CUDA Proxy Player: Combining CUDA Graphs and Persistent Kernels for Launch-Bound Inference

Haoran Zhang
zhhaoran@umich.edu

Xiayang Jin
ausummer@umich.edu

Yuchen You
yuchenxr@umich.edu

Abstract

Modern conditional-computation workloads, such as Mixture-of-Experts (MoE) models and recommendation systems, fragment GPU execution into thousands of fine-grained, dynamically-routed micro-operations. This fragmentation creates a launch-bound regime where host-side kernel dispatch overhead significantly impacts end-to-end latency: our end-to-end test shows that on realistic MoE forward passes, eliminating launch overhead reduces latency by up to 24%.

We present CUDA Proxy Player (CPP), a runtime system that addresses this bottleneck through a dual-path dispatch architecture. CPP decomposes workloads into *graph-safe segments*, which are mapped to pre-captured, bucketed CUDA Graphs for ultra-low-latency replay and *glue segments*, which are handled by GPU-resident persistent workers. Our micro-architectural analysis reveals that runtime graph capture incurs up to 16× higher latency than execution due to implicit driver allocations, validating our offline pre-capture design with shape bucketing to target allocation-friendly dimensions.

Evaluation on MoE inference workloads demonstrates that CPP achieves up to 1.56× latency reduction and 1.25× throughput improvement in the launch-bound regime, with benefits diminishing as workloads transition to compute saturation, matching our design intent. However, our analysis also reveals fundamental limitations: persistent workers introduce synchronization overhead that offsets their launch-avoidance benefits, and idle workers degrade graph segment performance by up to 13%. These findings provide actionable guidance for GPU runtime design and identify coordination overhead as the key barrier to effective hybrid execution.

1 Introduction

The evolution of hardware accelerators has widened the gap between GPU throughput and host-side control. Modern GPUs (for example NVIDIA H100) provide far higher arithmetic intensity, yet each CUDA kernel launch still incurs on the order of a few microseconds of driver and runtime overhead. Prior work reports launch costs in the 5–15 μs range on contemporary GPUs [4], and on our A100 platform we measure about 2.8 μs on average (3.1 μs at the 95th percentile). For dense Large Language Models (LLMs) this cost is usually hidden by large matrix multiplications, but conditional computation in Mixture of Experts (MoE) and large-scale recommendation models breaks monolithic computations into many fine-grained micro-operations. These workloads enter a launch-bound regime in which high-performance GPUs spend more time waiting for CPU commands than doing useful work, an effect we call the "launch wall" [3, 6].

Conditional workloads combine a relatively static *backbone* of dense matrix multiplications with highly dynamic *glue* for data dependent routing and expert selection. Existing systems usually optimize only one side. Monolithic graph engines such as TensorRT-LLM [10] and XLA [1] assume mostly static control flow and under dynamic routing either fall back to eager execution with the same launch overheads or pay millisecond-scale costs to re-capture graphs. Kernel fusion techniques, employed by frameworks like Triton [11] and DeepSpeed-Inference [2], reduce launch counts by merging adjacent operators into single kernels. While effective for regular operator chains, fusion requires substantial engineering effort for each new operator pattern and becomes brittle under shape variation or policy changes. This has shown to be a poor fit for the heterogeneous, dynamically-routed workloads we target. Purely eager execution handles dynamism naturally but underutilizes GPUs in the launch-bound regime. Treating these piecewise static workloads as entirely static or entirely dynamic forces an unfavorable trade off.

We observe that the backbone and glue expose a structural dichotomy that a runtime can exploit. Graph safe segments with static shapes can be pre-captured and replayed with near zero dispatch cost, while the remaining glue logic is too dynamic for CUDA Graphs but can in principle be offloaded to GPU resident workers that bypass the host launch path. Realizing this idea raises three challenges. (1) Runtime graph capture is prohibitively expensive for small kernels, with instantiation overhead up to 16× the execution time in our profiling. (2) Shape variation across requests can break graph reuse and bloat the cache. (3) CUDA Graphs and persistent kernels interact poorly since capture is sensitive to concurrent activity while workers occupy streaming multiprocessors (SMs) indefinitely and can starve graphs.

CUDA Proxy Player (CPP) is a runtime system that addresses these challenges by orchestrating execution directly on the GPU. CPP implements a dual path dispatch mechanism that identifies graph safe execution streams with static shapes and fixed memory footprints, maps them to pre-captured CUDA Graphs organized by shape buckets, and sends the remaining routing and dependency tracking logic to a device side worker queue consumed by persistent proxy workers. To ensure deadlock free coexistence, CPP completes graph pre-capture before launching workers, isolates the two paths using stream priorities, and maintains a baseline safe invariant in which any optimization failure triggers transparent fallback to eager execution.

We evaluate CPP on MoE inference workloads across multiple batch sizes and sequence lengths. In the launch bound regime, piecewise CUDA Graph replay reduces latency by up to 1.56× and improves throughput by up to 1.25× over an eager baseline, matching hand tuned fusion while preserving maintainability. On a realistic MoE forward pass, CPP cuts end-to-end latency from 40.3 ms to 30.8 ms with pre capture overhead of 90 ms that amortizes within ten iterations. At the same time, our analysis shows that persistent workers currently introduce coordination overhead that cancels most of their launch avoidance benefits, identifying synchronization cost as the main barrier to fully hybrid execution.

In summary, this paper makes four contributions. First, we characterize kernel launch overheads and CUDA Graph instantiation costs on NVIDIA RTX 6000 and A100 GPUs, identifying a launch bound regime ($N \leq 512$) where host overhead accounts for roughly 76% of end-to-end latency and showing that runtime graph capture can cost up to 16× the kernel execution time. Second, we present CUDA Proxy Player, a dual path runtime that maps compute heavy sub graphs to pre captured shape bucketed CUDA Graphs and uses persistent GPU workers to stitch dynamic glue on device. Third, we describe an implementation that achieves deadlock free coexistence between graphs and persistent kernels through memory pre allocation and stream coordination. Finally, we provide a comprehensive evaluation that demonstrates up to 1.56× speedup in launch bound workloads and quantitatively analyzes the limitations of persistent workers, including SM contention, barrier overhead, and queue latency, offering guidance for future GPU runtime designs.

We implement CUDA Proxy Player in C++/CUDA (about 2.6K lines of code) and open-source the implementation and evaluation scripts at `https://github.com/eecs582/fall2025-group-11`.

2 Related Work

CUDA Proxy Player relates to work on graph based execution, kernel level persistence, and LLM serving systems. We focus on intra GPU orchestration for launch bound inference, providing a baseline safe runtime that routes work between piecewise CUDA Graphs and persistent workers without requiring model or hardware changes.

Monolithic graph execution and fusion engines. CUDA Graphs [9] and graph engines such as TensorRT LLM [10] and XLA [1] reduce launch overhead by capturing and replaying static execution graphs, with operator fusion. These systems work well when control flow is predictable, but conditional workloads such as MoE introduce data dependent routing that forces repeated re capture or eager fallback. Our microbenchmarks (§5.1.3) show that runtime re capture can be up to 16× slower than executing the captured graph at small sizes, making just in time capture a negative optimization. CPP instead decomposes models into graphable and glue segments and applies graphs only to stable regions, using shape bucketing to handle variation without repeated capture.

Kernel level persistence and compiler optimization. Frameworks such as Triton [11], CUTLASS, and DeepSpeed Inference [2] generate highly optimized kernels, including fused and persistent thread variants that improve intra kernel efficiency. For fragmented MoE workloads, however, the bottleneck shifts to inter kernel orchestration, since the host must still launch many small, dynamically routed operators. CPP lifts persistence from a kernel implementation detail to a runtime abstraction in which GPU resident workers pull micro operations from a device queue. Our evaluation (§5.4) shows that this approach faces coordination limits: worker scheduling latency (4.3 µs) exceeds eager launch cost (2.8 µs), and synchronization barriers add about 1.25 ms per forward pass, providing empirical bounds on when persistent worker architectures are effective.

LLM serving and centralized scheduling. LLM serving systems such as Orca [12], Paella [8], and vLLM [7] improve batching and memory management (for example via PagedAttention) but keep the CPU to GPU control path unchanged, so kernel launches remain bounded by PCIe and driver latency. CPP operates at a complementary layer, optimizing the dispatch path under each request while remaining compatible with existing schedulers.

Domain specific MoE and RecSys optimizations. Systems such as DeepRecSys [6], FasterMoE [5], and MegaBlocks [3] reduce fragmentation using specialized layers and block sparse kernels tailored to particular architectures, often requiring model changes or custom operators.

CPP is complementary and acts at the runtime layer, orchestrating existing operators via its proxy API without domain specific fused kernels. For workloads where such specialized kernels are unavailable, our results show that piecewise CUDA Graphs can provide speedups comparable to manual fusion (§5.2) while better accommodating shape variation.

3 Design

3.1 Overview

CUDA Proxy Player provides a unified GPU execution runtime for kernel-launch-bound inference workloads. Rather than forcing developers to manually segment models or choose between eager execution, CUDA Graphs, and persistent kernels, the system exposes a single *Request* abstraction and automatically maps each request to an optimized execution plan.

The runtime supports two execution modes: a *baseline eager mode* that launches each operator as a conventional CUDA kernel, and an *optimized proxy mode* that combines CUDA Graph replay for shape-stable regions with persistent-kernel execution for micro-op-dense regions. As shown in Figure 1, the runtime routes requests through a *Router*, which dispatches work to either a *Graph Manager* (for graphable segments) or a *Device Queue* consumed by *Persistent Workers* (for glue segments).

The design follows three guiding principles: (1) *Separate graph-friendly regions from micro-op-heavy paths* so that each execution strategy can be optimized independently; (2) *Minimize steady-state CPU involvement* by preferring graph replay and device-side execution over repeated host launches; (3) *Exploit shape stability aggressively* while using dynamic paths as fallback, balancing efficiency with adaptability.

A key invariant is *baseline safety*: all optimizations are opportunistic overlays on eager execution. If graph capture fails, if the device queue is full, or if workers are unavailable, the affected operations transparently fall back to baseline launches without affecting correctness.

3.2 Router and Execution Modes

The router is the entry point of the runtime. It implements a lightweight, configuration-driven policy that maps each request to either baseline or proxy mode based on inexpensive request-level signals: batch size, tensor shape stability, and micro-op density.

Requests meeting configured thresholds for size and stability enter proxy mode, where they are partitioned into *graph-safe segments* (stable operator sequences suitable for CUDA Graph capture) and *glue segments* (fine-grained operations unsuitable for capture). User custom operations can be explicitly labeled as graph-safe, glue, or will implicitly fall back to the baseline approach (eager launch).

Path selection is decoupled from execution logic through a configuration abstraction, allowing routing strategies to evolve independently via offline profiling or online adaptation.

3.3 Piecewise CUDA Graphs

CUDA Graphs reduce kernel-launch overhead but require stability in operator topology and tensor shapes. For workloads with control-flow branches or dynamic routing, capturing the entire inference pass as a monolithic graph is infeasible. CUDA Proxy Player addresses this through *piecewise graphing*: decomposing computation into multiple graph-safe segments interleaved with glue segments.

Shape Bucketing. To enable graph reuse across requests with slightly different shapes, the runtime discretizes dominant shape parameters (batch size, sequence length) into a fixed set of buckets. Segment instances mapping to the same bucket and sharing an operator template are treated as *graph-equivalent* and reuse the same captured graph.

Graph Manager. As illustrated in Figure 2, the Graph Manager organizes execution around *graph classes* defined by segment identifier, bucketed shape, and operator template. The first request in each class triggers a full capture-and-instantiate sequence; subsequent requests reuse the cached executable. Small shape variations within a bucket trigger lightweight updates rather than full recapture.

3.4 Persistent Kernel Path

Many workloads contain fine-grained operations—packing, scattering, routing logic, short elementwise ops—that are not graph-safe and suffer from launch overhead when run eagerly. The persistent-kernel path executes such micro-ops entirely on the GPU, serving as glue between CUDA Graph segments.

Device-Resident Task Queue. The path is built around a ring buffer in device memory acting as a single-producer (host) / multiple-consumer (GPU workers) queue. The host enqueues compact task descriptors; persistent workers dequeue and execute them concurrently. This design achieves sub-microsecond host-side submission overhead.

Long-Lived GPU Workers. The runtime launches persistent workers—long-lived GPU kernels that continuously poll the device queue. Each worker decodes the task type and executes the corresponding micro-op inline. Because workers remain active for the runtime's lifetime, micro-op segments execute without per-operation kernel launches, converting dozens of tiny launches into a continuous GPU-executed task stream.

Design Rationale. The persistent path serves dual roles: a fast path for standalone micro-op-heavy requests and the glue layer between CUDA Graph segments. By supporting GPU-resident scheduling, it eliminates the CPU–GPU synchronization points that dominate eager execution.

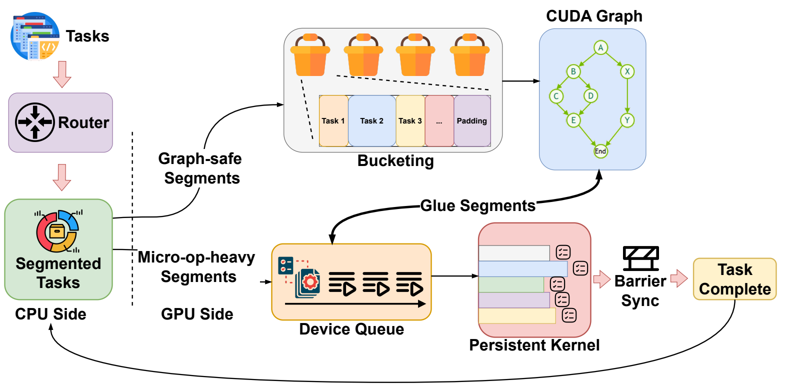

Figure 1: Overall architecture of the CUDA Proxy Player runtime. The system routes each MoE inference request to the most suitable execution path - baseline, CUDA Graph, or persistent kernel - based on request characteristics.

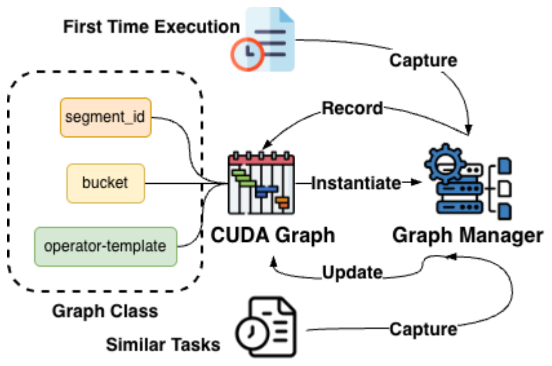

Figure 2: Graph Manager workflow. Requests are grouped into graph classes defined by *(segment id, bucket, operator template)*. The first request triggers capture; subsequent requests reuse the cached executable.

4 Implementation

We implemented CUDA Proxy Player as a standalone C++17/CUDA runtime (approximately 2.6K lines of code) that applications link against in place of the native CUDA API. The prototype builds against CUDA 12.1 and cuBLAS on NVIDIA A100 GPUs, without dependencies on cuDNN, CUTLASS, or other heavyweight libraries. Individual subsystems, such as the router, graph manager, and persistent engine, can be selectively enabled via configuration, facilitating controlled experiments.

4.1 Graph Manager and Bucketed Cache

The Graph Manager maintains a *bucketed graph cache* indexed by a key combining segment identifier, bucketed dimensions, and a compact operator-sequence descriptor. Within each bucket, different concrete shapes share the same graph and memory pool.

On cache lookup, the manager checks a bounded hash map. On a hit, it launches the cached executable on an execution stream and reuses bucket-local scratch buffers. On a miss, it will just fall back to the baseline mode for safety reasons.

To bound memory growth, we cap cached graphs per segment class and apply LRU eviction. During initialization, the runtime pre-captures a configured set of hot segment classes *before* launching persistent workers, ensuring common shapes are available without steady-state capture overhead.

4.2 Host-Mapped Device Queue

The worker path uses a host-mapped ring buffer in pinned memory, allocated once via `cudaHostAllocMapped`. This allows the host to submit micro-op tasks without explicit `cudaMemcpy` calls. The host is the sole producer; persistent kernels are consumers.

To minimize overhead, the host enqueues tasks in batches and publishes work to the device via periodic memory fences rather than per-task writes. Queue entries are fixed-size task records encoding a task type and parameters, avoiding dynamic allocation or pointer chasing on the GPU. If the queue is full, the producer executes remaining work on the baseline path rather than blocking, preserving the baseline-safe invariant while keeping the queue simple and bounded.

4.3 Persistent Worker Kernels

Persistent workers are long-running CUDA kernels that poll the device queue and dispatch micro-op tasks. To avoid starving other GPU work, we restrict their footprint: workers launch on low-priority streams with a small number of blocks, consuming a bounded fraction of SM capacity.

When the queue is empty, workers yield SM time using exponential backoff via `__nanosleep`, reducing contention with throughput-oriented kernels (e.g., cuBLAS GEMMs) while maintaining low wake-up latency for bursty traffic. A lightweight heartbeat structure in device-visible memory allows the host to detect hangs and request clean shutdown; if the worker subsystem becomes unavailable, the proxy disables it and falls back to baseline.

4.4 Request Pipeline

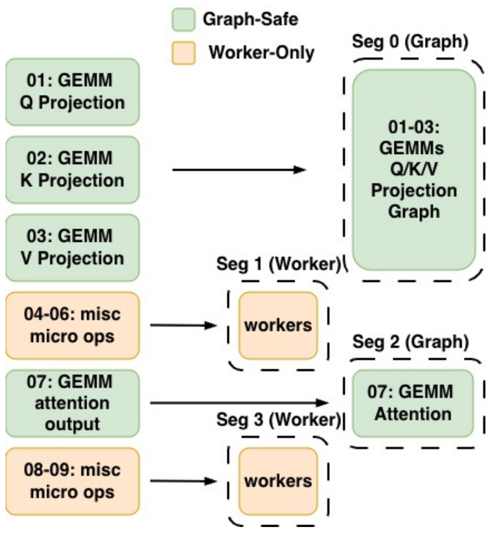

Figure 3: Example segmentation of an attention block. Operators are classified as graph-safe (green) or worker-only (orange). Consecutive operators of the same class form segments: ops 1–3 and op 7 become graph segments; ops 4–6 and 8–9 become worker segments executed by persistent workers.

At startup, the proxy loads configuration specifying the bucket grid, routing thresholds, and enabled paths. If graphs are enabled, it pre-captures hot segment classes on dedicated streams. If the worker path is enabled, it allocates the device queue and launches persistent workers. The proxy begins serving requests only after initialization completes.

Each request follows a simple pipeline: the proxy extracts features (batch size, shape hashes, micro-op counts) and invokes the router. For proxy-mode requests, the flattened operator list is scanned once, classifying operators as graph-safe or worker-only; consecutive operators of the same class are grouped into segments (Figure 3). Graphable segments are dispatched to the Graph Manager; glue segments are encoded as task descriptors and enqueued to the device queue. Baseline-mode requests launch all kernels eagerly on application-owned streams.

Throughout this pipeline, fallbacks are local to segments: if the Graph Manager cannot reuse or capture a graph, or if the device queue is full, only the affected segment falls back to baseline. This preserves correctness while allowing optimizations to be evaluated in isolation by toggling configuration flags.

4.5 Ordering and Deadlock Avoidance

CUDA Proxy Player preserves the execution semantics of the baseline eager path. Graph segments are captured and replayed on internal streams but are always synchronized back to application streams with CUDA events, so the same kernels appear to execute in the same order with the same dependencies as in the baseline.

The worker path is kept small and nonblocking. Workers run on a low priority stream with a fixed footprint and reuse preallocated synchronization state so that no device allocations occur after launch. The host never blocks on the device queue; if the queue is full or an operation is oversized or unsupported, that work is immediately routed to the eager path.

Only tiny micro operations such as bias, activation, and normalization on tensors below a configurable threshold are offloaded to workers, while heavier operators always run eagerly. Dynamic segments that enqueue to the worker path are executed in order and must complete a `wait_all()` before any subsequent eager or graph segment starts, which flushes outstanding tasks and preserves happens before relations. The worker path itself never initiates graph capture, preventing circular waits between capture and persistent kernels and avoiding new deadlocks beyond those already present in the baseline implementation.

5 Evaluation

5.1 Micro-benchmarks and Design Validation

This section uses micro-benchmarks to validate our key design decisions. The hardware platform was NVIDIA RTX 6000. We characterize the launch-bound regime that defines our optimization scope (Sec. 5.1.1), demonstrate scalability under fragmentation (Sec. 5.1.2), justify offline pre-capture over runtime JIT (Sec. 5.1.3), derive routing thresholds for persistent workers (Sec. 5.1.4), and verify runtime efficiency (Sec. 5.1.5).

5.1.1 Launch Latency and Regime Characterization

To delimit the effective optimization range, we profiled standard MoE expert computation (Pack-GEMM-Scatter) across matrix sizes 256^3–4096^3. Figure 4 identifies two distinct execution regimes. In the **Compute-Saturation Regime** ($N \geq 1536$), device execution ($> 500\mu$s) saturates the GPU. Consistent with Amdahl's Law, minimizing host overhead yields negligible returns ($1.00\times$) as scheduling is not the bottleneck.

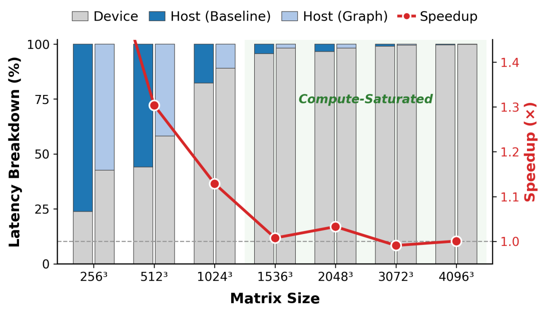

Figure 4: Latency breakdown across matrix sizes. The **Launch-Bound Regime** ($N \leq 512$) is the dominant operating point for online decoding, where host overhead is critical. In the **Compute-Saturation Regime**, device execution dominates.

Conversely, in the **Launch-Bound Regime** ($N \leq 512$), host overhead dominates ($\sim$76%), and our graph replay achieves a **1.35× speedup** by eliminating dispatch costs. Crucially, *online MoE decoding* inherently operates in this launch-bound regime. Unlike prefill, decoding involves a sequence length of 1; combined with the fragmentation from expert routing, effective GEMM dimensions typically remain $N \leq 512$. Consequently, the lack of speedup in the saturation regime is irrelevant to the latency-critical path. This validates our Router's policy: targeting launch-bound fragments for optimization while falling back to eager execution for heavy operators. Finally, this benefit remains robust to interruptions (**1.81**× speedup, see Appendix A).

5.1.2 Scalability under Fragmentation

Beyond individual kernel size, MoE workloads are characterized by high *fragmentation*—a single inference step may involve hundreds of lightweight micro-operations. To verify scalability, we quantified the relationship between kernel count and latency.

Setup. We fixed the matrix size at 512^3 and injected pairs of micro-operations (bias-add and GELU) into the loop, varying the launch count from 3 to 256 per iteration.

Results. Figure 5 shows a stark contrast in scaling. The baseline latency grows **linearly** ($O(N)$) from 0.02ms to 1.81ms as launch overhead accumulates. In contrast, the CUDA Graph latency grows much more slowly, tracking only the *device-side computation* of the added ops. By collapsing the per-iteration launch overhead to $O(1)$, the graph execution path achieves a **2.2× speedup** at 256 micro-ops, demonstrating resilience to the extreme fragmentation typical of MoE glue layers.

5.1.3 The Cost of Graph Construction

We evaluated the feasibility of *runtime JIT capture* as an alternative to pre-capture.

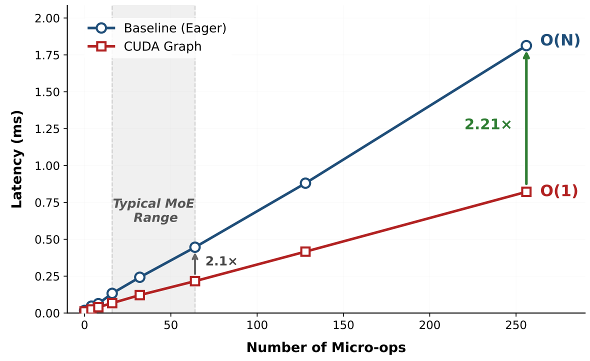

Figure 5: Latency scaling under micro-op fragmentation. Baseline latency grows linearly ($O(N)$) with kernel count, while CUDA Graph amortizes the launch cost ($O(1)$), growing only with actual device computation.

Setup. We measured the instantiation latency of a standard GEMM kernel under "warm" conditions (i.e., with precompiled internal kernels) to determine the theoretical lower bound of construction costs. We profiled the latency across varying dimensions using NVIDIA Nsight Systems.

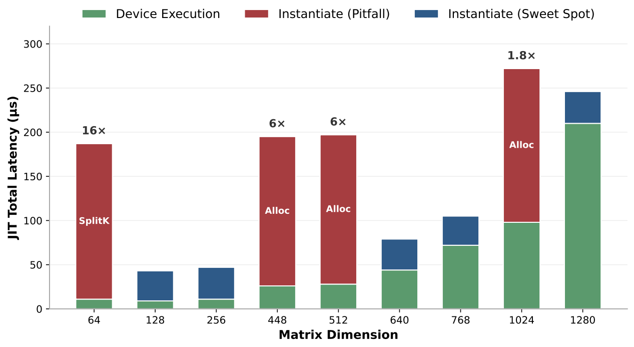

Figure 6: Bimodal distribution of graph instantiation latency. "Pitfall" dimensions (red) trigger expensive implicit workspace allocations, causing latency spikes up to **16.1**× the execution time. This overhead makes runtime JIT capture impractical for small kernels.

Results. Figure 6 reveals that graph construction latency exhibits a bimodal distribution. While "Sweet Spot" dimensions incur moderate overhead, "Pitfall" dimensions (e.g., $N = 64$) trigger implicit allocations, causing instantiation latency to spike to **176±11** μ**s**. This is **16.1**× **slower** than the actual device execution ($\sim$11μs).

Implication. This prohibitive overhead confirms that JIT capture is a "negative optimization" for launch-bound kernels. Consequently, our system adopts an **Offline Pre-capture** strategy, and our bucketing policy explicitly targets "Sweet Spot" dimensions to avoid allocation penalties. (We provide a detailed analysis of cuBLAS allocation in **Appendix B**).

5.1.4 Persistent workers and glue-size threshold

We next characterize when the persistent-worker path is preferable to eager launches for glue-only work. We construct a micro-benchmark that applies a chain of 1,000 bias-add and GELU micro-ops to a tensor, varying the tensor size from 16–128 KB. The worker configuration matches our system setup: a single persistent kernel with 256 threads on a low-priority stream, using a dedicated CUDA memory pool for its queue and scratch buffers.

Figure 7 reports the speedup of the worker path over eager launches as a function of task size. For small tensors (16–32 KB), workers provide 1.2–1.8× speedup by avoiding per-kernel launch overhead. The curves cross near 40 KB: interpolating between the 32 KB (1.16×) and 48 KB (0.81×) points, we estimate a break-even around 39 KB. Beyond 48 KB, the worker path becomes increasingly slower (0.32× at 64 KB, 0.08× at 128 KB), because the single-block persistent kernel cannot fully utilize the GPU, while the eager kernels achieve high SM occupancy.

We therefore configure the router to send glue segments smaller than 40 KB to the worker path and use eager launches for larger segments by default. This threshold is hardware- and workload-dependent, so we expose it as a configuration parameter; Appendix C provides a more detailed breakdown of launch versus compute time and explains the non-monotonic shape of the eager baseline.

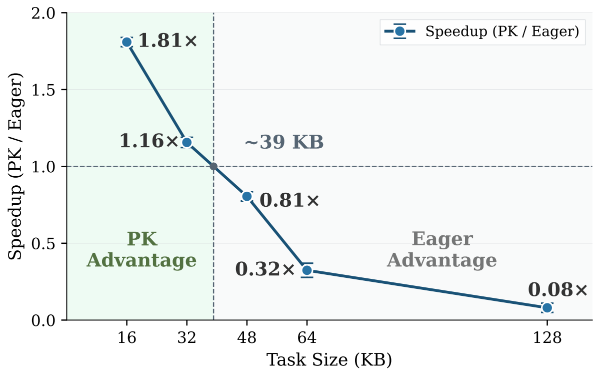

Figure 7: Persistent worker speedup over eager launches for a chain of 1,000 bias+GELU micro-ops at different tensor sizes. Workers provide strong benefits for small glue segments and fall behind eager kernels beyond ∼40 KB, which we use as the default glue-size threshold in the router.

Impl	Avg (ms)	P95 (ms)	Launch	Mem (MB)
Naive Dispatch	40.292	40.451	183	23.18
Kernel Fusion	30.663	30.733	45	30.76
CUDA Graph	30.765	30.774	45	30.76

Table 1: Performance comparison of different MoE forward-pass implementations. Pre-capturing 16 graph buckets takes ∼90 ms in total.

5.1.5 CUDA Proxy Player Runtime Efficiency

Finally, we verified whether our runtime abstraction (routing, hashing, lookup) introduces sufficient CPU overhead to negate graph benefits.

Setup. To isolate the pure software overhead, we implemented a CPU-side micro-benchmark that executes the full routing logic without issuing actual GPU commands. We measured the average latency over 10,000 iterations to capture the steady-state performance.

Results. The benchmark reveals negligible costs: the end-to-end routing consumes only **0.275μs** on average (P99: 0.425μs). This constitutes less than **3%** of the ∼14μs host-side savings provided by CUDA Graphs. Furthermore, the overhead demonstrates $O(1)$ **scalability**, increasing by only ∼18% (0.05μs) even when the graph cache grows to 500 entries. This confirms that our system effectively virtualizes execution while preserving the "bare-metal" performance of CUDA Graphs.

5.2 Macro-benchmark: MoE Inference Under Realistic Workloads

We evaluate CUDA Graphs on a MoE workload that approximates an online inference service, and compare them against naive dispatch and hand-crafted kernel fusion.

Setup. We run on a single NVIDIA A100-SXM4-40GB GPU (40 GB HBM2) paired with an AMD EPYC-Milan virtual CPU configuration (32 vCPUs) and ∼115 GiB host memory. The host system runs Ubuntu 24.04.3 LTS (x86_64) with kernel 6.14.0-33-generic and NVIDIA driver 550.163.01 (CUDA 12.4).

We use a small Transformer-style MoE stack with three blocks, 192-token sequences, a 512-dimensional hidden state, and 8 attention heads. Each block contains self-attention and a Top-2 MoE layer over 8 experts, where each expert is a two-layer feed-forward module with an inner dimension of 960. We run with $B \times S = 384$ tokens per forward pass to mimic latency-sensitive online inference with small batches and moderate expert fan-out.

To avoid framework overheads, we implement the MoE forward pass directly in CUDA C++. We run 20 warmup iterations and report averages over the subsequent 20 iterations. We compare three implementations: (i) a *naive* dispatch baseline that launches each kernel eagerly, (ii) a CUDA Graph replay baseline, and (iii) a fusion-based baseline. The fusion strategy applies only to the two largest and most frequent computations: QKV projection in the attention layer and Top-k gating when selecting experts. We expect fusion and graph replay to reduce kernel launch counts and host-side overhead relative to naive dispatch, with fusion potentially slightly faster but less robust to shape or policy changes.

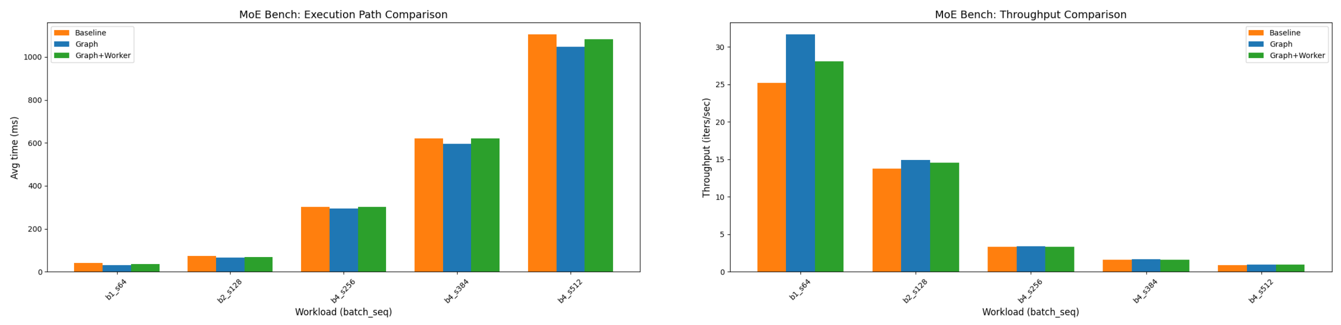

Figure 8: End-to-end MoE inference performance across workload configurations. **Left**: Average latency (lower is better). **Right**: Throughput in iterations per second (higher is better). The graph path provides the largest benefits in the launch-bound regime (small batch$\times$seq), with up to 1.56$\times$ latency reduction and 1.25$\times$ throughput gain at `b1_s64`. Benefits diminish as workloads approach compute saturation.

Results. Table 1 shows that both Kernel Fusion and CUDA Graph reduce average latency by about 24% relative to the naive baseline and cut per-iteration kernel launches from 183 to 45. Both optimized variants use slightly more memory (30.76 vs. 23.18 MiB) due to staging/fusion buffers and static replay allocations.

Pre-capturing the 16 graph buckets takes $\sim$90 ms in total. Graph replay saves roughly 9.5 ms per iteration versus naive dispatch (40.292 ms vs. 30.765 ms), so this one-time capture cost is amortized within about ten iterations. Even a short-running online service would therefore recover the pre-capture overhead quickly.

Implication. On a realistic-but-small MoE forward pass representative of online inference, carefully chosen kernel fusion and CUDA Graph replay deliver nearly identical end-to-end latency and tail improvements over naive dispatch, primarily by removing host launch overhead and reducing fine-grained synchronization. Given fusion's brittleness to shape and policy changes and our broader goal of robustness, CUDA Graphs offer comparable speed with better maintainability, making them a more suitable foundation for the "stable core" of MoE inference in our hybrid design.

Scope. In this section we isolate the stable-core optimization and therefore compare CUDA Graph replay only against naive dispatch and representative hand-tuned fusion. We defer evaluating persistent-kernel glue (Graph+Worker) to 5.3 and 5.4, where it can be assessed under the full request pipeline and its coordination costs can be decomposed explicitly.

5.3 End-to-end Evaluation: CUDA Proxy Player on MoE Inference

To validate our system under realistic conditions, we evaluate CUDA Proxy Player on a complete MoE inference benchmark spanning multiple batch sizes and sequence lengths.

Setup. We construct an MoE inference workload using the same model architecture described in Section 5.2. We vary the batch size $B \in \{1,2,4\}$ and sequence length $S \in \{64,128,256,384,512\}$ to cover both the launch-bound regime and the transition toward compute saturation. For brevity, we denote configurations as `bX_sY`, where `X` is the batch size B and `Y` is the sequence length S (e.g., `b4_s256` means B=4, S=256).

We compare three execution paths:

- **Baseline**: Eager kernel dispatch with no graph capture.
- **Graph**: Piecewise CUDA Graph replay for graph-safe segments, with eager fallback for glue operations.
- **Graph+Worker**: Full CUDA Proxy Player with both CUDA Graph replay and persistent workers handling glue segments.

Results. Figure 8 presents the end-to-end latency and throughput across workload configurations. Several trends emerge that meets our micro-benchmark findings.

Launch-bound benefits. For the smallest workload (`b1_s64`), the Graph path reduces latency from 42 ms to 27 ms, a **1.56$\times$ speedup**. This configuration processes only 64 tokens per forward pass, placing it right in the launch-bound regime where host dispatch overhead dominates. The `b2_s128` configuration (256 tokens) similarly benefits, achieving a **1.24$\times$ speedup** (77 ms $\rightarrow$ 62 ms). These results align with our characterization in Section 5.1.1: when effective GEMM dimensions remain small, eliminating launch overhead yields substantial returns.

Diminishing returns at scale. As batch size and sequence length increase, the speedup from graph replay diminishes. For `b4_s256` through `b4_s512`, the Graph path provides only 3–5% improvement over baseline. At `b4_s512` (2048 tokens), the workload transitions into the compute-saturation regime where device execution time dominates. This behavior matches our design intent: the router should apply graph optimizations where they matter most while gracefully degrading to baseline performance elsewhere.

Throughput perspective. The throughput measurements (Figure 8, right) mirror the latency trends. At `b1_s64`, the Graph path achieves 31.5 iterations/sec compared to the baseline's 25.2 iterations/sec, a **1.25× throughput improvement**. For `b2_s128`, throughput increases from 13.8 to 14.9 iterations/sec (1.08×). In the compute-saturated configurations, all three execution paths converge to similar throughput levels (3.3, 1.6, and 0.9 iterations/sec for `b4_s256`, `b4_s384`, and `b4_s512` respectively), confirming that the optimization headroom is exhausted when GPU compute becomes the dominant factor.

Persistent worker overhead. Contrary to our initial hypothesis, the Graph+Worker path does not outperform the Graph-only path. We see two main reasons:

- **Few eligible glue segments.** Our MoE workload contains relatively few glue segments small enough (<40 KB) to benefit from persistent workers; most routing and scatter operations exceed this threshold.

- **Coordination dominates savings.** The coordination overhead of enqueueing tasks to the device queue and synchronizing between graph segments and worker execution adds latency that offsets potential gains. This supports our router's conservative threshold policy: persistent workers are most effective when workloads contain a dense stream of very small micro-ops, which may be more common in other conditional-computation architectures (e.g., early-exit networks or sparse attention).

Why not kernel fusion. Since the macro-benchmark results already established that CUDA Graphs match fusion performance, the end-to-end evaluation therefore focuses on validating the practical applicability of our graph-based approach under realistic, dynamic conditions rather than re-confirming a comparison that favors neither method on raw performance.

5.4 Decomposing Persistent-Worker Overheads

The end-to-end results raise a key question: *why does the persistent worker path underperform?* Our micro-benchmarks in 5.1.4 demonstrated clear benefits for small tensors with high micro-op density, yet the full system shows no improvement. To understand this gap, we conduct a series of targeted experiments that decompose the overhead into four components: (1) worker configuration sensitivity, to rule out suboptimal tuning; (2) GPU resource contention, to quantify interference between idle workers and graph execution; (3) synchronization barriers, to measure the cost of coordinating between execution phases; and (4) queue scheduling overhead, to compare per-task costs against eager launches. Together, these experiments reveal that the persistent worker path faces fundamental architectural limitations for workloads like MoE inference.

5.4.1 Worker Configuration Sensitivity

To understand whether the persistent worker underperformance stems from suboptimal resource allocation, we swept five worker configurations varying the number of thread blocks and threads per block. Figure 9 shows the results.

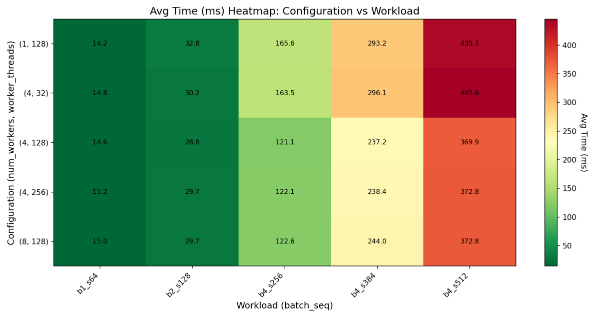

Figure 9: Average latency across worker configurations (num_workers, worker_threads) and workloads. The baseline (4, 128) achieves the best overall performance; neither scaling down nor scaling up improves results.

Multi-block distribution matters. Comparing (1, 128) against (4, 128), the single-block configuration suffers noticeably on medium-to-large workloads: 165.6 ms vs. 121.1 ms at `b4_s256` and 435.7 ms vs. 369.9 ms at `b4_s512`. Distributing workers across multiple blocks improves task pickup responsiveness, as independent blocks can poll and execute concurrently without serializing on a single SM.

Thread count has a floor. The (4, 32) configuration, with only 128 total threads, underperforms the baseline by 35% at `b4_s256` (163.5 ms vs. 121.1 ms). With too few threads, workers lack sufficient parallelism to process tasks quickly once dequeued, creating a bottleneck at the execution stage rather than the polling stage.

Scaling up yields no benefit. Increasing threads per block from 128 to 256, or increasing blocks from 4 to 8, produces nearly identical latency to the baseline (within 3%). This indicates that (4, 128) already provides sufficient worker capacity; additional resources introduce coordination overhead that offsets any throughput gains.

Implication. The baseline configuration (4, 128) sits at a sweet spot, with enough parallelism for responsive task execution without excessive SM contention.

5.4.2 GPU Resource Contention

To isolate whether persistent workers degrade graph segment performance even when idle, we measured graph execution latency while varying the number of worker blocks launched but enqueuing no tasks.

Setup. We execute 100 iterations of the MoE forward pass with 20-iteration warmup, measuring per-segment latency for the 400 static (graph-based) segments. We compare four configurations: 0 (pure Graph), 2, 4, and 8 idle worker blocks, each with 128 threads.

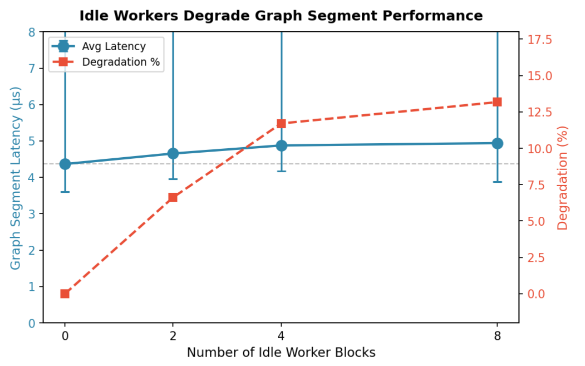

Figure 10: Graph segment latency increases with idle worker blocks. Error bars show min/max range. Baseline (0 workers): 4.366 µs. Degradation reaches 6.6%, 11.7%, and 13.2% for 2, 4, and 8 blocks respectively.

Results. Figure 10 shows a clear monotonic relationship between idle worker count and graph segment degradation. Average segment latency rises from 4.366 µs (no workers) to 4.655, 4.877, and 4.941 μs for 2, 4, and 8 workers (up to 13.2%). Tail latency grows by over 10×. This indicates severe scheduling conflicts.

Analysis. Even on low-priority streams, idle workers pin SM resources. Their continuous polling consumes warp slots and cache capacity, reducing peak occupancy for graph kernels. The near-linear degradation up to 4 blocks and diminishing returns beyond suggests that contention saturates once workers span enough SMs to interfere with most graph kernel launches.

5.4.3 Synchronization Barrier Overhead

The hybrid Graph+Worker architecture requires explicit synchronization barriers between execution phases: graph segments must complete before workers can process dependent glue tasks, and workers must finish before subsequent graphs can replay.

Setup. We instrument the runtime to measure barrier time (the cumulative cost of flush and wait operations) relative to total iteration time. We run 100 iterations with the Graph+Worker path across three workload configurations: `b1_s64` (64 tokens), `b2_s128` (256 tokens), and `b4_s128` (512 tokens).

Results. Figure 11 reveals an important trend. For the smallest workload (`b1_s64`), barriers consume **14.29%** of the total

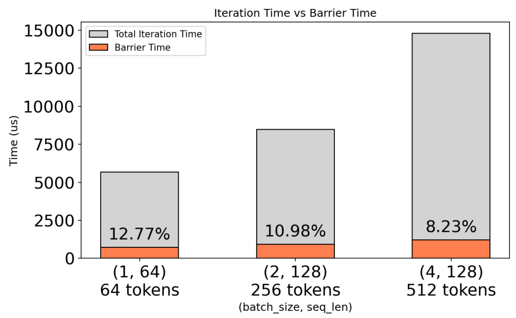

Figure 11: Iteration time breakdown showing barrier overhead across workloads. Barrier time remains roughly constant (∼800–1100 µs) while total iteration time grows with workload size, causing the relative overhead to decrease from 14.29% to 7.59%.

Operation	Avg (µs)	P95 (µs)
Host enqueue + publish	0.370	0.591
Worker poll → start	4.305	7.247
Total PK overhead	**4.675**	**7.838**
Eager kernel launch	2.820	3.076

Table 2: Persistent-worker queue and scheduling overhead vs. eager launch

iteration time (∼5.8 ms). At `b2_s128`, barrier overhead accounts for **13.66%** of ∼8.4 ms. For `b4_s128`, the percentage drops to **7.59%** of ∼14.8 ms.

Analysis. The barrier cost is approximately constant across workloads (∼800–1100 µs per iteration), representing a fixed synchronization overhead. As workload size increases, total iteration time grows proportionally with compute, but barrier time remains fixed. This explains why the persistent worker path is particularly detrimental in the launch-bound regime: when total execution time is small, the fixed barrier overhead consumes a substantial fraction (14%) of the forward pass. For larger, compute-bound workloads, the same absolute barrier cost becomes negligible.

5.4.4 Queue and Scheduling Overhead

Finally, we profile the end-to-end cost of routing a task through the persistent worker path versus launching it directly via eager execution.

Setup. We trace 800 task enqueues across 400 flush cycles, measuring host-side enqueue latency, worker-side scheduling latency (poll → task start), and compare against baseline eager launch timing.

Results. Queue efficiency metrics show an average batch size of 2.0 tasks per flush, with zero queue-full events.

Analysis. The persistent worker path is **1.66× slower** than eager launch (4.675 µs vs. 2.820 µs per task). The overhead breakdown reveals that worker scheduling latency (4.3 µs) alone exceeds the total eager launch cost. This latency reflects the time between the host publishing work and a worker actually beginning execution, encompassing memory fence propagation, worker wake-up from backoff sleep, task decoding, and dispatch.

6 Discussion

Building CUDA Proxy Player surfaced several practical challenges that shaped the final design and prototype.

Coexistence of CUDA Graphs and persistent kernels. The core technical challenge was making piecewise CUDA Graph capture and GPU-resident persistent kernels coexist in a single runtime. CUDA Graph capture is sensitive to concurrent activity on the same device and stream, while persistent kernels are designed to monopolize GPU resources for long periods. Naively mixing the two led to fragile behavior and hard-to-debug failures. Our solution of doing pre-capture before launching persistent workers required some memory overhead and longer initialization time.

Designing a practical routing abstraction. A second challenge was defining a routing abstraction that is both expressive and lightweight. The router must reason about batch size, shape stability, and micro-op density, yet it cannot depend on model-specific details or impose heavy per-request overhead. Currently we define all of these features in terms of simple InferenceRequest structure, but supporting more operations and more complex routing policies may require richer abstractions in the future.

7 Conclusion

This paper presented CUDA Proxy Player, a runtime system that addresses the launch bound regime in conditional computation workloads through a dual path dispatch architecture that combines piecewise CUDA Graphs with persistent GPU workers. Our evaluation confirms the core hypothesis that separating static backbone computation from dynamic glue enables significant performance gains: piecewise CUDA Graph replay achieves up to 1.56× latency reduction and 1.25× throughput improvement in launch bound MoE inference, matching hand tuned kernel fusion while preserving maintainability under shape variation.

Our most important findings concern the limits we uncovered. The persistent worker path, while promising in isolated microbenchmarks, fails to deliver end-to-end benefits due to three compounding effects: worker scheduling latency (4.3 µs) that exceeds eager launch cost (2.8 µs), synchronization barriers that add roughly 0.-1.1 ms of fixed cost per iteration, and SM contention that degrades graph segment performance by 6–13% even when workers are mostly idle. These results expose a fundamental tension in hybrid GPU execution: the coordination mechanisms required to move work between execution modes, such as memory fences, queue polling, and stream synchronization, introduce overhead that is comparable to the kernel launch costs they aim to remove.

This negative result carries a broader lesson. The appeal of device side scheduling is that it removes the CPU from the critical path, but current GPU architectures offer few primitives for truly lightweight task dispatch and inter kernel communication. Persistent kernels must choose between aggressive polling, which wastes SM resources, and backoff sleeping, which adds wake up latency, while the stream and event model imposes serialization that compounds across execution phases. Without hardware support for low overhead device side scheduling, for example dedicated scheduling units or finer grained preemption, hybrid execution will remain effective only in narrow operating regimes.

For practitioners, our results suggest a clear recommendation. When targeting launch bound inference, piecewise CUDA Graphs with offline pre capture and shape bucketing provide robust speedups with limited complexity. Persistent workers should be reserved for workloads dominated by very small micro operations (for example glue tensors under 40 KB) where coordination overhead can be amortized across many tasks. In the common MoE regimes we study, the simpler graph only path delivers nearly all available performance while avoiding the failure modes of hybrid execution.

Looking forward, we see several promising directions. Compiler integration could automate the identification of graph safe segments and bucket boundaries, reducing manual configuration. Emerging GPU features such as CUDA Dynamic Parallelism 2.0 and thread block clusters may eventually offer the low latency device side dispatch primitives that persistent workers currently lack. Finally, the detailed breakdown we provide for SM contention, barrier cost, and queue latency suggests concrete targets for both runtime and hardware co design. We hope these findings guide system designers who work within today's constraints and hardware architects who are considering which primitives would make hybrid GPU execution practical at scale.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, Nov. 2016. USENIX Association.

[2] M. DeepSpeed. Deepspeed-inference: Multi-gpu inference with optimized kernels for large models. https://www.deepspeed.ai/2021/03/15/inference-kernel-optimization.html, 2021.

[3] T. Gale, D. Narayanan, C. Young, and M. Zaharia. Megablocks: Efficient sparse training with mixture-of-experts, 2022.

[4] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, Nov. 2020.

[5] J. He, J. Zhai, T. Antunes, H. Wang, F. Luo, S. Shi, and Q. Li. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 120–134, New York, NY, USA, 2022. Association for Computing Machinery.

[6] S. Hsia, U. Gupta, M. Wilkening, C.-J. Wu, G.-Y. Wei, and D. Brooks. Cross-stack workload characterization of deep recommendation systems, 2020.

[7] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention, 2023.

[8] K. K. W. Ng, H. M. Demoulin, and V. Liu. Paella: Low-latency model serving with software-defined gpu scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 595–610, New York, NY, USA, 2023. Association for Computing Machinery.

[9] NVIDIA. Getting started with cuda graphs. https://developer.nvidia.com/blog/cuda-graphs/, 2019.

[10] NVIDIA. Tensorrt-llm: Large language model inference engine. https://github.com/NVIDIA/TensorRT-LLM, 2023.

[11] P. Tillet, H. T. Kung, and D. Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL '19)*, pages 1–10, London, UK, 2019. ACM.

[12] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

A Sensitivity Analysis: Dynamic Inputs

In Section 5.1.2, we demonstrated the benefits of CUDA Graph under static shapes. However, production MoE inference involves dynamic behaviors that prevent monolithic graph execution. To validate the robustness of our piecewise design, we evaluated the performance impact of interleaved non-graphable operations.

Setup. We simulated a dynamic inference scenario using a $512 \times 512 \times 1024$ matrix size (representing a medium-small expert). We injected Host-to-Device (H2D) memory copies (4MB transfer every 8 iterations) and varying numbers of micro-operations (from 0 to 50) into the execution loop. This setup tests whether the overhead of "breaking" the graph for data movement negates the benefits of graph replay.

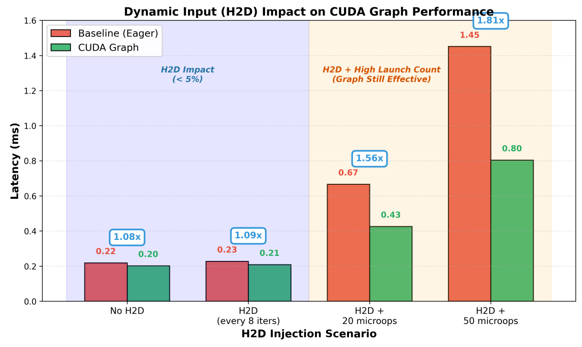

Figure 12: Impact of Host-to-Device (H2D) operations on speedup. H2D operations execute outside the graph boundaries. The results show that even with frequent H2D interruptions, the graph path retains significant performance gains.

Results. As shown in Figure 12, H2D operations introduce minimal overhead (approximately 3.5%–4.1%) for both baseline and graph modes. Crucially, even when combined with high kernel launch counts (e.g., H2D + 50 micro-ops), the

CUDA Graph path maintains a **1.81× speedup** compared to the baseline (down slightly from 1.90× without H2D).

Conclusion. This result confirms that the performance degradation caused by necessary interruptions is marginal ($< 5\%$). It validates that executing H2D operations outside the graph (as "glue" segments) does not break the launch overhead reduction benefits of the graph segments, verifying the robustness of our piecewise execution strategy in online inference scenarios.

B Root Cause of Instantiation Latency

In Section 5.1.3, we reported a bimodal distribution in graph construction latency. Here, we provide a detailed root cause analysis based on differential profiling using NVIDIA Nsight Systems.

B.1 Bimodal Distribution Mechanism

We observed that for specific tensor dimensions (e.g., $N = 64, 448, 512, 1024$), instantiation latency spikes to $\sim 170\mu$s, while for others (e.g., $N = 128, 640$), it remains low ($\sim 35\mu$s). Through API tracing, we identified two primary causes for the "Pitfall" dimensions:

1. **Implicit Workspace Allocation:** For certain non-aligned shapes, cuBLAS triggers an implicit `cudaMallocAsync` call during graph instantiation to allocate temporary workspace. This driver-level allocation dominates the latency.

2. **Split-K Kernels:** For very small M dimensions (e.g., $M = 64$), cuBLAS selects a Split-K reduction algorithm, which inserts additional reduction nodes into the graph, increasing instantiation complexity.

B.2 Extended Characterization

Table 3 presents the construction latency across the full spectrum of matrix sizes, including those outside the launch-bound regime.

Size	Exec (μs)	Instantiate (μs)	Behavior
64^3	11	**176**	Pitfall (SplitK)
128^3	9	34	Sweet Spot
512^3	28	**169**	Pitfall (Alloc)
4096^3	6200	20	Fast Path

Table 3: Extended breakdown of construction overhead. Note the non-linear scaling at 4096^3.

The Large Matrix Anomaly. Interestingly, at $N = 4096$, the capture time drops to $\sim 20\mu$s. We attribute this to driver-side "fast paths" for large, perfectly aligned power-of-two textures where parameter checking is trivial. However, this optimization is irrelevant for system design: at 4096^3, the execution time (>6ms) dwarfs the capture cost (<0.3%), rendering the capture overhead negligible regardless of the path taken.

Design Implication. Our system's **Shape Bucketing** policy is explicitly derived from this analysis. We quantize dynamic shapes to the nearest "Sweet Spot" dimensions (e.g., 128, 640, 768) to avoid implicit allocation penalties, ensuring deterministic low-latency behavior.

C Explaining the 48 KB vs. 64 KB eager anomaly

One seemingly counter-intuitive result in Section 5.1.4 is that the eager baseline at 64 KB (1,682 μs) is *faster* than at 48 KB (3,230 μs), even though the 64 KB tensor is larger. This section explains why this behaviour is expected once we separate host launch overhead from device execution time.

Step 1: kernel time is roughly constant across sizes. Using Nsight Systems, we measure per-kernel GPU time for the eager bias-add and GELU kernels in this experiment. Across all tensor sizes from 16 KB to 128 KB, each micro-op kernel takes roughly 1.1–1.2 μs of device time; the additional elements in larger tensors are absorbed by launching more blocks and using more SMs in parallel. In other words, the GPU compute time per micro-op is almost constant and does *not* explain the 48 KB vs. 64 KB gap.

Step 2: launch overhead dominates small tensors. Nsight also shows that the average host-side launch overhead is ≈ 4.3 μs per kernel, roughly 3–4× the GPU compute time. With 1,000 bias and 1,000 GELU kernels, the aggregate launch time is on the order of 20 ms, while the total GPU compute time is only a few milliseconds. For small tensors (16–48 KB), each kernel launches relatively few blocks and completes quickly; there is little opportunity to overlap host launches with ongoing GPU work, so the end to end latency is effectively "launch-bound."

Step 3: 64 KB crosses into a more parallel regime. At 64 KB, each eager kernel launches enough blocks to keep nearly all SMs busy. The GPU compute phase becomes long enough that many of the subsequent host launches can overlap with in-flight kernels. As a result, the *effective* per-op cost of launch+compute drops, and the total eager time at 64 KB is lower than at 48 KB, even though the tensor is larger. This matches the per-element analysis in our raw logs: the nanoseconds per element at 16 KB are several times higher than at 64 KB, indicating poor GPU utilization in the small-tensor regime.

Implication for PK. In contrast, our persistent-worker configuration deliberately uses a single small block on a low-priority stream to limit its SM footprint. Under this constraint, PK cannot exploit the same occupancy and overlap benefits as eager for large tensors, so its per-op cost grows with tensor size. The 48 KB vs. 64 KB anomaly in the eager baseline is therefore not a measurement artefact, but a consequence of transitioning from a launch-bound, under-utilized regime (16–48 KB) to a more compute-saturated, highly parallel regime (64 KB+). This is precisely why PK is only attractive for small glue segments and why we treat 40 KB as a conservative threshold.