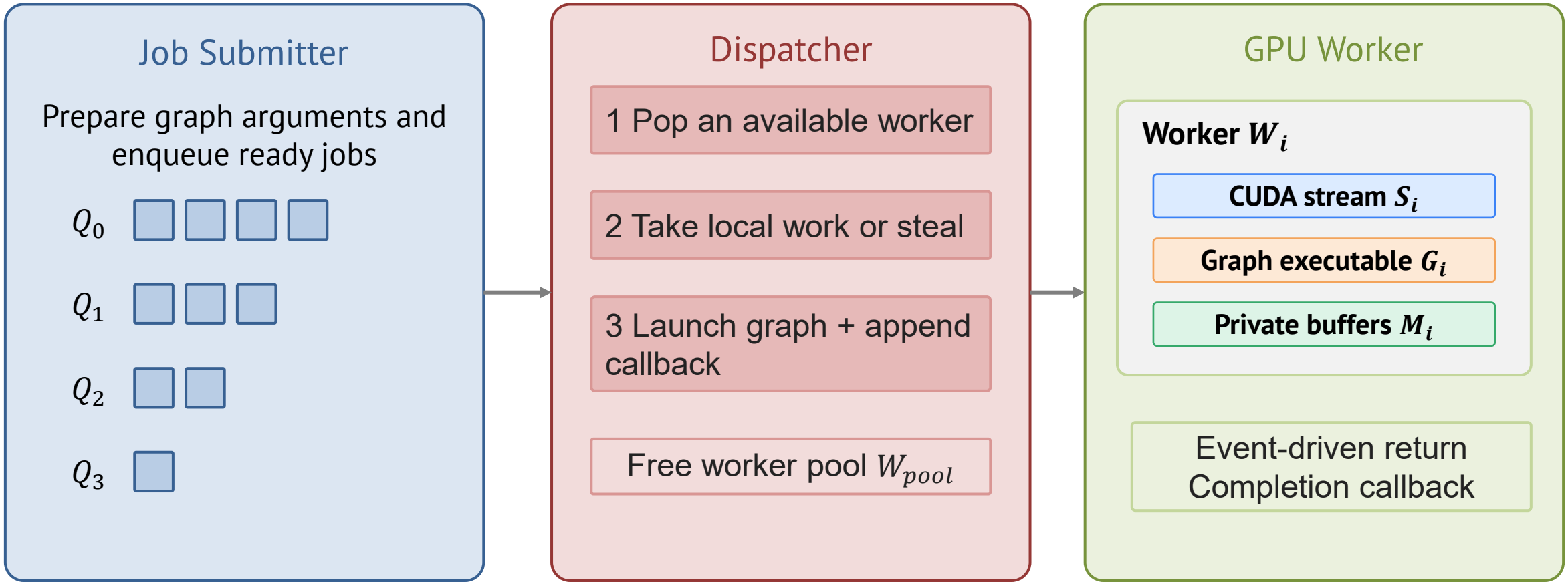


Shared state updated on completion

Less shared contention becomes especially important for tiny kernels

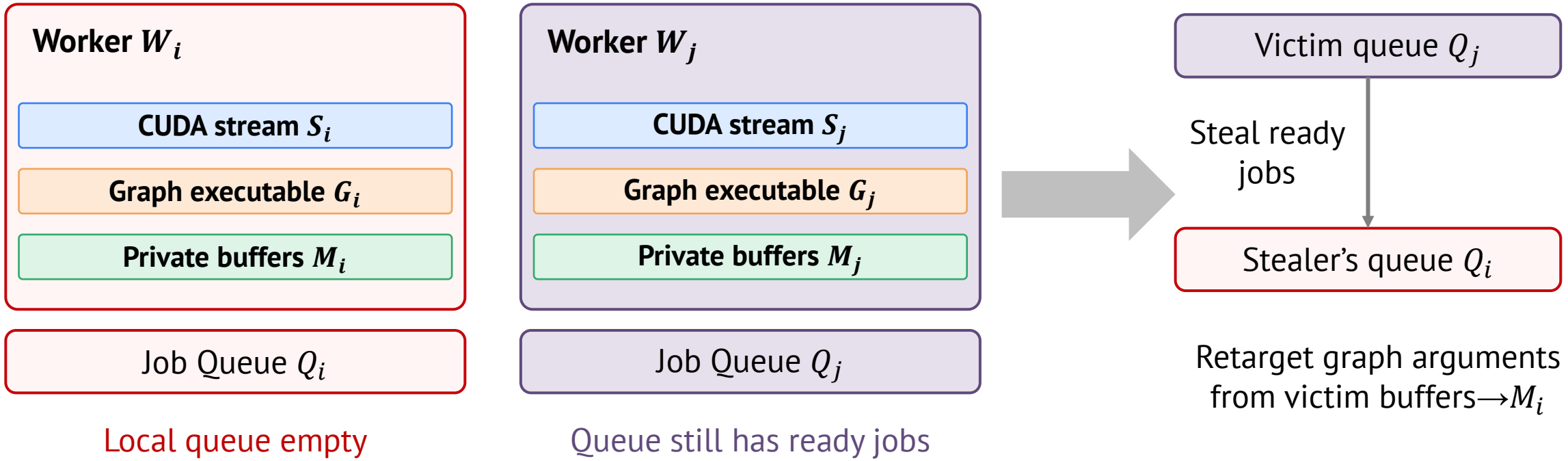
Our Proposal: SET

- **SET separates job preparation from job dispatch**



Our Proposal: SET

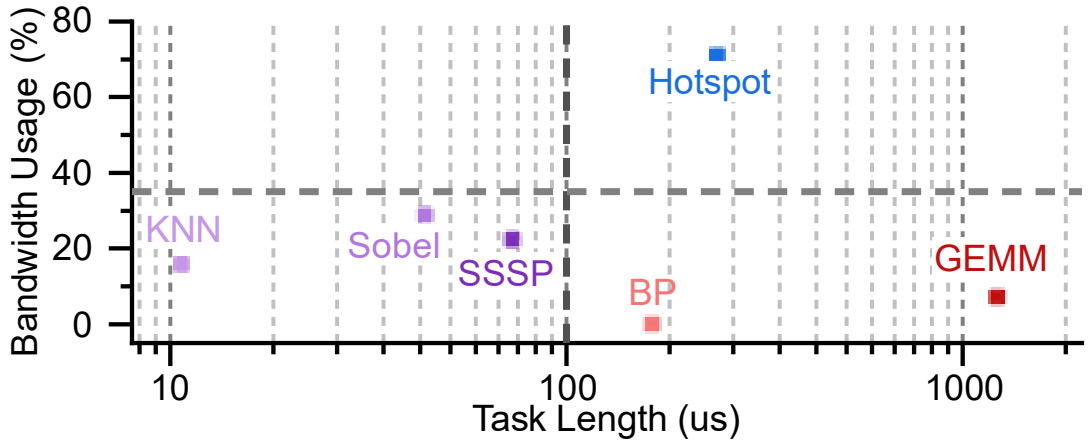
- **Work stealing keeps idle workers productive and memory-safe**



Idle worker gets useful work without allocating new device memory

Evaluations

- **Benchmark applications**
  - Sobel operator; General matrix multiplication (GEMM); Back propagation (BP); K-nearest neighbor (KNN); Hotspot; Single-source shortest path (SSSP).
- **Baselines**
  - Synchronous, Graph, Static batching, Queue
- **Platforms**
  - RTX 3090, Ampere architecture
  - RTX 5090, Blackwell architecture
- **Metrics**
  - Throughput
  - Scheduling overhead

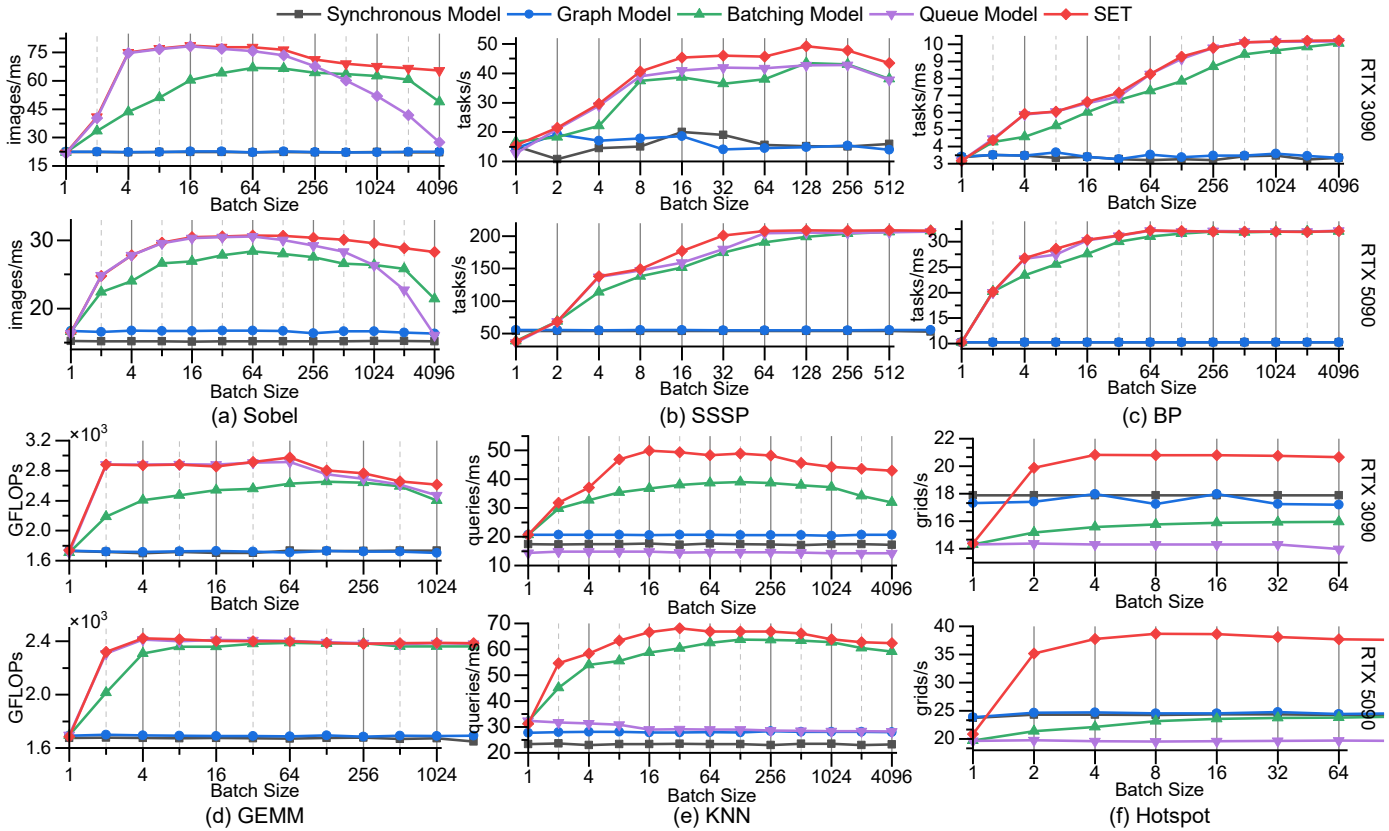


Six workloads span different task lengths and memory pressure

SET should help when the bottleneck is scheduling – not only in one kernel regime

Evaluations

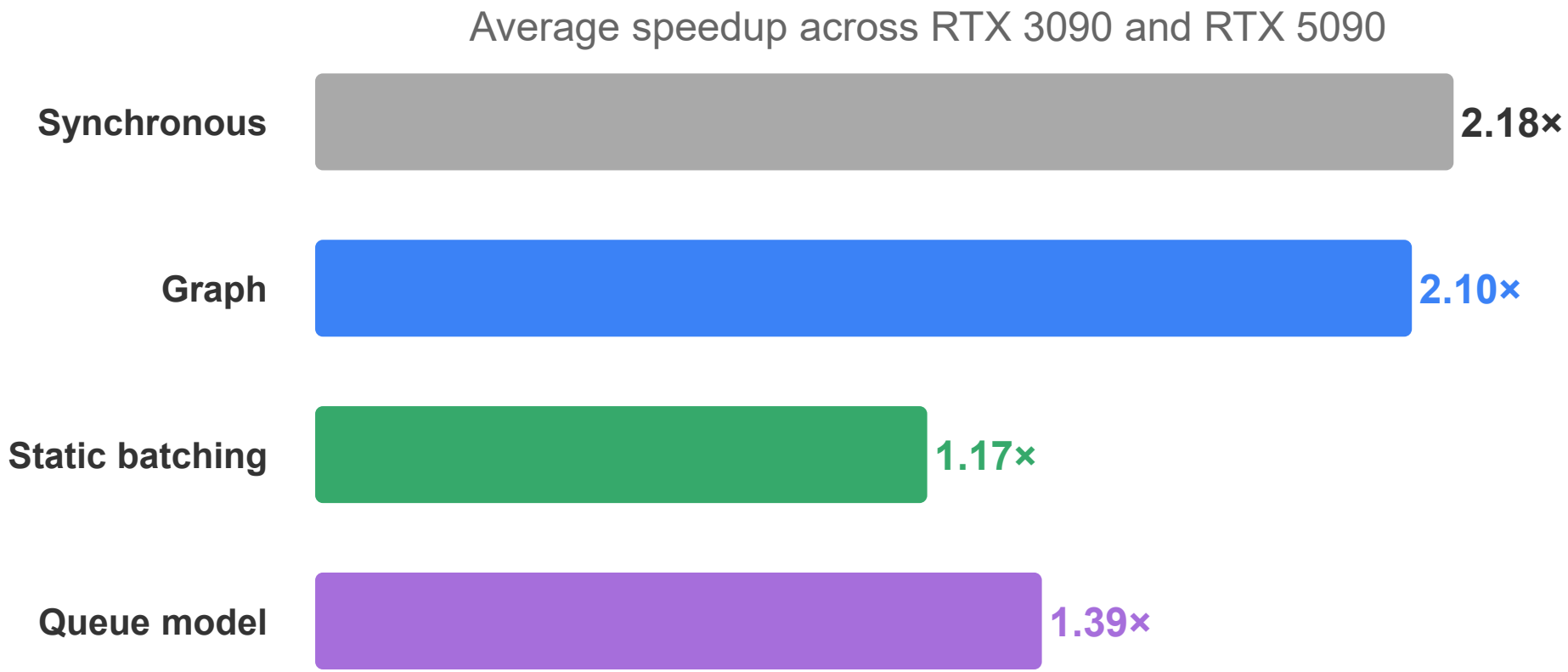
- **Performance comparison**
  - SET achieves 1.15-1.44× speedup over baselines



(a) Throughput vs. batch size in Sobel workload (img/ms) **(b)** Throughput vs. batch size in GEMM workload (GFLOPs) **(c)** Throughput vs. batch size in BP workload **(d)** Throughput vs. batch size in KNN workload (queries/ms) **(e)** Throughput vs. batch size in Hotspot workload (grids/s) **(f)** Throughput vs. batch size in SSSP workload (tasks/s)

Evaluations

- **Performance comparison**

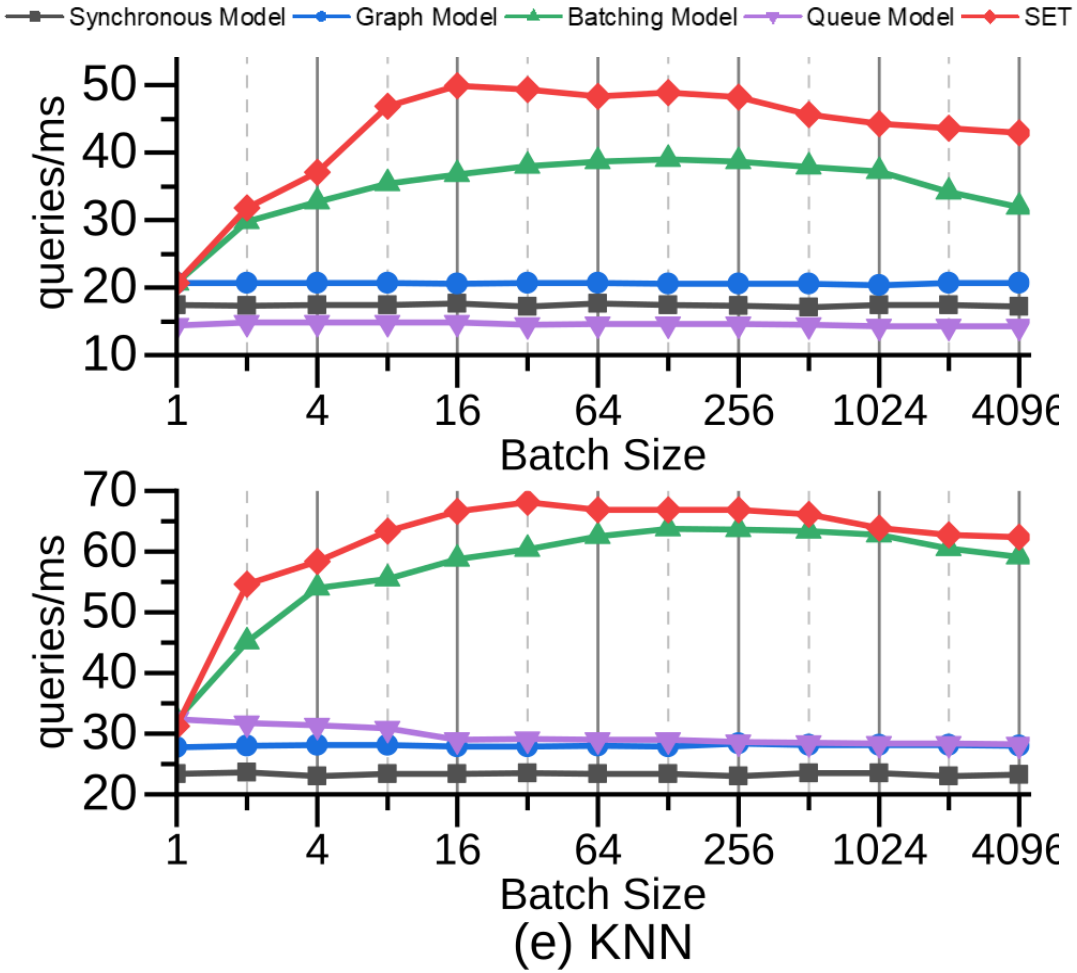


Against the strongest scheduling baselines: 1.17× vs batching and 1.39× vs queue

The exact gain depends on whether the workload is compute-limited, scheduler-limited, or bandwidth-limited.

Evaluations

- **KNN: 10µs kernels expose queue-management cost**



(e) KNN

~10 µs average KNN kernel duration

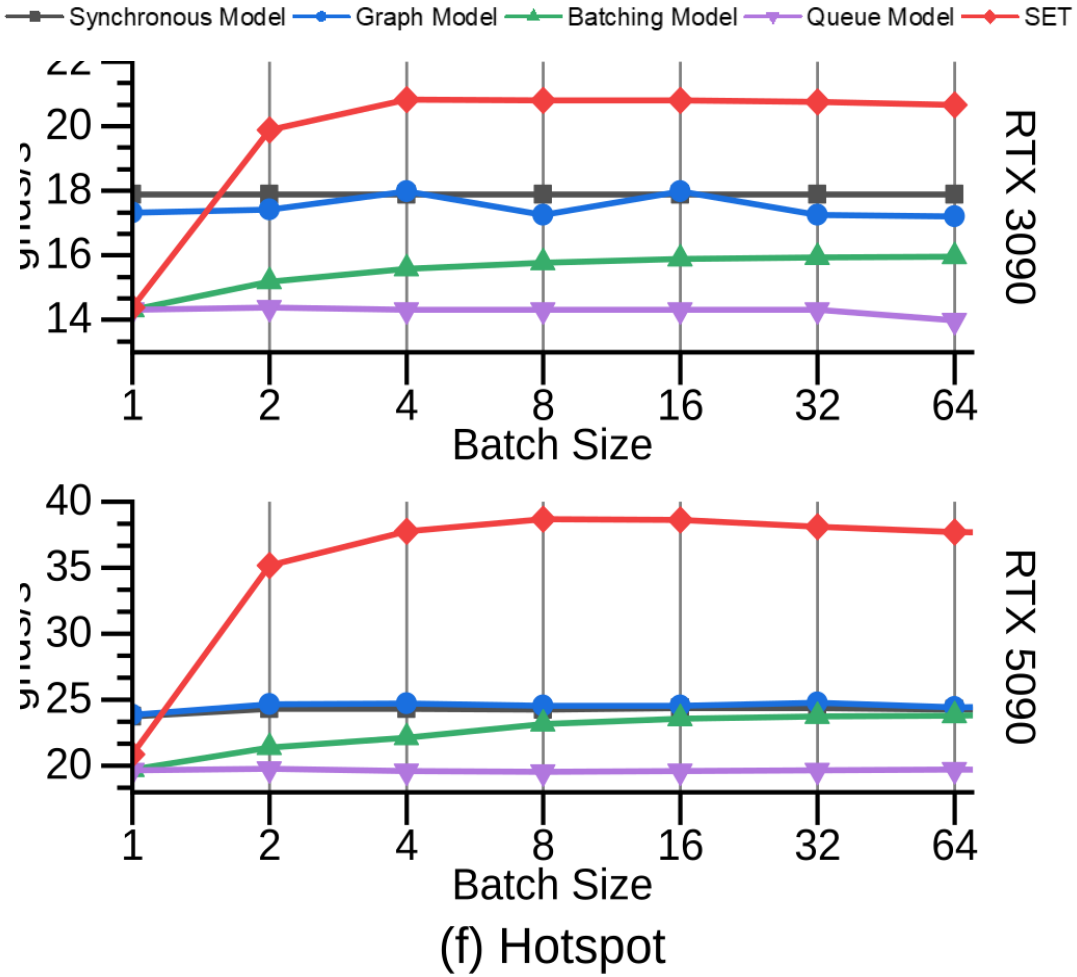
3–5 µs lock acquire/release under contention

When scheduler cost is the same order as useful work, a global mutex becomes a bottleneck

SET keeps high throughput with per-worker queues and ready-to-launch graph executables.

Evaluations

- **Hotspot: more concurrency is not always better**



(f) Hotspot

Up to ~90% DRAM bandwidth

and ~50% L2 traffic

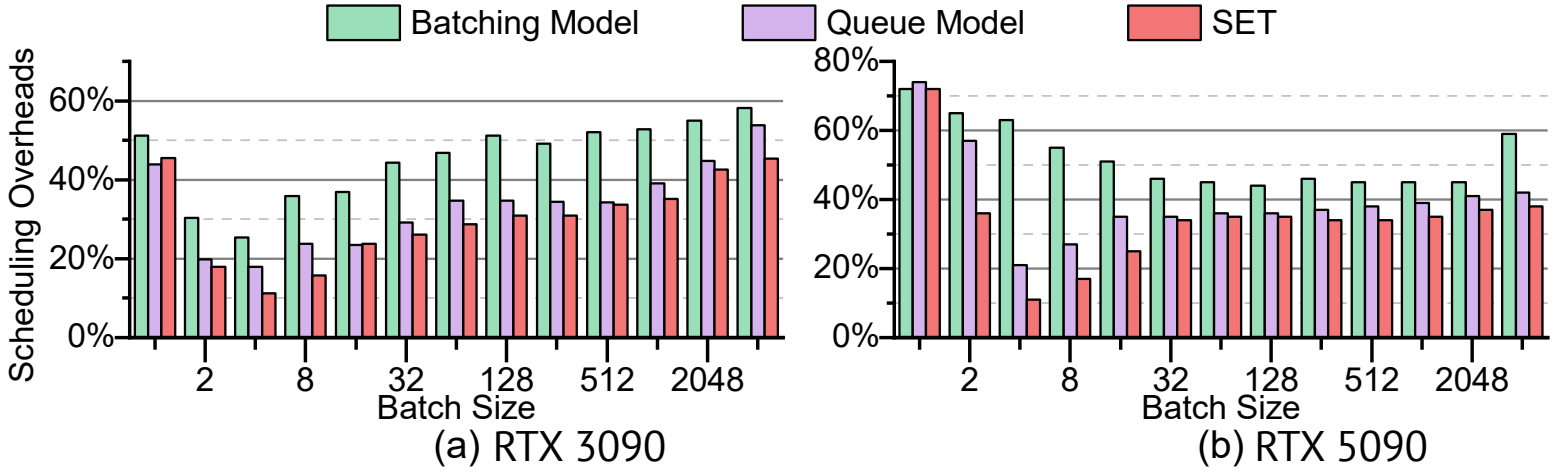
Batching / queue models can oversubscribe memory resources: more jobs simply compete for the same fixed bandwidth

SET launches on demand and still overlaps copies with kernels

Takeaway: adaptive scheduling matters even when raw parallelism is already abundant

Evaluations

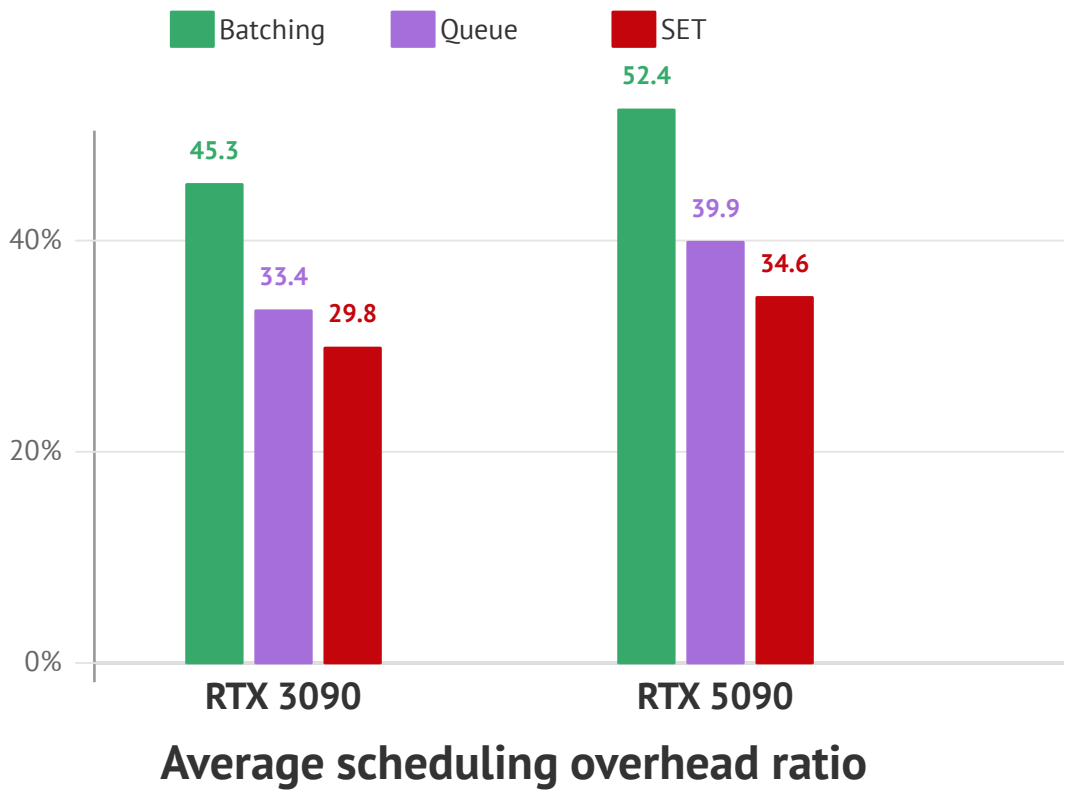
- **Scheduling overheads reduction**
  - SET achieves 1.15-1.44× speedup over baselines
  - SET reduces the scheduling overheads across different platforms, even reduces more on newer device due to Amdahl's law



Scheduling overheads with different batch sizes in different models

Evaluations

- **Scheduling overheads reduction**
  - SET reduces the scheduling overheads across different platforms, even reduces more on newer device due to Amdahl's law
  - As GPUs get faster, runtime overhead matters more



Average scheduling overhead ratio

RTX 5090 executes kernels ~1.79× faster

↓

Host-side graph updates and scheduling do not speed up at the same rate

↓

Amdahl's law: runtime overhead becomes a larger fraction of total time

SET remains the lowest-overhead scheduler on both GPUs

Takeaways

EURO-PAR CONFERENCE 2026

1. CUDA Graph largely removes launch overhead—not scheduling gaps

2. SET reacts to per-stream completion with per-worker queues, callbacks, and work stealing

3. Result: 1.15–1.44× over strong scheduling baselines and 18–54% lower scheduling overhead

As GPUs get faster, efficiently scheduling them matters even more