



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

Zhengxiong Li^(✉), Tsung-Wei Huang, and Umit Ogras

University of Wisconsin-Madison, Madison, WI 53706, USA
{zhengxiong.li, tsung-wei.huang, uogras}@wisc.edu

Abstract. Achieving peak GPU performance remains a significant challenge as the system throughput is constrained by host-device synchronization delays and kernel scheduling overheads, even with aggressive kernel optimizations and batch processing. Furthermore, existing approaches often underutilize hardware resources such as compute cores and copy engines due to scheduling overheads. To address these problems, we propose a CUDA runtime framework for task-parallel pipelines to minimize the synchronization overheads and the gap between kernel executions. The proposed solution combines two innovations: (1) a multi-stream task-parallel pipeline programming model that leverages event-chaining and work-stealing mechanisms to fully utilize available hardware resources; (2) a graph-based execution flow with per-stream buffers to ensure memory safety for multiple in-flight jobs running concurrently. Extensive evaluations on representative real-world workloads show 1.15–1.44× speedup and reduce scheduling overheads by 18–54% compared to state-of-the-art CUDA graph baselines.

Keywords: CUDA Graphs · GPU Runtime · Dynamic Task Scheduling

1 Introduction

Modern GPU applications, ranging from AI and machine learning to scientific and image processing workloads, execute as pipelines of interdependent kernels connected by data transfers and synchronization operations. To reduce the repeated cost of launching these kernels, NVIDIA introduced the CUDA Graph execution model, which captures a directed acyclic graph (DAG) of kernel launches and memory operations that can be efficiently instantiated and replayed [4]. CUDA Graph reduces the CPU-side launch overhead and improves throughput compared to traditional stream-based execution. It enables developers to achieve substantial performance improvements without optimizing memory access patterns and kernels.

Many real workloads fail to achieve full GPU utilization even with CUDA graphs [16]. Idle intervals — called kernel gaps — often appear on the GPU timeline between successive kernel executions. These gaps arise from various

reasons, such as synchronization mechanisms and CPU-side operations, including argument updates, device buffer management, stream selection, and `memcpy` enqueueing. Nsight Systems Profiler’s event timeline manifests these issues as recurring white spaces between the last instruction of one kernel and the first of the next, as shown in Fig. 1. Minimizing these kernel gaps is crucial to realizing the full performance potential of CUDA graph pipelines.

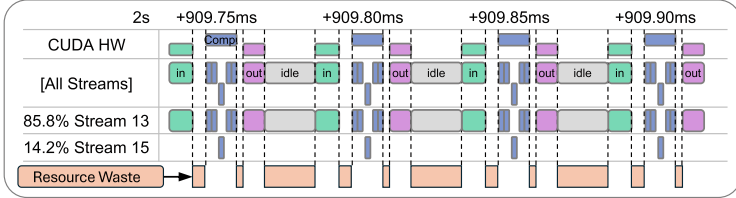


Fig. 1. Gaps between operations in Nsight profiler.

State-of-the-art approaches often submit jobs in batches [6]. They launch CUDA graphs that represent a batch of multiple parallel processing chains, while the host synchronizes after each batch completes. While this method effectively parallelizes the GPU’s computational tasks, it fails to address the underlying issue: CPU-GPU dependency. Specifically, this strategy assigns work to a fixed set of streams and relies heavily on host-side polling or blocking to safely reuse stream buffer space, avoiding overwriting outputs from prior kernels. As a result, copy engines are underutilized, and compute cores may remain idle when any stream has no work to do. Although CUDA graphs instantiation, limited batching, or stream callbacks attempt to reduce the average overhead, they fail to react to fine-grained readiness events or keep enough independent jobs in flight to absorb jitter. Hence, these approaches often fail to hide the CPU overheads or fully utilize hardware resources.

This paper proposes SET, a stream-event-triggered runtime scheduling framework for CUDA graphs that maximizes system throughput and minimizes idle gaps. SET co-schedules host and device tasks by representing each job as a reusable CUDA graph executable and coordinating multiple workers through event chaining. Rather than relying on batch-level synchronization or global polling, SET maintains multiple in-flight job queues and dynamically assigns work to streams as they become available. When a stream completes execution, event callbacks trigger resource release and enable the scheduler to promptly dispatch the next ready job. This event-driven design reduces host-side scheduling latency, mitigates inter-batch gaps, and improves hardware utilization while preserving per-job ordering and memory safety. As a result, SET sustains high throughput across workloads with diverse characteristics, including short-lived kernels and memory-bound tasks.

Our contributions in this paper are outlined as follows:

- **Bottleneck and gap characterization.** We analytically decompose scheduling overheads in CUDA graph pipelines into intra-batch and inter-batch components and explain why kernel gaps persist despite graph instantiation and multi-stream execution.
- **Event-chained host–device co-scheduling.** SET achieves adaptive task assignment with $O(1)$ synchronization overhead while preserving per-job ordering and memory safety by using bounded in-flight CUDA graph executables and callback-driven coordination to minimize inter-kernel gaps.
- **Comprehensive evaluation across diverse workloads.** We evaluate SET on six representative compute- and memory-bound workloads on NVIDIA RTX 3090 and RTX 5090, and demonstrate $1.15\text{--}1.44\times$ speedup and $18\text{--}54\%$ lower scheduling overheads than state-of-the-art host-side scheduling baselines.

In the rest, Sect. 2 discusses the prior work, Sect. 3 provides background and motivation, Sect. 4 presents the proposed SET framework. Finally, Sect. 5 presents the experimental evaluation and Sect. 6 concludes the paper.

2 Related Work

Prior work has explored three directions to minimize CUDA programming overheads: (1) *host-side scheduling* targets how the jobs are launched; (2) *domain-specific language (DSL) compilers and runtime engines* move scheduling/fusion decisions upstream; (3) *device-side dispatchers* move kernel launching decision to the device to minimize the host-device synchronization overheads.

Host-Side Scheduling enables performance optimization without modifying the kernels and source code. *The synchronous model* [4] launches kernels one by one on a single stream without any explicit scheduling optimization mechanism. *The CUDA Graph model* [4] reduces CPU overheads by pre-recording directed acyclic graphs (DAGs). However, it still pays per-replay and per-iteration parameter-update overheads for `memcpy` and kernel nodes, with limited adaptivity. *The static batching model* [6] aggregates multiple jobs into a large CUDA graph to amortize launch overheads across multiple tasks, improving throughput at the cost of increased latency and memory pressure. *The queue model* [7] dynamically balances workloads across workers. It smooths kernel gaps but incurs $O(b)$ host-side queries and can perform badly when kernels have the same order of time as the overheads of managing shared queues. Since SET is most closely related to these models, Sect. 5 presents detailed comparisons to them under six workloads. Unlike these models, SET provides adaptive per-worker queues, bounded in-flight graphs which allow on-the-fly argument updates, and $O(1)$ shared resources overheads via callback events to sustain overlap.

Device-Side Dispatching aims to minimize the host interaction using persistent kernels or threads on the device. *Whippletree* [14] and *Softshell* [13] keep *persistent kernels* on the GPU pulling tasks from on-device queues. By scheduling

work directly on the streaming multiprocessors (SMs), they reduce host launch overhead and improve fine-grained load balance. However, they cause fairness, backpressure, and termination detection challenges, which complicate interoperability with vendor libraries. *Kernelet* [17] time-slices and co-schedules small kernels to increase SM utilization for irregular workloads.

DSL Compilers and Task Runtimes offer an orthogonal direction to reduce the number of kernels that need to be launched. They also improve memory utilization since merged kernels reuse intermediate data without costly DRAM accesses. For example, the *kernel fusion* [15] is widely used in ML compilers and libraries. Similarly, algorithmic fusion, such as FlashAttention [5], reduces kernel launches and memory traffic. However, fusion may reduce modularity and be limited by dynamic control flow since it is workload- and compiler-dependent.

Heterogeneous runtime environments express task dependencies and manage data movement across CPUs and GPUs. For example, *StarPU* [1] uses “codelets” and performance models that drive dynamic scheduling and automated data coherence across memories. *Legion* [2] introduces a “data-centric” model, where tasks declare privileges over logical regions, enabling mappers and runtime to infer dependencies, place data, and generate asynchronous copies. Recent systems combine task runtimes with CUDA Graph to cut host dispatch overhead [9]. The proposed SET framework is orthogonal to these approaches and can be integrated seamlessly with them.

In summary, existing techniques either optimize host-side submission, rely on coarse-grained batching, or move scheduling entirely to the device. None provides fine-grained, event-driven coordination between host and device while preserving CUDA Graph semantics. Most approaches operate either entirely on the host (e.g., CUDA Graph [4], static batching [6], queue models [7]) or device (e.g., persistent kernels [14]). SET fills this gap through an event-chained host-device scheduling that preserves compatibility with CUDA Graph while minimizing inter-kernel gaps and maintaining runtime flexibility.

3 Background and Motivation

3.1 CUDA Graph

CPU submits worker threads to GPU through a sequence of individual kernel launches and memory copy calls. Each call incurs a non-trivial amount of overhead, leading to significant bottlenecks in applications with many small operations. CUDA Graph is introduced to mitigate this issue by allowing a series of operations to be defined, optimized, and launched as a single unit of work [4]. Developers can consolidate multiple kernels into a single graph launch operation, thereby reducing the overall kernel launch overheads.

3.2 Bottleneck Analysis

We abstract general CUDA programs into three key steps: `memcpy H2D` (host-to-device), kernels, and `memcpy D2H` (device-to-host). We denote the batch size as b ,

time spent on `memcpy` H2D as t_{in} , time spent on `memcpy` D2H as t_{out} , time spent on kernels as t_k . As illustrated in Fig. 2(a), the *ideal execution time* without any gaps between operations can be written as:

$$T_{ideal} = bt_{in} + t_k + t_{out} \quad (1)$$

During actual execution, there are two types of additional idle periods, intra-batch and inter-batch overheads as described next.

Intra-batch Overheads: First, the host-to-device copies (`memcpy` H2D) execute with a delay between them, as denoted by t_{in-in} in Fig. 2(b). Second, the kernels execute with a delay (t_{in-k}) after the input is copied. Third, there is delay between finishing the kernel execution and writing the output from the device to host (`memcpy` D2H), which is denoted as t_{k-out} in Fig. 2(b). Finally, suppose the total kernel execution time increases by Δt_k due the *inter-kernel* gaps. Then, the *total intra-batch overheads* can be written as the difference between the overall execution time and T_{ideal} as:

$$t_{intra} = (b-1)t_{in-in} + t_{in-k} + \Delta t_k + t_{k-out} \quad (2)$$

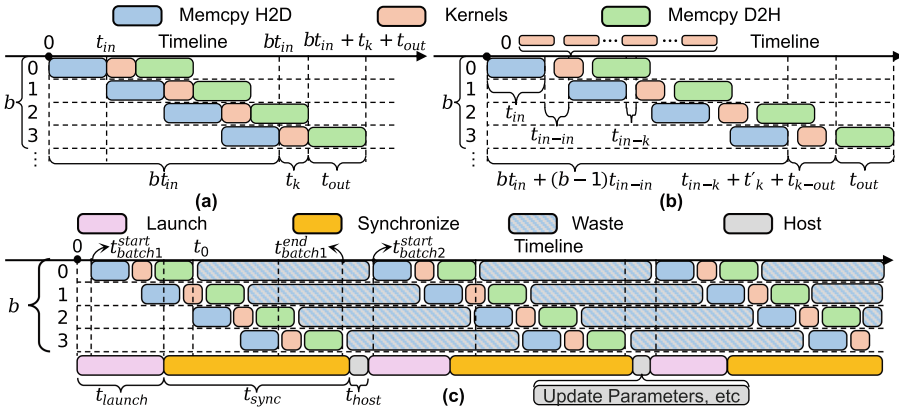


Fig. 2. (a) Ideal execution flow of a static batching CUDA program (b) Execution flow with intra-batch gaps (c) Execution flow with inter-batch gaps.

Inter-batch Overheads: At the inter-batch level, the gap persists between batches, as shown in Fig. 2(c). At time t_0 , the first job in the batch has already finished. However, the job in the next batch cannot start until t_{batch2}^{start} due to synchronization and parameter updates. The preceding batch ends at t_{batch1}^{end} . Therefore, the *inter-batch overheads* can be written as:

$$t_{inter} = t_{batch2}^{start} - t_{batch1}^{end} \quad (3)$$

placeholders to facilitate runtime updates. For tasks implemented without third-party libraries (e.g., CUTLASS and cuDNN), the dependencies are defined explicitly. Otherwise, the graph is built with a combination of explicit descriptions and `streamcapture` to avoid unnecessary placeholders [16].

To ensure data integrity, write operations to active memory slots are prohibited. When there are multiple workers, SET keeps a batch size of b memory copies because the tasks in the batch run concurrently. The pre-allocated memory slots are reusable across different batches since allocating new memory is costly. The stolen jobs are retargeted to the thief’s buffers to guarantee memory safety without new device-side allocations.

4.2 Event-Chained Scheduling

Scheduler Components: SET relies on four key components: (1) **Workers** are the fundamental units of execution. The device is configured with b workers, matching the batch size. Each worker $W_i, \forall i \in Z, 0 \leq i < b$ encapsulates a dedicated CUDA stream S_i , a unique, pre-instantiated graph executable G_i and a dedicated set of device-side buffers M_i . Crucially, G_i is strictly bounded to operate only on the memory in M_i . This isolation facilitates memory-safe work-stealing.

(2) **Per-worker job queue.** Each worker W_i is assigned a thread-safe, host-side job queue Q_i . To mitigate the overhead of on-the-fly argument updates, Q_i stores fully prepared graph executables rather than simple task indices.

(3) **Free worker pool.** The thread-safe queue W_{pool} maintains the indices of currently available workers. Rather than employing polling or round-robin checks, the pool is updated only upon task completion, thereby reducing host-device communication.

(4) **Submitter and Dispatcher.** These are two distinct host threads. The submitter serves as a *producer*. It continuously monitors the job queues of all workers, prepares the graph executable by updating its parameters and determines when to add the new job and which queue to add it to. The dispatcher serves as a *consumer*. It monitors whether there is an available worker. If there is, it fetches a job to feed it. They work closely together to ensure that all workers are fed quickly without oversubscribing host-device synchronization.

Initialization: The main thread (1) populates all per-worker job queues $\{Q_0, \dots, Q_b\}$, and (2) initializes all workers $\{W_0, \dots, W_b\}$ with their corresponding streams, buffers, and graph executables. Then, it spawns the job submitter and dispatcher threads.

Job Submission The job submitter monitors all queues for available slots. Upon finding a free slot, it proceeds to acquire the corresponding queue, updates the parameters for job J and enqueues it, as detailed in Algorithm 1 (lines 3–5) and Fig. 3(a). The jobs within a selected queue $Q_i, \forall i \in Z, 0 \leq i < b$ are

Algorithm 1: Job Submitter

Data: Next job counter
 c_{next} , job queue Q_i ,
 job J

Result: Populated job
 queue $Q_i \forall i \in$
 $Z, 0 \leq i < b$

```

1 id  $\leftarrow c_{\text{next}}$ ;
2 if  $Q_i$  not full then
3    $c_{\text{next}} \leftarrow$ 
     compare_exchange( $c_{\text{next}}$ ,
       id, id+1);
4   UpdateGraphParams( $J_{\text{id}}$ ,
      $i$ );
5    $Q_i$ .push( $J_{\text{id}}$ );
```

Algorithm 3: Asynchronous Resource Return (Callback)

Data: Done job counter
 c_{done} , completed
 worker w_{id}

Result: Freed worker
 returned to pools

```

1  $c_{\text{done}}$ .atomic_fetch_add(1);
2  $W_{\text{pool}}$ .push( $w_{\text{id}}$ );
3 notify_one();
```

Algorithm 2: Job Dispatch & Launch

Data: Free worker pool W_{pool} , job queues Q_i

Result: Dispatched GPU jobs

// Dispatcher:

```

1 if  $W_{\text{pool}}$  not empty then
2    $w_{\text{id}} \leftarrow W_{\text{pool}}$ .pop();
3    $job \leftarrow \text{FindJob}(w_{\text{id}}, Q_i)$ ;
4   if  $job \neq \text{NULL}$  then
5     LaunchJob( $job, w_{\text{id}}$ );
6   else
7      $Q_W$ .push( $w_{\text{id}}$ );
8 Function FindJob( $w_{\text{id}}, Q_i$ )
9   if  $Q_{w_{\text{id}}}$ .try_pop( $job$ ) = true then
10     $job.is\_stolen \leftarrow \text{false}$ ;
11    return  $job$ ;
12   for  $k \leftarrow 1$  to  $b - 1$  do
13     $victim\_id \leftarrow (w_{\text{id}} + k) \bmod b$ ;
14    if  $Q_{victim\_id}$ .try_steal( $job$ ) = true
15      then
16         $job.is\_stolen \leftarrow \text{true}$ ;
17        return  $job$ ;
18   return NULL;
18 Function LaunchJob( $job, w_{\text{id}}$ )
19   if  $job.is\_stolen = \text{true}$  then
20     UpdateGraphParams( $job, w_{\text{id}}$ );
21   GraphLaunch( $job, w_{\text{id}}$ );
22   AddCallback( $job, w_{\text{id}}, \text{Callback}$ );
23   return;
```

expected to be executed by worker W_i . This one-to-one correspondence allows their parameters to be updated for the worker before insertion.

Job Dispatch and Launch. In parallel with *job submission*, the dispatcher thread monitors the free worker pool W_{pool} , as shown in Fig. 3(b). Upon retrieving an idle worker W_{free} , the dispatcher invokes the **FindJob** function to search for a ready job, as in Algorithm 2 (lines 2–3). While searching, the dispatcher first attempts to acquire a job from the selected worker’s (W_{free}) dedicated queue (Q_{free}). If the local queue is not empty, the head of Q_{free} is popped and launched. Otherwise, the dispatcher invokes a work-stealing policy by iterating through peer queues Q_k , where $k \neq \text{free}$ and attempting to steal the first job it meets, as shown in Algorithm 2 (lines 9–17) and Fig. 3(c).

The next step is launching the job (**LaunchJob** on line 5 in Algorithm 2). If the job was found in the free worker’s local queue, then the dispatcher directly launches the job on worker W_{free} . Otherwise (i.e., it was stolen from another queue), the graph executable is natively bound to the victim’s memory resources. To address this mismatch, a just-in-time (JIT) update is performed to retarget

the graph parameters. This update rebinds the executable to (1) read from W_{free} ’s input memory, (2) use W_{free} ’s buffers and (3) write to W_{free} ’s output memory. Following the reconfiguration, the dispatcher launches the “stolen” job on w_{free} , as shown in Fig. 3(c) and Algorithm 2 (lines 19–23).

Asynchronous Resource Return. After each launch, whether that job was stolen or not, the dispatcher appends an asynchronous callback event to worker W_{free} at the end of the stream. When W_{free} completes all operations in the graph, the callback event is triggered by the CUDA driver on a separate thread. This event atomically increments the counter for finished jobs and returns W_{free} to the W_{pool} . This mechanism automatically recycles the worker, making it available for subsequent iterations of the dispatch loop, as detailed in Algorithm 3.

5 Experimental Evaluations

5.1 Evaluation Setup

Hardware Configuration. All evaluations are conducted on two Linux servers, one equipped with Intel Xeon Gold 6330, NVIDIA RTX 3090 and 192 GB RAM, the other equipped with Intel I7-11700, NVIDIA RTX 5090 and 128 GB RAM. The servers run on Ubuntu 22.04, with NVIDIA CUDA v12.8. All source code is compiled with O2 optimization and C++20 standard.

Benchmark Applications. We employ six workloads that span compute- and memory-bound behavior, as shown in Fig. 4.

1. **Sobel operator** [12] applies normalization, edge detection, mean filtering, binary thresholding, and blending operation to input images.
2. **General matrix multiplication (GEMM)** performs tile-based dense matrix multiplication.
3. **Back propagation (BP)** [11] workload performs a single-layer training step with on-device synthetic minibatch generation.
4. **K-nearest neighbor (KNN)** [8] workload performs supervised classification via brute-force search for feature vectors.
5. **Hotspot** [10] runs iterative thermal simulations solving differential equations.
6. **Single-source shortest path (SSSP)** [3] workload performs Bellman–Ford graph traversal with frontier-based relaxation for improved GPU parallelism.

We employ throughput as the primary performance metric and analyze the scheduling overheads using NVIDIA Nsight Systems Profiler.

Baseline Models. We compare the proposed SET framework against four representative programming models, which are implemented with best-practice optimizations to ensure fairness.

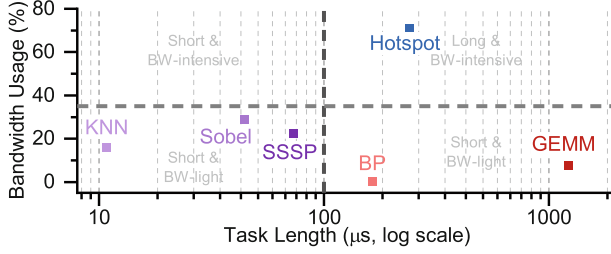


Fig. 4. Average memory usage and task lengths of benchmarks used in this work.

1. **Synchronous model** [4] launches kernels one by one on the same stream without any explicit scheduling mechanism.
2. **Graph model** [4] pre-instantiates the job as a graph, then launches it repeatedly. We reduced data movement overheads when copying input values to graph’s placeholders [16] using explicit API calls and stream capturing.
3. **Static batching model** [6]. While maintaining the batching model’s scheduler, we adopt dynamic CUDA graph construction to enable on-the-fly parameter updates.
4. **Queue model** [7]. We keep the queue model’s scheduler “issue queue” unchanged, and change its job wrapper from merged kernels to CUDA graphs to make sure it holds the same input as other baseline models.

5.2 Performance (Throughput) Evaluation

Figure 5 plots the throughput as a function of the batch size b across all workloads and programming models. GEMM, Hotspot, and SSSP have a lower maximum batch size since they exceed the memory capacity with larger batch sizes.

General Trends: Throughput increases with the batch size since the GPU can run more tasks in parallel and maximize hardware utilization. For all workloads, SET, queue model, and batching models achieve higher throughput as the batch size grows until reaching maximum hardware utilization. As the batch size continues to grow, the scheduling overheads become dominant, as explained in Sect. 5.3. In contrast, the synchronous and graph models do not benefit from batching and appear as flat lines since they only have one worker stream. Therefore, their performance is limited by the host latency to emit jobs. They cannot utilize all hardware resources. Consequently, they have significantly lower throughput than other approaches except for Hotspot workload, as detailed under the workload-specific analysis.

Sobel, SSSP, BP, GEMM Analysis (Fig. 5(a)-(d)): SET outperforms the synchronous and graph models with a significant margin, achieving $2.31\times$ and $2.27\times$ speedup on average, respectively, as summarized in Table 1. Unlike the

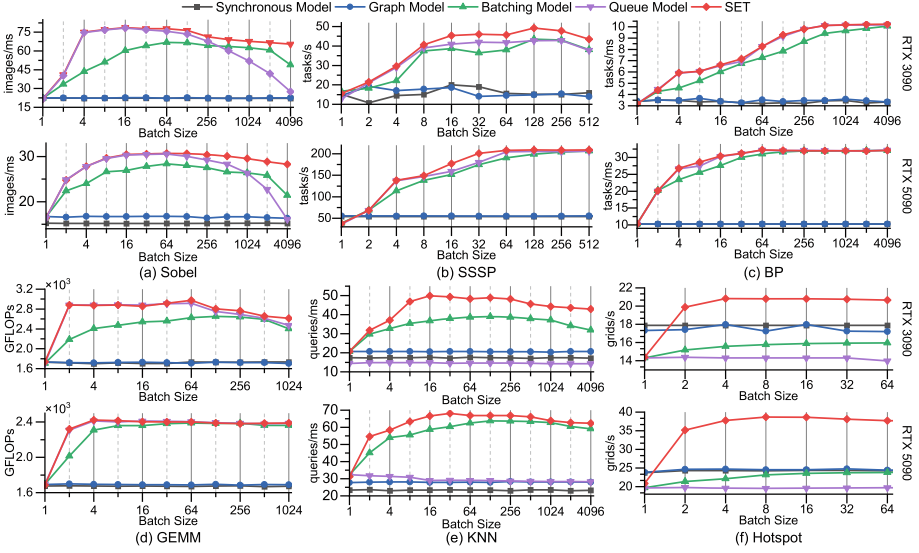


Fig. 5. Throughput vs. Batch size in (a) Sobel (img/ms), (b) GEMM (GFLOPs), (c) BP (tasks/s), (d) KNN (queries/ms), (e) Hotspot (grids/s), and (f) SSSP (tasks/s) workloads.

synchronous and graph models, the batching model benefits from larger batch sizes and achieves a higher throughput. However, the inter-batch level overheads become the bottleneck, thus degrading its throughput. As a result, SET achieves $1.10\times$ speedup over the batching model on average. The queue model has the closest performance to the proposed SET framework under these workloads. It achieves the same performance for batch sizes up to 32 in Sobel, 4 in SSSP, 4096 in BP, and 32 in GEMM. Beyond the thresholds, SET outperforms the queue model due to lower scheduling overheads when kernels saturate the hardware, as detailed in Sect. 5.3. The queue model oversubscribes the bandwidth, exacerbating its performance degradation in Sobel, where kernels involve over 30% L2 cache traffic. On average, SET still achieves $1.06\times$ speedup over the queue model.

The KNN Workload Fig. 5(e): We analyze the KNN workload separately since it exhibits a different behavior, where the queue model performs poorly even worse than the synchronous model. As shown in Fig. 4, the KNN workload involves many kernels whose execution time is as small as an average of $10\ \mu\text{s}$, the same order to acquire and release thread lock (usually around $3\ \mu\text{s}$ to $5\ \mu\text{s}$ when facing data contention). In the queue model, many small kernels invoke the host to acquire/release a mutex to its global queue frequently, while workers are waiting for the scheduler to dispatch them a job. Therefore, its performance is even worse than the synchronous and graph models and cannot improve with larger batch sizes. In contrast, the proposed SET framework still holds a high throughput in KNN workload thanks to its per-worker job queue and ready-to-go

job wrapper, eliminating the frequent mutex acquire/release and the as-needed updates to graph nodes’ arguments, which become important when dealing with small kernels.

The Hotspot Workload (Fig. 5(f)): Another extreme workload is Hotspot, which is a heavily memory-bound workload as shown in Fig. 4. Profiling data from Nsight Compute shows that Hotspot uses up to 90% of DRAM bandwidth and up to 50% of L2 cache traffic. So, the GPU throughput can easily saturate when the DRAM reaches its maximum utilization, as more kernels just split the same fixed bandwidth across more requests. The queue and batching models suffer from high data contention since they launch more jobs than the device can hold efficiently. In this case, the synchronous model and graph model even perform better than them, since they only process one job at a time, which puts less pressure on TLB, L2 cache and DRAM and has higher spatial locality and temporal locality. In contrast, SET still benefits from launching the jobs on demand and overlapping device copies with kernels. As shown in Fig. 5(f), SET is the only technique that benefits from larger batch sizes and outperforms all baselines by a significant margin.

Throughput Comparison Summary: SET delivers the best throughput compared to other models, achieving an average (average on RTX 3090 and 5090) of $2.18\times$, $2.1\times$, $1.17\times$, $1.39\times$ speedup against synchronous, graph, batching and queue models, respectively (see Table 1). These results, aggregated across RTX 3090 and 5090 GPUs, demonstrate that SET maintains high performance even under demanding workloads characterized by small kernels and high memory pressure (Fig. 4, Fig. 5). Such improvements underscore the robustness of our approach across diverse execution environments.

Table 1. Speedups over baseline models on RTX 3090 (Ampere architecture) and RTX 5090 (Blackwell architecture)

Speedup	Synchronous		Graph		Batching		Queue	
	Ampere	Blackwell	Ampere	Blackwell	Ampere	Blackwell	Ampere	Blackwell
Sobel	$2.99\times$	$1.86\times$	$2.97\times$	$1.71\times$	$1.23\times$	$1.12\times$	$1.20\times$	$1.10\times$
SSSP	$2.45\times$	$3.07\times$	$2.46\times$	$2.99\times$	$1.15\times$	$1.07\times$	$1.10\times$	$1.03\times$
BP	$2.34\times$	$2.78\times$	$2.26\times$	$2.78\times$	$1.10\times$	$1.04\times$	$1.01\times$	$1.01\times$
GEMM	$1.58\times$	$1.39\times$	$1.58\times$	$1.38\times$	$1.12\times$	$1.02\times$	$1.01\times$	$1.01\times$
KNN	$2.47\times$	$2.63\times$	$2.08\times$	$2.19\times$	$1.23\times$	$1.08\times$	$2.94\times$	$2.09\times$
Hotspot	$1.10\times$	$1.47\times$	$1.39\times$	$1.45\times$	$1.27\times$	$1.56\times$	$1.38\times$	$1.81\times$
Average	$2.15\times$	$2.20\times$	$2.12\times$	$2.08\times$	$1.18\times$	$1.15\times$	$1.44\times$	$1.34\times$

5.3 Scheduling Overhead Analysis

Figure 6 plots the scheduling overheads over the total execution time for the three best performing programming models: batching, queue model and SET. As shown in Eq. (4), the fraction is calculated by $\text{Fraction} = \frac{t_{\text{schedule}}}{T_{\text{measured}}}$.

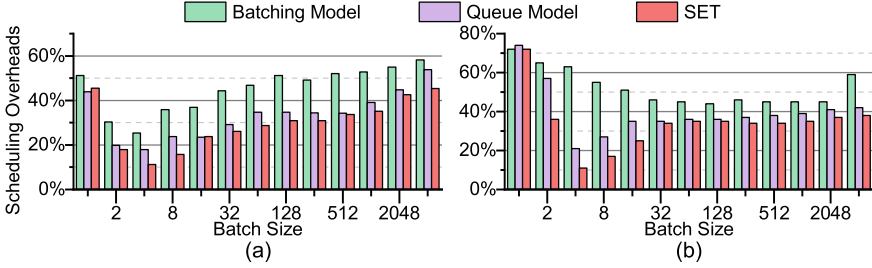


Fig. 6. Scheduling overheads with different batch sizes in different models. (a) on RTX 3090 (b) on RTX 5090.

All workloads have low hardware utilization when the batch size equals one since the launch overheads are high relative to the kernel execution time. Specifically, scheduling overhead accounts for 45-51% of the total execution time on RTX 3090, resulting in low throughput. This behavior is an artifact of inter-kernel gaps and consistent with the analysis in Sect. 3 and the execution flow shown in Fig. 2(b). Throughput increases with the batch size, as more kernels can collectively saturate the GPU, thereby reducing the relative contribution of scheduling overhead. Indeed, the scheduling overhead of the SET, queue, and batch models drops to 11%, 18%, and 25% on RTX 3090, respectively, when the batch size is four.

As the batch size grows, the inter-batch overheads become significant. This effect is most pronounced in the batching model, as it synchronizes across batches at the end of each batch. As a result, its scheduling overheads increase from 36% with a batch size of 8 to 59% when the batch size is 4096. The queue model and our method incur host-side overheads (e.g., job queues and free worker pool monitoring). These costs remain lower than the batching model and are negligible when workers do not cause high data contention. However, the queue model suffers more from managing shared resources than SET, which incurs substantial contention on its global mutex, resulting in up to 54% overhead at a batch size of 4096. In contrast, SET alleviates this bottleneck by using per-worker queues and minimizing shared-queue operations, resulting in lower overhead at high batch sizes. Overall, SET achieves consistently lower scheduling overheads (15% with a batch size of 8 and 45% with a batch size of 4096). On average, SET has 54.64% and 18.62% lower scheduling overhead than the batch and queue models.

While the previous analysis focused on the Ampere architecture (RTX 3090), these overheads persist, and are often exacerbated on Blackwell architecture

Table 2. Comparison of Scheduling Overheads on RTX 3090 and RTX 5090

Average Ratio	Batching	Queue	SET
RTX 3090 (Ampere)	45.32%	33.36%	29.83%
RTX 5090 (Blackwell)	52.38%	39.85%	34.62%

(RTX 5090). Our profiling indicates that moving to Blackwell yields a $1.79\times$ average speedup in kernel execution time (T_{compute}), driven by increased CUDA core counts, higher clock frequencies, and improved IPC (instruction-per-cycle). However, host-side scheduling (T_{overhead}) for operations such as graph instantiation and parameter updates scales more slowly than raw compute throughput. In accordance with Amdahl’s Law, as T_{compute} diminishes, T_{overhead} exerts a greater influence on total execution time. As shown in Table 2, the relative overhead ratio is higher on the RTX 5090, confirming that modern GPU architectures are increasingly bottlenecked by scheduling inefficiencies.

In summary, the scheduling overheads are large with small batches due to low hardware utilization. They reduce with higher GPU utilization and reach a minimum value. However, they increase again due to inter-batch overheads, resulting in a U-shaped behavior. This behavior explains the inverted U-shaped throughput curves observed in Sobel, GEMM, and SSSP in Fig. 5.

6 Conclusions

This paper presents SET, a stream-event-triggered scheduling framework for efficient CUDA graph pipelines. The analytical model clearly shows that the scheduling overheads consist of two parts, intra-batch overheads and inter-batch overheads. To minimize the overheads, SET maintains a job submitter to feed the job queues and a dispatcher to fetch a job for any available worker. SET implements an event-driven callback signal to enable asynchronous resource release upon a job completion. We evaluate its performance on six real-world workloads, where SET achieves $1.15\text{--}1.44\times$ speedup over the state-of-the-art baselines and reduces the scheduling overheads by 18–54%. SET provides an orthogonal, kernel-agnostic design and is friendly to other optimizations to kernels or compilers.

Disclosure of Interests. Dr. Ogras serves as a contractor for Samsung Austin Research & Development Center and Advanced Computing Lab (SARC/ACL). This relationship has been approved under applicable outside activities policies.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: STARPU: a unified platform for task scheduling on heterogeneous multicore architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03869-3_80

2. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: 2012 International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE (2012). <https://doi.org/10.1109/sc.2012.71>
3. Bellman, R.: On a routing problem. *Q. Appl. Math.* **16**(1), 87–90 (1958). <https://doi.org/10.1090/qam/102435>
4. Corp., N.: Cuda c++ programming guide (2025). <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Accessed: 3 Nov 2025
5. Dao, T., et al.: Flashattention: fast and memory-efficient exact attention with IO-awareness. In: Oh, A.H., Agarwal, A., Belgrave, D., Cho, K. (eds.) *Advances in Neural Information Processing Systems* (2022). <https://openreview.net/forum?id=H4DqfPSibmx>
6. Ekelund, J., Markidis, S., Peng, I.: Boosting performance of iterative applications on gpus: Kernel batching with CUDA graphs. In: 2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 70–77. IEEE (2025). <https://doi.org/10.1109/pdp66500.2025.00019>
7. Guevara, M., et al.: Enabling task parallelism in the CUDA scheduler (2009). <https://api.semanticscholar.org/CorpusID:306206>
8. Guo, G., Wang, H., Bell, D., Bi, Y., Greer, K.: KNN model-based approach in classification. In: Meersman, R., Tari, Z., Schmidt, D.C. (eds.) *OTM 2003. LNCS*, vol. 2888, pp. 986–996. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39964-3_62
9. Huang, T.W., Lin, D.L., Lin, C.X., Lin, Y.: Taskflow: a lightweight parallel and heterogeneous task graph computing system. *IEEE Trans. Parallel Distrib. Syst.* **33**(6), 1303–1320 (2022). <https://doi.org/10.1109/tpds.2021.3104255>
10. Huang, W., et al.: Hotspot: a compact thermal modeling methodology for early-stage VLSI design. *IEEE Trans. Very Large Scale Integration (VLSI) Syst.* **14**(5), 501–513 (2006). <https://doi.org/10.1109/tvlsi.2006.876103>
11. Mitchell, T.M.: *Machine learning*. McGraw-Hill international editions, McGrawHill, New York [u.a.], [nachdr.] edn. (2013)
12. Sobel, I., Feldman, G.: An isotropic 3x3 image gradient operator (2015). <https://doi.org/10.13140/RG.2.1.1912.4965>
13. Steinberger, M., Kainz, B., Kerbl, B., Hauswiesner, S., Kenzel, M., Schmalstieg, D.: Softshell: dynamic scheduling on GPUS. *ACM Trans. Graphics* **31**(6), 1–11 (2012). <https://doi.org/10.1145/2366145.2366180>
14. Steinberger, M., et al.: Whippletree: task-based scheduling of dynamic workloads on the GPU. *ACM Trans. Graphics* **33**(6), 1–11 (2014). <https://doi.org/10.1145/2661229.2661250>
15. Wang, G., Lin, Y., Yi, W.: Kernel fusion: an effective method for better power efficiency on multithreaded GPU. In: 2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing, pp. 344–350. IEEE (2010). <https://doi.org/10.1109/greencom-cpscom.2010.102>
16. Zheng, B., et al.: Grape: Practical and efficient graphed execution for dynamic deep neural networks on GPUS. In: 56th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 1364–1380. MICRO '2023, ACM (2023). <https://doi.org/10.1145/3613424.3614248>
17. Zhong, J., He, B.: Kernelet: high-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Trans. Parallel Distrib. Syst.* **25**(6), 1522–1532 (2014). <https://doi.org/10.1109/tpds.2013.257>