EVENT TENSOR: A UNIFIED ABSTRACTION FOR COMPILING DYNAMIC MEGAKERNEL

Hongyi Jin [*1] **Bohan Hou** [*1] **Guanjie Wang** [*†2] **Ruihang Lai** [*1] **Jinqi Chen** [3] **Zihao Ye** [3] **Yaxing Cai** [3] **Yixin Dong** [1] **Xinhao Cheng** [1] **Zhihao Zhang** [1] **Yilong Zhao** [4] **Yingyi Huang** [3] **Lijie Yang** [5] **Jinchen Jiang** [6†] **Gabriele Oliaro** [1] **Jianan Ji** [1] **Xupeng Miao** [7] **Vinod Grover** [3] **Todd C. Mowry** [1] **Zhihao Jia** [1] **Tianqi Chen** [1 3]

ABSTRACT

Modern GPU workloads, especially large language model (LLM) inference, suffer from kernel launch overheads and coarse synchronization that limit inter-kernel parallelism. Recent megakernel techniques fuse multiple operators into a single persistent kernel to eliminate launch gaps and expose inter-kernel parallelism, but struggle to handle dynamic shapes and data-dependent computation in real workloads. We present *Event Tensor*, a unified compiler abstraction for dynamic megakernels. Event Tensor encodes dependencies between tiled tasks, and enables first-class support for both shape and data-dependent dynamism. Built atop this abstraction, our Event Tensor Compiler (ETC) applies static and dynamic scheduling transformations to generate high-performance persistent kernels. Evaluations show that ETC achieves state-of-the-art LLM serving latency while significantly reducing system warmup overhead.

1 INTRODUCTION

Efficient deployment of machine learning (ML) applications requires minimizing latency and maximizing hardware utilization, making system performance optimization (Kwon et al., 2023; Zhu et al., 2025; Ye et al., 2025; Zhong et al., 2024) a critical research frontier. As GPUs continue to scale in speed and parallelism, several forms of system overhead in conventional GPU scheduling models have emerged as dominant bottlenecks that constrain end-to-end efficiency.

The first source of overhead arises from kernel launches. Current systems such as PyTorch (Paszke et al., 2019) launch GPU kernels sequentially from the host CPU (Figure 1, upper left). During LLM inference, each autoregressive decoding step may involve hundreds or even thousands of fine-grained operations, where the launch overhead cannot be effectively amortized. Each kernel launch typically incurs 5–10 μs of latency, while the fastest kernels may complete in 2 μs, making the launch overhead dominant.

The second source of overhead stems from kernel boundaries, which enforce implicit synchronization between consecutive kernels. In many cases, later kernels depend only on a subset of results from prior ones; in principle, these kernels could be overlapped or pipelined to improve throughput. However, the boundaries between kernels hinder such fine-grained inter-kernel parallelism, leaving significant performance on the table.

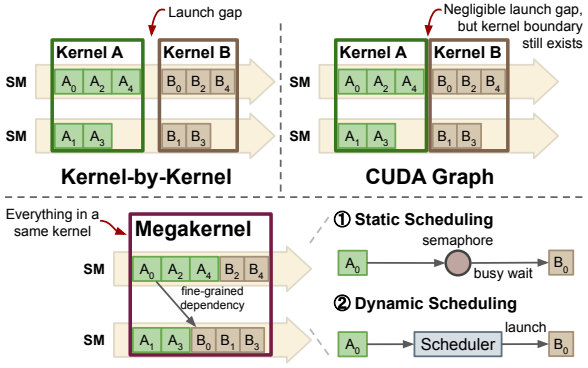

Figure 1. Different GPU scheduling models. Kernel-by-kernel and CUDA Graph scheduling models enforce a coarse-grained sequential execution. Megakernels break operations into smaller tasks, achieving inter-kernel parallelism.

Several recent efforts have attempted to partially address these limitations. First, many systems adopt runtime techniques such as CUDA Graphs (Gray, 2019) (Figure 1, upper right), which reduce kernel launch overhead by capturing and replaying a fixed sequence of kernels. However, CUDA Graphs preserve kernel boundaries and thus cannot expose inter-kernel parallelism.

[*]Equal contribution [†]Work done while at CMU. [1]Carnegie Mellon University [2]Shanghai Jiao Tong University [3]NVIDIA [4]University of California, Berkeley [5]Princeton University [6]Tsinghua University [7]Peking University. Correspondence to: Hongyi Jin <hongyij@andrew.cmu.edu>.

Proceedings of the 9th MLSys Conference, Bellevue, WA, USA, 2026. Copyright 2026 by the author(s).

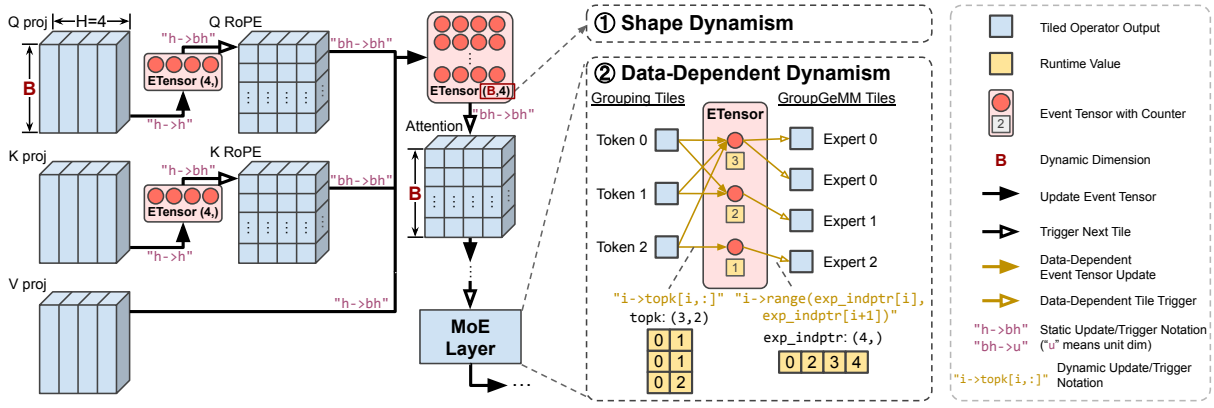

Figure 2. Event Tensor abstraction overview. A computation graph (left) is partitioned into tiled operators (*tasks*), and the Event Tensor captures fine-grained dependencies between tasks as a first-class, symbolic-shaped object, handling the primary sources of dynamism inherent to LLM serving: ① Shape Dynamism: Tiled tensors and Event Tensors have symbolic dimensions, such as the dynamic batch size **B**. ② Data-Dependent Dynamism: The MoE layer (right) details how dependencies are resolved at runtime. Data-dependent updates and triggers (yellow arrows) use runtime-computed values such as `topk` and `exp_indptr` to dynamically manage the task execution.

More recently, *megakernel* optimizations (Spector et al., 2025; Cheng et al., 2025) have emerged as a promising alternative (Figure 1, lower). The key idea is to fuse multiple operators into a single persistent kernel, eliminating kernel launch overheads and enabling inter-kernel parallelism. Each operator is decomposed into fine-grained tiles of computation, or *tasks*, which are distributed across streaming multiprocessors (SMs). The tasks and their dependencies form a task graph, whose execution is orchestrated through lightweight runtime signaling to preserve dependency while maximizing concurrency.

Despite its promise, deploying megakernels for LLM inference workloads remains challenging for two key reasons:

Dynamism challenges. Modern LLM serving workloads are inherently dynamic. With the introduction of continuous batching, the system must handle variable input shapes. Supporting such dynamic shapes within a single megakernel is challenging, as it often requires regenerating or recompiling the kernel for every possible shape. This can incur prohibitive startup latency or become impossible when the space of potential shapes is too large. Furthermore, models such as Mixture-of-Experts (MoE) introduce data-dependent control flow (e.g., expert routing), which requires dynamic tracking of fine-grained task dependencies to exploit inter-operator parallelism. Current approaches lack abstractions to express such fine-grained data dependency. Notably, this challenge is not unique to megakernels—runtime dynamism also poses significant difficulties for conventional methods such as CUDA Graphs, where recapturing and managing CUDA Graphs across dynamic shapes is a major pain point for production LLM serving systems. These dynamism challenges are particularly important for emerging latency-sensitive applications such as real-time agentic workflows and interactive coding assistants, where low-batch inference dominates and inter-kernel parallelism is critical for reducing per-request latency.

Programmability challenges. Megakernel programming introduces substantial complexity. Developers must reason about complex, fine-grained dependencies among tasks, which are error-prone and difficult to maintain. The shape and data-dependent dynamism further complicate this process. Moreover, multiple task-management strategies may be desirable depending on the workload. For instance, *static scheduling* assigns each SM a predetermined queue of tasks before kernel launch, while *dynamic scheduling* employs an on-GPU scheduler to dispatch ready tasks at runtime. Ideally, developers should be able to seamlessly select or combine these scheduling strategies without reimplementing the entire kernel, enabling workload-specific scheduling.

In this paper we present **Event Tensor** (Figure 2), an abstraction designed to simplify the compilation and execution of dynamic megakernels. We define an *event* as a primitive representing the completion of a set of tasks at the granularity of GPU SMs. Because megakernels partition operators into a large number of tile-level tasks, the corresponding synchronization events naturally form multi-dimensional structures similar to data tensors. An *Event Tensor* is a multi-dimensional array of such events, providing a compact, first-class representation for fine-grained synchronization within a megakernel. While semaphore-based synchronization is a well-known primitive, our core novelty lies in elevating these primitives into first-class tensors within the compiler IR. By unifying events into tensor form, Event Tensors leverage existing compiler support for symbolic shapes (Ansel et al., 2024; Lai et al., 2025), allowing tensor dimensions to remain symbolic and thereby compactly representing dynamic-shape computations. Furthermore, Event Tensors express data-dependent dynamism through index expressions that map task coordinates to event coordinates. Built upon this abstraction, ETC automatically transforms

these dependencies into highly optimized, persistent megakernels, generalizing optimizations that previously required manual, specialized engineering.

Building on the Event Tensor abstraction, we develop a systematic compiler pipeline that automatically fuses and schedules operators for inter-kernel parallelism. Starting from a computational graph annotated with explicit operators and Event Tensors, the compiler applies a series of scheduling transformations to lower the program into an executable megakernel. These transformations support multiple scheduling strategies—ranging from fully static to dynamically load-balanced execution—each representing a different trade-off between synchronization overhead and runtime adaptability. By unifying previously manual fusion and hand-crafted scheduling techniques (Spector et al., 2025; Cheng et al., 2025) into compiler transformations, ETC significantly reduces engineering effort to construct megakernels, while improving their runtime performance.

We evaluate ETC on a diverse set of LLM serving workloads, comparing against highly competitive, industry-level baselines (e.g., vLLM and SGLang) that already employ aggressive optimizations such as CUDA Graphs, Programmatic Dependent Launch (PDL), and `torch.compile`. Our results show that ETC achieves substantial speedups over these systems while efficiently supporting both shape- and data-dependent dynamism, without requiring runtime graph recapture or recompilation. For tensor-parallel workloads, our compiler-driven overlap of computation and communication achieves up to 1.40x speedup on fused GEMM and Reduce-Scatter kernels. For data-dependent workloads such as MoE, our megakernels outperform specialized libraries by up to 1.23x. In dynamic-shape, low-batch inference scenarios, ETC matches or exceeds the performance of these highly optimized inference systems, while reducing engine warm-up overheads by up to 3.5x. Achieving even moderate speedups over such strong baselines translates to substantial economic value at datacenter scale. Beyond raw speed, ETC achieves true ahead-of-time (AOT) compilation for dynamic workloads, completely eliminating runtime compilation overhead and the management complexity of repeated CUDA Graph recapture—a major pain point in production serving systems. Furthermore, ETC automates megakernel fusion of complex, data-dependent subgraphs (e.g., MoE layers, GEMM+communication), significantly reducing programming complexity while remaining composable with existing serving engines. ETC has been incorporated into a major open-source system. The simplicity and generality of the Event Tensor abstraction, together with the accompanying compiler framework, can benefit the broader machine learning systems and compiler community.

```python
class IRModule:
  @device_func
  def partial_sum(i: int, j: int, A: Tensor, B: Tensor):
    # i, j are task coordinates
    # The following tasks can be implemented at CTA/cluster level
    B[i*32 : i*32+32, j] = sum(A[i*32 : i*32+32, j*32 : j*32+32])

  @device_func
  def final_sum(i: int, B: Tensor, C: Tensor):
    # i is the task coordinate
    # The following tasks can be implemented at CTA/cluster level
    C[i*32 : i*32+32] = sum(B[i*32 : i*32+32, :])

  @graph_func
  def main_graph(A: Tensor(("n*32", 128))) -> Tensor(("n*32",)):
    n = sym_var()  # an integer symbolic variable
    E = ETensor((n,), wait_count=4)
    B: Tensor((n * 32, 4)) = call_device(
      IRModule.partial_sum, tile_num=(n, 4),
      args=[A], in_edges={}, out_edges={E: "ij->i"},
    )
    C: Tensor((n * 32,)) = call_device(
      IRModule.final_sum, tile_num=(n,),
      args=[B], in_edges={E: "i->i"}, out_edges={},
    )
    return C
```

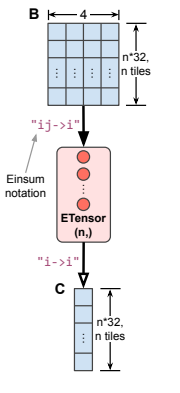

Figure 3. Example Event Tensor–based program.

2 Event Tensor Abstraction

2.1 Language Constructs

We first introduce the main language constructs in Event Tensor–based programs.

Device Function. A device function defines a grid of tasks launched in parallel on the GPU. Each launch is parameterized by a multidimensional coordinate, where each coordinate identifies a task tile executed on a streaming multiprocessor (SM). Each task can include specialized logic, such as warp specialization or tensor core calls.

Event Tensor. An Event Tensor is a multi-dimensional structure whose elements represent events—the completion of task sets at the SM level—following established practices in parallel programming systems (Blumofe et al., 1995; Treichler et al., 2014; Bauer et al., 2012; Dagum & Menon, 1998). Each element has an initial wait count recording the number of tasks it depends on and supports several operations: `E[i].notify()` signals task completion, `E[i].wait()` blocks until the event is triggered, and in dynamic scheduling, events can also trigger dependent tasks. Compared with approaches that manage standalone events individually, Event Tensors greatly reduce task-graph management overhead when scaling to millions of fine-grained events in real LLM inference workloads.

Graph function. A graph function represents a computational graph consisting of `call_device` calls that explicitly launch device functions with specified task shapes. Unlike traditional computational graphs, it includes both data tensors and Event Tensors. Each device function launch can annotate explicit input/output dependencies and coordinate mappings that track fine-grained task relationships through Event Tensors.

Figure 3 shows an example Event Tensor–based program. The general program can be viewed as a compact representation of task graphs in the form of "producer task → event → consumer task". We use generic lambda functions

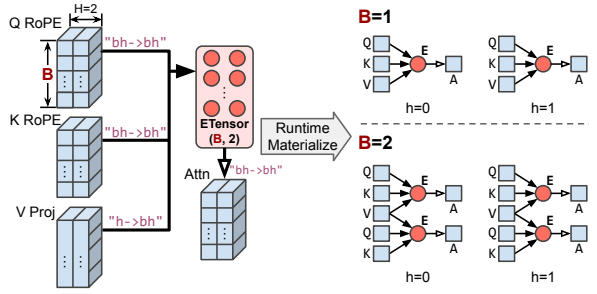

Figure 4. **Event Tensor handles shape dynamism** with symbolic-shape tensors that define a template for dependency graphs. At runtime, the template is instantiated with concrete shape values (e.g., producing a 1×2 graph for batch size 1 or a 2×2 graph for batch size 2) without recompilation or repeated graph capture.

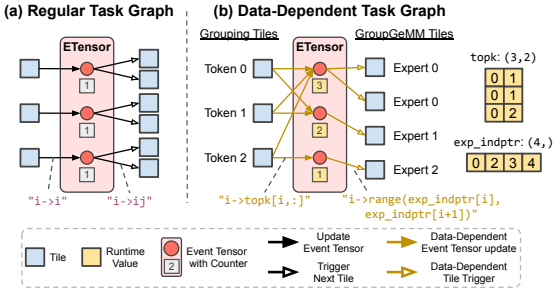

Figure 5. **Event Tensor handles data-dependent dynamism** by (a) A regular workload with static dependencies. (b) A data-dependent MoE workload where runtime tensors `topk` and `exp_indptr` define an irregular task graph, enabling data-dependent event updates and task triggering.

to represent event task relations.[1] These dependency annotation implicitly maps to event notifications at the end of each producer task and waiting at the beginning of each consumer task. We find this notation to be sufficient for most use cases. Importantly, we also allow device function to explicitly take in Event Tensors as arguments and call the event notify and wait inside each task. The first-class support of the Event Tensor in device function enables us to represent advanced use cases. It also enables explicit fusion optimizations as transformations within the representation that we will discuss in detail in the next section.

2.2 Representing Fine-Grained Dependencies

To illustrate how Event Tensors are used in practice, we walk through the example shown in Figure 3. It shows a task graph that performs a summation over the inner axis of the input tensor A, which has symbolic shape $(n \times 32, 128)$: $C[i] = \sum_{k \in [0,128)} A[i,k]$. The example adopts a split-K algorithm that divides the summation into two stages $B[i,j] = \sum_{k \in [j*32, j*32+32)} A[i,k]$ and $C[i] = \sum_{k \in [0,4)} B[i,k]$, where the first stage computes partial sums of each row into B, and the second stage aggregates them to produce C. In a traditional kernel-by-kernel approach, tasks in the second stage are launched only after all tasks in the first stage have completed. However, each output row $C[i]$ depends only on the corresponding row $B[i,:]$, meaning that its computation can proceed concurrently with partial sums of other rows. To capture this fine-grained dependency, we partition the computations of B and C into finer tasks and introduce an Event Tensor E:

$$\begin{aligned} \text{Task } \hat{B}_{i,j}: &\quad B[i*32 : i*32+32, j], \\ \text{Event } E_i: &\quad E[i], \\ \text{Task } \hat{C}_i: &\quad C[i*32 : i*32+32]. \end{aligned}$$

[1]For simplicity, we use Numpy einsum-like notations (Harris et al., 2020) in figures and example codes.

With task partitioning, the dependency relations becomes $\hat{B}_{i,j} \to E_i$ ($\hat{B}_{i,j}$ produces E_i) and $E_i \to \hat{C}_i$ ($\hat{C}_i$ consumes E_i), where each E_i corresponds to the completion of 32 consecutive rows of B starting from row $32 * i$.

The `main_graph` function provides the description of the overall computation where the primitive `call_device` first launches `partial_sum` and then `final_sum` function. The dependencies between `partial_sum` and `final_sum` tasks are specified through `out_edges` and `in_edges` arguments.

2.3 First-Class Dynamic Shape Support

The Event Tensor-based graph can be viewed as a more compact representation of the task graph representations (Treichler et al., 2014) in parallel systems, where each event element and tasks are explicitly represented as individual nodes and edges materialized as task graphs in the runtime. In our case, we can use a single tensor to represent thousands of events. The Event Tensor representation also allows us to bring in first-class support for symbolic dimensions. For example, we can have an Event Tensor shape to contain symbolic variables such as batch size of sequence length. The symbolic-shape Event Tensor graph serves as a generic template that corresponds to different task graphs (Figure 4) at runtime. This powerful representation gives us the ability to overcome limitations in static task-graph systems such as CUDA Graph, and *ahead of time* optimize dynamic shape Event Tensor graph for multiple shapes without recompilation or repeated graph capture, providing greater flexibility in handling dynamic shape workloads.

2.4 Supporting Data-Dependent Dynamism

A critical challenge in modern workloads is handling irregular, data-dependent task graphs. A representative example is the Mixture-of-Experts (MoE) layer (Shazeer et al., 2017), where input tokens are dynamically routed to different expert sub-networks based on routing decisions computed at

runtime. An efficient MoE implementation typically first groups tokens according to their assigned experts and then uses GroupGEMM operators to compute the results for all tokens within each group.

Dynamic routing introduces fine-grained, data-dependent task dependencies that are unknown at compile time (specifically, which experts GroupGEMM tile processes which tokens). This dynamism poses a significant challenge for traditional compilers and schedulers, which assume a static task graph with fixed dependencies. To efficiently represent such dynamic workloads, we need an abstraction that can (1) determine at runtime which tasks each consumer task depends on, and (2) trigger a variable number of consumer tasks based on runtime data. The Event Tensor abstraction is designed precisely for this purpose. It manages data-dependent task graphs through two core mechanisms:

Data-Dependent Event Update. Unlike conventional task graphs that support only static dependencies (Figure 5a), the Event Tensor abstraction allows dynamic event dependencies (Figure 5b). In the MoE example, runtime routing decisions stored in the `topk` tensor determine which grouping tiles (one per token) update which events (one per expert). Each expert's event counter is initialized to the number of tokens routed to it, and this initialization occurs dynamically at runtime, together with the computation of `topk`.

Data-Dependent Task Triggering. Similarly, an event can trigger a runtime-dependent number of tasks. Based on routing decisions in `topk`, we can compute how many tokens each expert must process and, therefore, how many GroupGEMM tiles each expert requires. As illustrated in Figure 5b, this information is encoded in the tensor `exp_indptr`, which stores the prefix sum of GroupGEMM tiles to be triggered per expert.[2] Leveraging `exp_indptr`, we enable data-dependent triggering, where expert `i` activates tiles in the range `(exp_indptr[i], exp_indptr[i+1])`.

Together, these two mechanisms allow the compiler to generate megakernels that efficiently adapt to highly dynamic workloads—cases that static task graphs handle poorly. Combined with the symbolic-shape support described above, all compilation in ETC occurs offline; at inference time, the compiled binary handles both shape and data-dependent dynamism with zero compilation overhead.

Note that the overall dependency chain remains strictly feed-forward (Figure 2, right): Attention Output → Token Routing (TopK) → Token Grouping → Token Computation (GroupGEMM). Computing TopK depends only on the preceding Attention output—a standard static dependency. The data-dependent Event Tensor mechanisms described above govern only the later stages, where routing results dynamically determine which grouping tasks notify which expert events and how many GroupGEMM tiles each expert triggers.

[2] `indptr` is a term commonly used in sparse matrix representations such as the compressed sparse row (CSR) format.

3 Event Tensor Compilation

This section describes optimizations in ETC that make use of the proposed Event Tensor abstractions.

3.1 Static Scheduling and Transformation

Static scheduling fuses multiple device functions together by explicitly distributing tasks across streaming multiprocessors (SMs) ahead of time. As a result, each task is pre-assigned to the task queue of a specific SM. Task dependencies are managed through low-level synchronization primitives, such as counter-based semaphores and event-triggered waits. This approach achieves minimal synchronization overhead and is particularly effective for predictable workloads where SM partitions can be optimized ahead of time.

We implement a static scheduling transformation in ETC with three main steps (Algorithm 1): (1) construct per-SM execution queues on the host; (2) generate a persistent main loop that lets each SM execute tasks continuously without relaunching; and (3) lower Event Tensor dependencies into explicit `notify()` and `wait()` calls to enforce fine-grained execution order. Figure 6 illustrates this process for GEMM + Reduce-Scatter, fundamental to tensor-parallel execution. It fuses the device functions into a single persistent kernel, whose main loop continuously fetches tasks from a precomputed queue in `tile_scheduler` and issues `notify()` at the end of GEMM tasks and `wait()` at the beginning of Reduce-Scatter tasks.

Figure 7 illustrates how the statically fused GEMM (MM) + Reduce-Scatter (RS) kernel operates in practice. Each RS task depends on two MM tasks (i.e., an RS tile spans twice the size of an MM tile), so the initial counter for each event is two. At T_1, MM0 on SM0 finishes and notifies the Event Tensor, reducing the counter to one. The RS task, statically scheduled next on SM0, cannot yet proceed and enters a spin-wait state. Between T_1 and T_2, SM1 continues executing MM0, keeping the GPU busy. At T_2, MM0 on SM1 completes, decrementing the counter to zero and satisfying the dependency, which releases the RS task from its wait loop and allows execution to begin on SM0.

Plain static scheduling does not naturally support dynamic workloads. To handle shape dynamism, we sample a set of representative shapes; unseen shapes reuse the execution queue of the next larger sampled value. To handle data-dependent dynamism, we conservatively assume the worst case for Event Tensor updates and triggers by rewriting related `notify()` and `wait()` operations to

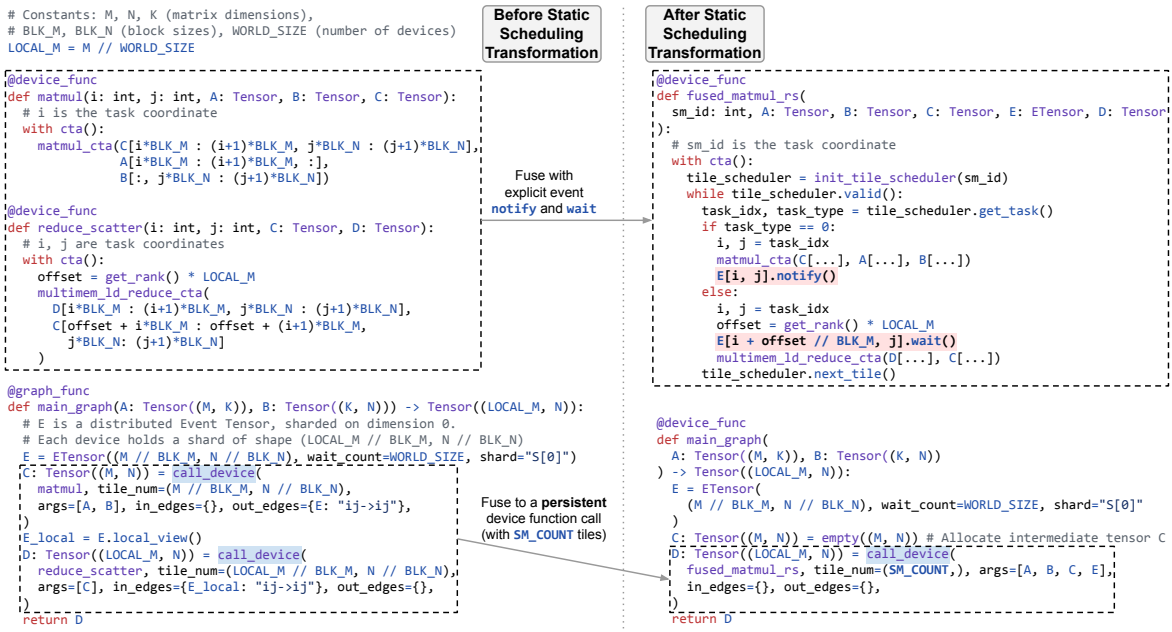

Figure 6. GEMM + Reduce-Scatter before and after static scheduling transformation. Two separate device functions are fused into a single persistent function, with explicit `notify` and `wait` calls on the Event Tensor to coordinate dependencies.

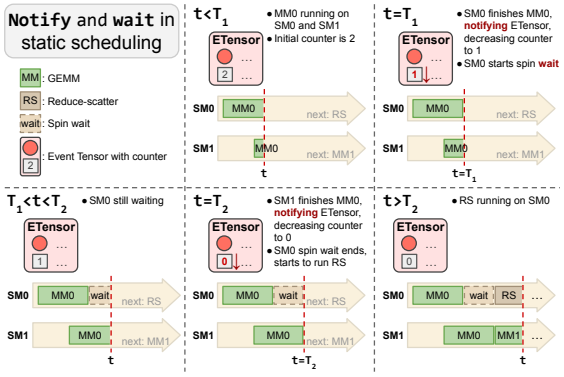

Figure 7. Notify-and-wait mechanism for static scheduling.

`E[0].notify()` and `E[0].wait()`. For simplicity, we use a round-robin policy to construct execution queues.

Algorithm 1 Static Scheduling Transformation in ETC

Input: A module `mod` containing a tile-level dataflow graph `G` with Event Tensor dependencies.
Output: An updated module with a fused, statically scheduled megakernel.
`mod_updated` ← `mod.Copy()`
`static_schedule` ← `GenerateStaticSchedule(G)`
`fused_kernel` ← `NewPersistentKernel()`
// Embed the pre-computed schedule into global memory
`fused_kernel.AddBuffer(static_schedule)`
for all `task_grid` in `G` **do**
  `fused_kernel.AddDispatchLogic(task_grid)`
  for all `event` in `task_grid.in_edges` **do**
    `fused_kernel.AddWaitLogic(event)`
  end for
  `fused_kernel.AddTileLogic(task_grid)`
  for all `event` in `task_grid.out_edges` **do**
    `fused_kernel.AddNotifyLogic(event)`
  end for
end for
`mod_updated.Replace(G, fused_kernel)`
return `mod_updated`

3.2 Dynamic Scheduling and Transformation

When task execution time is unpredictable, dynamic scheduling improves load balance across SMs. ETC implements Event Tensor–based dynamic scheduling using a lightweight on-GPU task scheduler. When an event is triggered—after all its dependent tasks complete—it atomically **pushes** all associated consumer tasks into the scheduler, marking them ready for execution. Any available SM can then atomically **pop** a ready task and execute it.

We implement a dynamic scheduling transformation in our compiler[3]. Figure 8 shows the transformed code for the same GEMM (MM) + Reduce-Scatter (RS) example discussed in §3.1. Figure 9 illustrates the pushpop mechanism. At T_1, MM0 on SM0 finishes and decrements the event counter to one; SM0, now idle, immediately pops a ready task (MM1) from the scheduler. At T_2, MM0 on SM1 completes, reducing the counter to zero and triggering the RS task to be pushed into the scheduler. SM1 then pops the RS task and begins execution. This entire process of dependency tracking and task dispatching occurs efficiently on the GPU, without requiring any host-precomputed task queue.

Dynamic scheduling inherently supports both kind of dynamism, because tile execution order is decided on the fly at runtime, when the symbolic shape value and runtime value are already resolved by scheduler. Our implementa-

[3]The pseudocode of the dynamic scheduling transformation is provided in Appendix A.

After Dynamic Scheduling Transformation

On-Device Scheduler
Tile Queue
RS
Device functions fused with task push and pop

```
@device_func
def fused_matmul_rs(
  sm_id: int, A: Tensor, B: Tensor,
  C: Tensor, E: ETensor, D: Tensor,
):
  # sm_id is the task coordinate
  with cta():
    tile_scheduler = init_tile_scheduler(sm_id)
    while tile_scheduler.valid():
      task_idx, task_type = tile_scheduler.pop_task()
      if task_type == 0:
        i, j = task_idx
        matmul_cta(C[...], A[...], B[...])
        state = E[i, j].notify()
        if state == 0:
          tile_scheduler.push_task(task_idx, task_type)
      else:
        i, j = task_idx
        offset = get_rank() * LOCAL_M
        multimem_ld_reduce_cta(D[...], C[...])
      tile_scheduler.next_tile()

@device_func
def main_graph(
  A: Tensor((M, K)), B: Tensor((K, N))
) -> Tensor((LOCAL_M, N)):
  # Same as the statically fused "main_graph"
  ...
```

Figure 8. GEMM + Reduce-Scatter after dynamic scheduling transformation. Task `push` and `pop` are inserted, and task execution is dynamically coordinated by the scheduler.

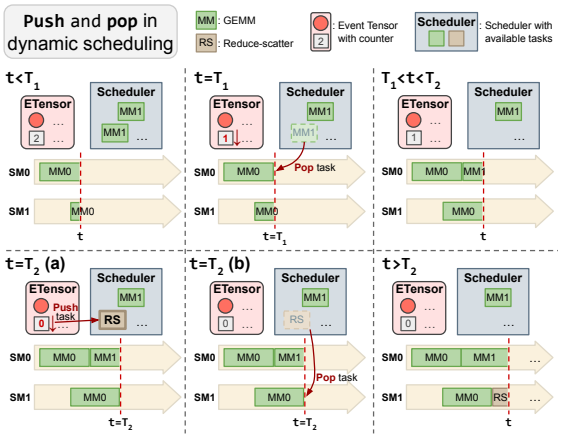

Figure 9. Push-and-pop mechanism for dynamic scheduling.

tion of push-pop interface uses a centralized queue in global memory shared across all SMs. We choose this design for its implementation simplicity, though we acknowledge potential contention at scale. We also discuss the runtime optimization for dynamic scheduler in Appendix E.

Trade-off between static and dynamic scheduling. The choice between static and dynamic scheduling reflects a classic trade-off. Static scheduling leads to minimal scheduling overhead, making it well-suited for predictable workloads. Dynamic scheduling, by contrast, offers flexibility for data-dependent dynamic workload or unpredictable task completion times, naturally achieving load balance at the cost of a small runtime overhead of task queue pushes and pops.

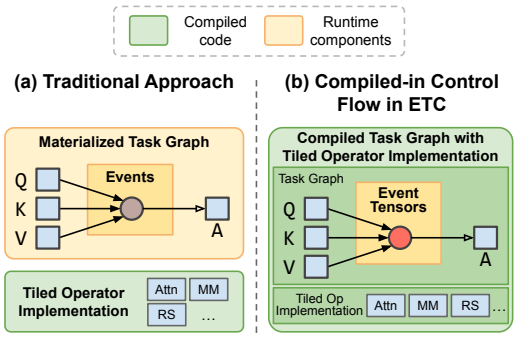

Figure 10. Comparison of runtime architectures. (a) The task graph in traditional runtime executor is materialized in memory, and only tiled operators are compiled. (b) ETC compiles scheduling logic into megakernels without runtime task graph materialization.

3.3 Lowering to Minimal Runtime

The static and dynamic scheduling in our compiler allows us to encapsulate low-level task dependencies and their handling directly in the transformed program, resulting in minimal needs for corresponding supporting runtime (Figure 10). Specifically, each Event Tensor is lowered to an integer tensor, reusing the existing tensor data structure and avoiding any dedicated runtime data structures for events. The `notify()` and `wait()` operations on this integer tensor are implemented with efficient hardware atomics: `notify()` performs an atomic decrement, while `wait()` spin-waits for the counter to reach zero. Our runtime data state consists solely of these integer tensors and scheduler's task queue. This compiled-in approach brings a smaller runtime requirement compared to a typical task-graph approach, where the entire task graph needs to be materialized in memory and relies on a generic task executor that traverses this graph to launch device functions.

3.4 End-to-End Compilation Flow

The end-to-end compilation flow of ETC[4] starts from an unoptimized computational graph where Event Tensors are defined and operators are already partitioned into CTA-level tiles—either user-specified through a kernel DSL such as Triton (Tillet et al., 2019) or provided as compiler builtins. In our implementation, device functions are written in a TVM-based DSL (Hou et al., 2026) that supports standard tile-based programming. Notably, the Event Tensor abstraction is DSL-agnostic: its dependency graph and scheduling logic can be integrated into other compiler stacks (e.g., Triton, CuteDSL) without fundamental design conflicts.op The graph first undergoes standard graph-level optimizations, including memory planning, similar to existing deep learning compilers (Sabne, 2020; Lattner et al., 2021; Lai et al., 2025; Ansel et al., 2024). Next, tile-level optimizations refine each operator by determining low-level details such as

[4] A compilation pipeline figure can be found in Appendix B.

hardware instruction mapping and pipelining strategies. The graph is then transformed via either the static or dynamic scheduling pass described in §3.1 and §3.2. The resulting fused device function is emitted as GPU code in the persistent-kernel style. A subsequent prefetching pass generates weight-prefetching functions for tiled operators based on user annotations, enabling each tile to prefetch weights before input activations arrive. Finally, if static scheduling is chosen, the compiler computes the per-SM task order and materializes it as the megakernels static execution queue.

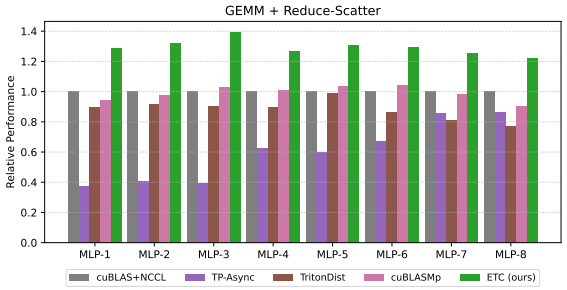

Figure 11. Performance results of GEMM + Reduce-Scatter on 8 B200s with dynamic scheduler.

4 EVALUATION

We implement ETC as a series of compiler passes building upon Apache TVM. Notably, our proposed abstraction can be applied to other compilers as well. This section provides evaluations to answer the following key questions:

- How effectively does the Event Tensor abstraction manage fine-grained dependencies for workloads with both static task graphs (§4.1) and task graphs with shape dynamism and data-dependent dynamism (§4.2)?
- Do ETC-compiled megakernels achieve lower end-to-end latency in dynamic low-batch serving scenarios? (§4.3)
- Does the Event Tensor's support for shape dynamism eliminate the significant engine warmup overhead required by static-shape runtime just-in-time (JIT) compilation and graph capture systems? (§4.4)
- Do static and dynamic scheduling exhibit distinct performance trade-offs on different workloads? (§4.5)

All experiments are conducted on a server equipped with 8 NVIDIA B200 GPUs connected via NVLink, running Ubuntu 24.04 with PyTorch 2.8.0, CUDA 13.0 and driver version 580.82.07. We choose B200 as it represents state-of-the-art hardware; the Event Tensor abstraction operates at the compiler IR level and is not specific to any particular GPU generation. We evaluate ETC against state-of-the-art deep learning compilers, specialized libraries, and high-performance LLM serving systems. Existing megakernel frameworks are tailored to single-batch inference and thus cannot be fairly compared under dynamic-shape or data-dependent workloads.

4.1 Fused Communication and Computation Performance

To evaluate how effectively the Event Tensor abstraction optimizes static compute-communication patterns, we benchmark two fundamental fused kernels for tensor-parallel LLMs: GEMM + Reduce-Scatter and All-Gather + GEMM. These kernels are critical for minimizing latency and maximizing hardware utilization in distributed inference. We use MLP configurations derived from a range of modern LLMs, fixing the tensor-parallel size to 8 and the number of tokens to 8192 in all experiments. The configuration details are provided in Appendix C. We compare ETCs generated kernels against several baselines, with implementation choices tailored to each workloads characteristics:

- GEMM + Reduce-Scatter: The Reduce-Scatter collective is implemented using CUDA multimem PTX instructions. We employ ETCs **dynamic scheduler** to handle unpredictable workloads arising from network contention and fluctuation, where its ability to adapt and balance tasks on the fly is most effective.
- All-Gather + GEMM: The All-Gather operation uses a ring algorithm implemented via the copy engine (DMA). We use the **static scheduler**, as only GEMM tiles execute on SMs following the data arrival order dictated by the ring algorithm. A precomputed static schedule effectively overlaps communication and computation with minimal runtime overhead.

The baselines for comparison include:

- cuBLAS+NCCL: A non-overlapped baseline executing cuBLAS and NCCL kernels sequentially, representing performance without fusion.
- TP-Async (Liang et al., 2025): A PyTorch-based approach that manually orchestrates asynchronous operations for overlap.
- Triton Distributed v0.0.2-rc (Zheng et al., 2025): A compiler-based system that generates overlapping kernels, serving as a state-of-the-art open-source baseline.
- cuBLASMp (NVIDIA, 2023): A high-performance, multi-process fused-kernel library that overlaps distributed computation and communication.

Figure 11 and Figure 12 show clear improvements over the baselines, particularly on larger model configurations, achieving up to a 1.40x execution time speedup over the cuBLAS+NCCL baseline for both workloads. Among the fused baselines, TP-Async's coarse-grained splitting can lead to chunks that are either too small to saturate SMs or

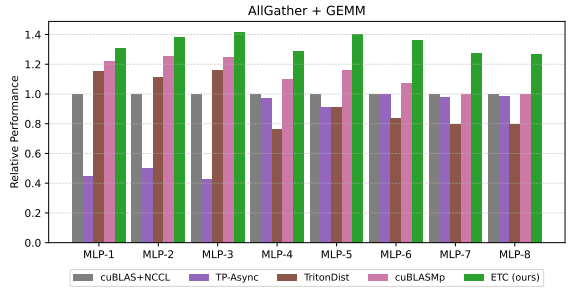

Figure 12. Performance results of All-Gather + GEMM on 8 B200s with static scheduler.

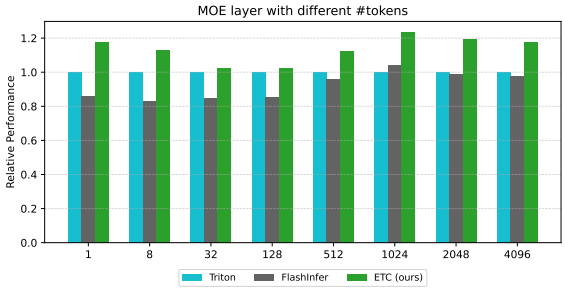

Figure 13. Performance results of MoE layer on a single B200.

too large to effectively hide communication latency, and Triton-Dist's experimental B200 support means its Triton-based GEMM is not yet fully optimized for the Blackwell architecture. As a result, the unfused cuBLAS+NCCL baseline is sometimes competitive with these fused approaches, underscoring the difficulty of achieving efficient fusion. The consistent performance advantage of ETC stems from the Event Tensor abstraction. By representing fine-grained dependencies as a first-class Event Tensor, our compiler can transform monolithic operations into a deeply pipelined task graph. This abstraction also allows our unified scheduling transformations to be applied effectively: we use the dynamic scheduler for GEMM + Reduce-Scatter to handle any potential unpredictability of communication latency, and the static scheduler for All-Gather + GEMM to orchestrate overlap with the predictable ring algorithm with minimal overhead. This fine-grained, compiler-driven approach keeps both compute (SMs) and network resources continuously busy, achieving a degree of overlap that matches or exceeds existing systems.

4.2 Mixture-of-Experts (MoE) Layer Performance

To evaluate our Event Tensor abstraction's ability to manage task graphs with shape dynamism and data-dependent dynamism, this subsection benchmarks a complete MoE layer with variable number of tokens. While existing systems use efficient persistent GroupGEMM kernels, they still require a sequence of separate launches for a complete MoE layer. ETC allows us to fuse the entire data-dependent MoE dataflow into a single megakernel, and we evaluate its performance against existing optimized multi-kernel baselines. We benchmark a complete MoE layer in Qwen3-30B-A3B, which has 128 experts with a top-k of 8. The workload consists of processing a variable number of input tokens. For this workload, we use ETC's dynamic scheduler, as its adaptive load balancing is ideal for the irregular, data-dependent task graph. We compare against several baselines:

- Triton 3.4.0 (Tillet et al., 2019): A highly optimized MoE implementation widely used in state-of-the-art serving systems, including SGLang (Zheng et al., 2024) and vLLM (Kwon et al., 2023).
- FlashInfer 0.2.14.post1 (Ye et al., 2025): A high-performance library providing optimized kernels for LLM inference, including fused MoE kernels.

Figure 13 plots the relative end-to-end performance for the MoE layer across different token counts. ETC's megakernel approach significantly outperform the best baseline, achieving up to a 1.23x speedup at 1024 tokens. Between the two baselines, FlashInfer's GroupGEMM is more optimized for larger token counts, while Triton benefits from fusing gather/scatter into GroupGEMM; their relative ranking thus varies with token count. The performance gains of ETC are a direct result of the Event Tensor abstraction and dynamic scheduling transformation. Firstly, data-dependent Event Tensors break the global synchronization barrier in the baselines, creating a fine-grained pipeline between the two GroupGEMM stages in MoE, which also reduces wave quantization by smoothing out SM allocation across fused operators. Secondly, and more importantly for MoE, our on-chip dynamic scheduler provides superior load balancing for the irregular token routing, minimizing SM idle time and consistently surpassing all other methods as token counts grow.

4.3 End-to-End Low-Batch Serving Performance

This subsection tests whether ETC-compiled megakernels achieve lower end-to-end latency in dynamic low-batch serving scenarios, which are increasingly dominant for latency-sensitive applications such as real-time agentic workflows and interactive coding assistants. We focus on the decoding stage, as prefilling and large-batch serving usually have high GPU utilization and benefit marginally from megakernels (except for computation-communication overlap as we showcased in §4.1). Importantly, ETC does not degrade large-batch performance, as hardware utilization of individual operators remains unchanged without extra overhead. We integrate ETC-compiled megakernels for two representative models: the Qwen3-30B-A3B MoE model and the Qwen3-32B dense model, using the static scheduler for both to avoid runtime overhead. The compiled megakernels cover the full decoding pipeline (Attention, RoPE, KV-Cache, Norm, MLP, MoE), not just GEMM. We chose the Qwen

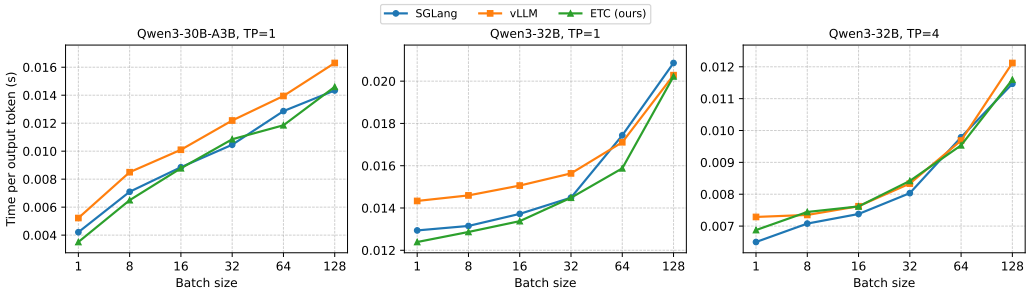

Figure 14. End-to-end performance of model serving on Qwen3-30B-A3B and Qwen3-32B (Lower is better).

family (both dense and MoE variants) as it is architecturally representative of modern LLMs (e.g., LLaMA 3, GPT). The benchmark uses a synthetic dataset with a prefill length of 512 and generates 100 output tokens, with batch size varying from 1 to 128. We measure the time-per-output-token (TPOT) metric, which best reflects the LLM engine decoding performance. The raw kernel running time of one decoding iteration can be found in Appendix D, which concentrates only on the kernel performance and excludes all irrelative overhead (e.g., framework-specific scheduling latencies and other CPU overhead), ensuring a fair comparison of different frameworks. We compare ETC against leading serving systems vLLM (v0.11.0rc2) and SGLang (v0.5.3rc0) both of which use CUDA Graph and torch.compile (Ansel et al., 2024) for performance optimization.

Figure 14 (left) shows the end-to-end performance for Qwen3-30B-A3B, where ETC achieves a 1.48x speedup over vLLM and 1.20x over SGLang at batch size 1. In Figure 14 (mid), for Qwen3-32B, ETC consistently delivers the lowest latency, outperforming vLLM by up to 1.15x at batch size 1, and SGLang by up to 1.09x at batch size 64. In four-way TP tensor-parallel (TP) execution (Shoeybi et al., 2020) for the Qwen3-32B (Figure 14, right), ETC matches performance of vLLM with a speedup varying between 0.99x and 1.06x. The latency of both ETC and vLLM is higher than SGLang in this setting because SGLang's highly optimized CPU scheduler incurs lower distributed runtime overhead. The occasional small gaps where ETC trails the best baseline are attributable to engineering factors—specifically, compiler-generated GEMM tiles that are less tuned than cuBLAS in certain configurations and higher CPU-side overhead in our serving engine—rather than fundamental limitations of the abstraction.

The strong performance of ETC stems from its fused megakernel architecture, enabled by the Event Tensor abstraction and static scheduling transformation. Unlike conventional approaches that launch a sequence of kernels with implicit synchronization at kernel boundaries, ETC executes the entire workload within a single persistent kernel. This compiler-driven fusion enables fine-grained optimizations that are difficult for CUDA-Graph-based systems: it exposes parallel execution in attention (e.g., Q's Norm+RoPE running concurrently with K's Norm+RoPE+CacheAppend), pipelines GroupGEMMs in MoE and GEMMs in MLP to reduce wave quantization, and prefetches model weights before input activations are ready to hide memory latency. ETC's ability to pipeline and overlap operations across operator boundaries is the key for the latency reduction compared to baselines. Crucially, ETC breaks kernel boundaries, successfully applying performance on par with CUDA Graphs to inherently dynamic workloads like MoE.

Table 1. Warmup time of Qwen3-32B model serving using different graph capturing methods.

Method	Warmup Time (s)	# JIT Graph Capture
SGLang (JIT)	583	51
vLLM (JIT)	123	67
ETC (AOT)	35	0

4.4 Warmup Overhead

This subsection evaluates the deployment impact of ETCs compilation strategy by measuring LLM engine warmup overhead. We define warmup time as the total wall-clock time from engine launch to the first request served, including engine initialization, model loading, and all JIT compilation or CUDA Graph capture overheads. We test whether ETCs ahead-of-time (AOT) compilation, enabled by shape dynamism support, can eliminate the runtime cost of just-in-time (JIT) compilation and CUDA Graph capture. Table 1 shows substantial difference: vLLM requires 123 s and SGLang 583 s to warm up on Qwen3-32B, whereas ETC initializes in only 35 s. This speedup stems from the Event Tensor abstraction, whose first-class shape dynamism support enables AOT compilation. While baselines must capture many static CUDA Graphs at runtime to cover different shapes (e.g., 67 for vLLM), ETC compiles a single persistent, shape-generic megakernel offline (107 s for Qwen3-32B). At runtime, it simply loads the precompiled graph, avoiding the repetitive warmup penalty inherent to JIT and runtime-capture approaches.

Table 2. Relative performance of different ETC scheduling methods on MoE layer against unfused megakernel. Higher is better.

Num tokens	1	128	1024	4096
Static	1.03	1.02	1.04	1.02
Dynamic	0.95	1.06	1.08	1.03

Table 3. Relative performance of different ETC scheduling methods on Qwen-3-32B with TP=4 against unfused megakernel. Higher is better.

Batch Size	1	16	32	128
Static	1.09	1.06	1.07	1.06
Dynamic	0.83	0.82	0.85	0.89

4.5 Tradeoff Between Different Scheduling Methods

This section analyzes the performance characteristics and trade-offs of ETC's static and dynamic scheduling strategies, and quantifies the gains from fusion by comparing them to an unfused megakernel baseline. This baseline uses a single event between different operator stages to enforce a global synchronization barrier, simulating a sequential execution model within a single kernel launch. Crucially, the unfused baseline uses identical operator code as ETC, so the speedups reported in Tables 2 and 3 are driven purely by the inter-kernel parallelism unlocked by ETC's fine-grained Event Tensor dependencies—the same sources of gain discussed in §4.2 and §4.3 (reduced wave quantization, weight prefetching, parallel execution, and on-chip load balancing)—rather than better operator-level implementations.

Data-Dependent Workloads. For workloads with data-dependent control flow, such as the MoE layer in §5.2, the dynamic scheduler provides load balancing. As shown in Table 2, dynamic scheduler outperforms static scheduler except on single-batch inference, with the largest speedup being 4.0% over static scheduler and 8.1% over unfused baseline when batch size is 1024. The data-dependent routing of tokens creates an inherent workload imbalance. A rigid static scheduler can cause some SMs to accumulate a queue of longer-running tiles, forcing them to become stragglers while other SMs sit idle.

Regular Workloads. Conversely, for the regular, dense transformer layer workload (analyzed with TP=4 in §5.3), the static scheduler is the clear winner (Table 3). The dynamic scheduler's overhead becomes very large on distributed setting, especially when trying to push tasks to remote task queue. There is also a consistent 6-8% speedup of ETC-static over the ETC-unfused version, a gain purely from fine-grained pipelining.

These results also explain why the multi-GPU evaluation results in Figures 11/12 and Figure 14 appear to show different trends. Figures 11 and 12 represent a bandwidth-bound regime with large batches (8192 tokens), where ETC excels by utilizing fine-grained signaling to overlap communication and computation. In contrast, Figure 14 (right) represents a latency-critical regime with low batches, where communication overhead and CPU scheduling overhead are exposed. This necessitates different scheduling strategies: dynamic scheduling handles large-batch jitter effectively, while static scheduling minimizes overhead for latency-sensitive low-batch tasks.

These findings demonstrate a clear, workload-dependent trade-off, confirming the value of supporting both scheduling transformations within the ETC framework.

5 Related Work

Deep learning compilers such as MLIR (Lattner et al., 2021), XLA (Sabne, 2020), TVM (Chen et al., 2018; Feng et al., 2023; Lai et al., 2025), and the PyTorch compiler (Ansel et al., 2024) have laid the foundation for optimizing deep learning models. These systems perform graph-level optimizations and run execution kernel-by-kernel. CUDA Graph (Gray, 2019) provides a way to drastically reduce launch overhead by capturing and replaying a sequence of kernels, but relies on static input. Machine learning compilers are also developing vertical fusion (Zheng et al., 2020; Niu et al., 2021) and horizontal fusion (Jia et al., 2019; Li et al., 2022) to optimize kernel launch overhead. Rammer (Ma et al., 2020) and Roller (Zhu et al., 2022) perform software launch of tile-based tasks. All the previous works do not have explicit abstraction for fine-grained dependencies tracking and optimizations, and can benefit from our proposed Event Tensor to enable megakernel optimizations. Dynamic tensor compilers such as DynaTune (Zhang et al., 2021), DietCode (Zheng et al., 2022), and SparseTIR (Ye et al., 2023) handle dynamic shapes or sparsity at the single-kernel level; ETC complements them by fusing their operator implementations into megakernels to unlock inter-kernel parallelism.

LLM inference systems such as SGLang (Zheng et al., 2024), vLLM (Kwon et al., 2023), TensorRT-LLM (NVIDIA, 2024), and Orca (Yu et al., 2022) achieve high performance through system optimizations such as continuous batching and speculative execution. Our event tensor compiler can serve as a backend for these frameworks to enable more efficient GPU execution. Recent works (Cheng et al., 2025; Spector et al., 2025) start to build megakernels for LLMs. These approaches only supports single-batch dense model inference, and focus on a single scheduling strategy. The event tensor abstraction proposed in this paper complement these approaches by providing systematic compiler abstraction support for shape dynamism and data-dependent dynamism. The event tensor compiler also supports both static scheduling and dynamic scheduling. CuSync (Jangda et al., 2024) optimizes co-scheduling

of separate kernels on distinct CUDA streams, and FlashMoE (Aimuyo et al., 2025) provides a hand-optimized kernel for distributed MoE. ETC differs from both by fusing entire subgraphs into a single persistent megakernel via a systematic compiler pipeline, generalizing beyond any single operator pattern.

Our approach is closely related to task-based parallel programming models such as Cilk (Blumofe et al., 1995), Legion (Bauer et al., 2012), Realm (Treichler et al., 2014), and OpenMP Tasks (Dagum & Menon, 1998). Most of the previous approaches focus on coarse-grained tasks typically orchestrated by the CPU. Our approach is also related to Graphene (Hagedorn et al., 2023) and Cypress (Yadav et al., 2025) that optimizes a single kernel. Graphene models threads as a tensor with synchronization capabilities, which is conceptually related; however, it targets single-kernel optimization rather than multi-operator megakernel fusion with dynamic shape and data-dependent support. Our approach builds on these previous insights and proposes the event tensor abstraction that compactly represents fine-grained dependencies across operator sub-tasks and runs both static and dynamic scheduling in the GPU streaming multiprocessors.

6 Conclusion

This work introduces Event Tensor, a unified abstraction that expresses fine-grained synchronization for compiling dynamic GPU megakernels. Event Tensor provides first-class support for both shape and data-dependent dynamism. Built on this abstraction, ETC systematically generates high-performance persistent kernels using static and dynamic scheduling. ETC achieves state-of-the-art serving latency while substantially reducing warmup overhead. In the future, we envision higher-level passes that automatically generate Event Tensor task graphs from standard computational graphs, further reducing manual synchronization effort. We hope this work will encourage additional studies of megakernels and highlight new possibilities for ML compilers.

Acknowledgements

We thank all anonymous MLSys reviewers and our shepherd for their constructive feedback and comments. This work is supported in part by gifts from NVIDIA, Google and Amazon. We also acknowledge support from NVIDIA for the DGX B200.

References

Aimuyo, O. J., Oh, B., and Singh, R. Flashmoe: Fast distributed moe in a single kernel, 2025. URL https://arxiv.org/abs/2506.04667.

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A., DeVito, Z., Ellison, E., Feng, W., Gong, J., Gschwind, M., Hirsh, B., Huang, S., Kalambarkar, K., Kirsch, L., Lazos, M., Lezcano, M., Liang, Y., Liang, J., Lu, Y., Luk, C. K., Maher, B., Pan, Y., Puhrsch, C., Reso, M., Saroufim, M., Siraichi, M. Y., Suk, H., Zhang, S., Suo, M., Tillet, P., Zhao, X., Wang, E., Zhou, K., Zou, R., Wang, X., Mathews, A., Wen, W., Chanan, G., Wu, P., and Chintala, S. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, pp. 929947, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640366. URL https://doi.org/10.1145/3620665.3640366.

Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11. IEEE, 2012.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices*, 30 (8):207–216, 1995.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

Cheng, X., Zhang, Z., Zhou, Y., Ji, J., Jiang, J., Zhao, Z., Xiao, Z., Ye, Z., Huang, Y., Lai, R., Jin, H., Hou, B., Wu, M., Dong, Y., Yip, A., Ye, Z., Wang, S., Yang, W., Miao, X., Chen, T., and Jia, Z. Mirage persistent kernel: A compiler and runtime for mega-kernelizing tensor programs, 2025. URL https://arxiv.org/abs/2512.22219.

Dagum, L. and Menon, R. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

Feng, S., Hou, B., Jin, H., Lin, W., Shao, J., Lai, R., Ye, Z., Zheng, L., Yu, C. H., Yu, Y., et al. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 804–817, 2023.

Gray, A. Getting started with cuda graphs, Sep 2019. URL https://developer.nvidia.com/blog/cuda-graphs/.

Hagedorn, B., Fan, B., Chen, H., Cecka, C., Garland, M., and Grover, V. Graphene: An IR for Optimized Tensor Computations on GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, pp. 302313, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399180. doi: 10.1145/3582016.3582018. URL https://doi.org/10.1145/3582016.3582018.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Ro, J. F., Wiebe, M., Peterson, P., Grard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with numpy. *Nature*, 585(7825):357362, September 2020. ISSN 1476-4687. doi: 10.1038/s41586-020-2649-2. URL http://dx.doi.org/10.1038/s41586-020-2649-2.

Hou, B., Jin, H., Wang, G., Chen, J., Cai, Y., Yang, L., Ye, Z., Ding, Y., Lai, R., and Chen, T. Axe: A simple unified layout abstraction for machine learning compilers, 2026. URL https://arxiv.org/abs/2601.19092.

Jangda, A., Maleki, S., Dehnavi, M. M., Musuvathi, M., and Saarikivi, O. A framework for fine-grained synchronization of dependent gpu kernels, 2024. URL https://arxiv.org/abs/2305.13450.

Jia, Z., Padon, O., Thomas, J., Warszawski, T., Zaharia, M., and Aiken, A. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 47–62, 2019.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Lai, R., Shao, J., Feng, S., Lyubomirsky, S., Hou, B., Lin, W., Ye, Z., Jin, H., Jin, Y., Liu, J., et al. Relax: composable abstractions for end-to-end dynamic machine learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 998–1013, 2025.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14. IEEE, 2021.

Li, A., Zheng, B., Pekhimenko, G., and Long, F. Automatic horizontal fusion for gpu kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 14–27. IEEE, 2022.

Liang, W., Liu, T., Wright, L., Constable, W., Gu, A., Huang, C.-C., Zhang, I., Feng, W., Huang, H., Wang, J., Purandare, S., Nadathur, G., and Idreos, S. Torchtitan: One-stop pytorch native solution for production ready LLM pretraining. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=SFN6Wm7YBI.

Ma, L., Xie, Z., Yang, Z., Xue, J., Miao, Y., Cui, W., Hu, W., Yang, F., Zhang, L., and Zhou, L. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 881–897. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/ma.

Niu, W., Guan, J., Wang, Y., Agrawal, G., and Ren, B. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pp. 883898, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454083. URL https://doi.org/10.1145/3453483.3454083.

NVIDIA. cuBLASMp: A High-Performance CUDA Library for Distributed Dense Linear Algebra. https://docs.nvidia.com/cuda/cublasmp/index.html, 2023.

NVIDIA. TensorRT-LLM. https://docs.nvidia.com/tensorrt-llm, 2024.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019.

Sabne, A. Xla : Compiling machine learning for peak performance, 2020.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017. URL https://arxiv.org/abs/1701.06538.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020. URL https://arxiv.org/abs/1909.08053.

Spector, B., Juravsky, J., Sul, S., Dugan, O., Lim, D., Fu, D., Arora, S., and R, C. Look ma, no bubbles! designing a low-latency megakernel for llama-1b, May 2025. URL https://hazyresearch.stanford.edu/blog/2025-05-27-no-bubbles.

Tillet, P., Kung, H.-T., and Cox, D. 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*, pp. 10–19, 2019.

Treichler, S., Bauer, M., and Aiken, A. Realm: An event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 263–276, 2014.

Yadav, R., Garland, M., Aiken, A., and Bauer, M. Task-based tensor computations on modern gpus. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025. doi: 10.1145/3729262. URL https://doi.org/10.1145/3729262.

Ye, Z., Lai, R., Shao, J., Chen, T., and Ceze, L. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 660–678, 2023.

Ye, Z., Chen, L., Lai, R., Lin, W., Zhang, Y., Wang, S., Chen, T., Kasikci, B., Grover, V., Krishnamurthy, A., et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

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

Zhang, M., Li, M., Wang, C., and Li, M. Dynatune: Dynamic tensor program optimization in deep neural network compilation. In *ICLR*, 2021.

Zheng, B., Jiang, Z., Yu, C. H., Shen, H., Fromm, J., Liu, Y., Wang, Y., Ceze, L., Chen, T., and Pekhimenko, G. Dietcode: Automatic optimization for dynamic tensor programs. In Marculescu, D., Chi, Y., and Wu, C. (eds.), *Proceedings of Machine Learning and Systems*, volume 4, pp. 848–863, 2022. URL https://proceedings.mlsys.org/paper_files/paper/2022/file/f89b79c9a28d4cae22ef9e557d9fa191-Paper.pdf.

Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pp. 863–879, 2020.

Zheng, L., Yin, L., Xie, Z., Sun, C. L., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

Zheng, S., Bao, W., Hou, Q., Zheng, X., Fang, J., Huang, C., Li, T., Duanmu, H., Chen, R., Xu, R., Guo, Y., Zheng, N., Jiang, Z., Di, X., Wang, D., Ye, J., Lin, H., Chang, L.-W., Lu, L., Liang, Y., Zhai, J., and Liu, X. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler, 2025. URL https://arxiv.org/abs/2504.19442.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, 2024.

Zhu, H., Wu, R., Diao, Y., Ke, S., Li, H., Zhang, C., Xue, J., Ma, L., Xia, Y., Cui, W., Yang, F., Yang, M., Zhou, L., Cidon, A., and Pekhimenko, G. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 233–248, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/zhu.

Zhu, K., Gao, Y., Zhao, Y., Zhao, L., Zuo, G., Gu, Y., Xie, D., Ye, Z., Kamahori, K., Lin, C.-Y., et al. {NanoFlow}: Towards optimal large language model serving throughput. In *19th USENIX Symposium on Operating Systems*

Design and Implementation (OSDI 25), pp. 749–765, 2025.

Algorithm 2 Dynamic Scheduling Transformation in ETC

Input: A module `mod` containing a tile-level dataflow graph `G` with Event Tensor dependencies..
Output: An updated module with a fused, dynamic scheduled megakernel.
```
mod_updated ← mod.Copy()
fused_kernel ← NewPersistentKernel()
// Not runtime scheduler. Only provides push/pop functions
scheduler ← GPUScheduler()
fused_kernel.AddPopLogic(scheduler.f_pop_tasks)
for all task_grid in G do
  fused_kernel.AddDispatchLogic(task_grid)
  fused_kernel.AddTileLogic(task_grid)
  for all event in task_grid.out_edges do
    fused_kernel.AddCompleteOnLogic(event,
    scheduler.f_push_tasks)
  end for
end for
mod_updated.Replace(G, fused_kernel)
return mod_updated
```

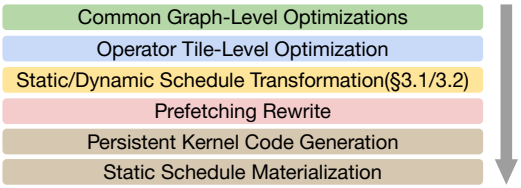

Figure 15. End-to-end compilation pipeline in ETC.

A Dynamic Scheduling Pseudocode

Algorithm 2 describes the compiler pass to transform an event tensor graph to a dynamically scheduled megakernel. A call to `scheduler.pop_tasks` is inserted whenever an SM finishes its current task, while a call to `scheduler.push_tasks` is inserted when the completion of a task decrements the associated event counters to zero, thereby unblocking dependent tasks.

B ETC End-to-End Compilation Flow

Figure 15 summarizes the end-to-end compilation flow of ETC, as described in §3.4.

C MLP configuration used in §4.1

Table 4 shows MLP configurations used in fused communication and computation evaluation, which are derived from a range of modern LLMs.

D Raw Kernel Time Evaluation in End-to-End LLM Serving

§5.3 reports the time-per-output-token (TPOT) metric of ETC and baselines in end-to-end LLM serving. The TPOT number includes not only the total GPU kernel execution time but also CPU-side overheads, such as framework-specific request scheduling latency. To provide a more direct and fair comparison unaffected by unrelated overheads, this section evaluates the raw GPU kernel execution time in end-to-end LLM serving, using the same baselines and experimental settings as in §5.3.

Table 4. Model configurations for MLP, where S = sequence length, H = hidden dim, I = intermediate size.

MLP Configurations				
Name	**Source Model**	**S**	**H**	**I**
MLP-1	Qwen3-8B	8192	4096	12288
MLP-2	LLaMA-3.1-8B	8192	4096	14336
MLP-3	Gemma-2-9B	8192	3584	14336
MLP-4	Gemma-2-27B	8192	4608	36864
MLP-5	Qwen3-32B	8192	5120	25600
MLP-6	LLaMA-3.1-70B	8192	8192	28672
MLP-7	GPT-3-175B	8192	12288	49152
MLP-8	LLaMA-3.1-405B	8192	16384	53248

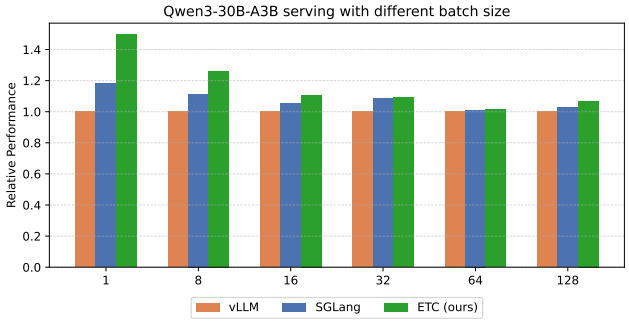

Figure 16. Raw kernel relative performance results of Qwen-30B-A3B on a single B200.

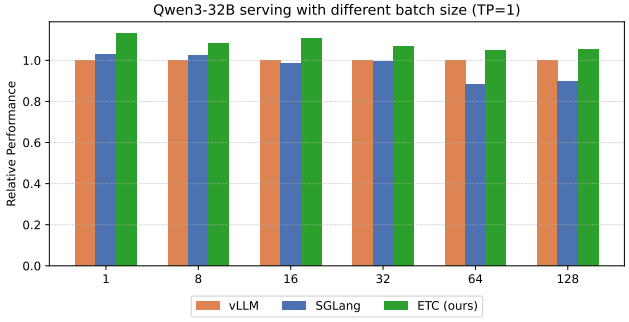

Figure 17. Raw kernel relative performance results of Qwen-32B on a single B200.

Figure 16 shows the relative raw kernel end-to-end performance of ETC and baselines across multiple batch sizes on Qwen3-30B-A3B, where ETC achieves consistent speedups over the baselines, with the most significant improvement being 1.49x over vLLM and 1.27x over SGLang at batch size 1. With the Event Tensor abstraction supporting data-dependent dependencies in MoE, ETC executes the entire

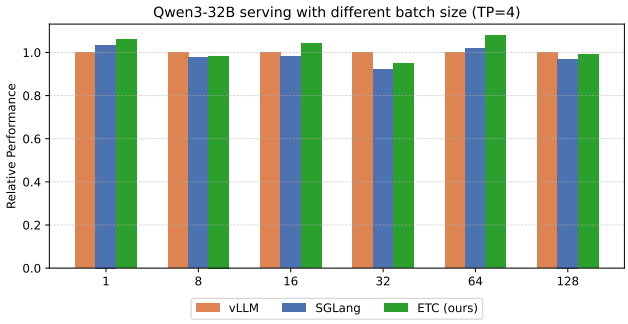

Figure 18. Raw kernel relative performance results of Qwen-32B on four B200s with tensor parallelism.

MoE model within a single kernel, enabling optimizations such as increased parallelism across attention operators, fine-grained pipelining between GroupGEMMs, and model-weight prefetching.

Figure 17 and Figure 18 present the raw kernel performance of Qwen3-32B serving on a single B200 and on four B200s with tensor parallelism, respectively. Under the single-GPU setting (TP = 1), ETC achieves consistent gains over both vLLM and SGLang across all batch sizes, with up to a 1.13x speedup over vLLM at batch size 1 and an average improvement of about 7%. In the tensor-parallel case (TP = 4), ETC maintains comparable or better performance across all settings, achieving up to a 1.08x speedup over vLLM while sustaining similar scalability as batch size increases. This on-par performance reflects the effectiveness of ETC's megakernel design and static scheduling based on the Event Tensor abstraction.

E Dynamic Scheduler Runtime Optimization

We adopt an early push strategy to hide scheduling overhead. Rather than waiting for a task's dependencies (its producer tasks) to finish executing, the scheduler proactively pushes a consumer task into the ready queue as soon as all of its producer tasks have been dispatched to SMs (not requiring task completion), and the dependency is guaranteed by the extra wait prior to the consumer's execution. This proactive measure prevents the overhead of the push operation from falling onto the critical path. The push is performed concurrently with the execution of the producer tasks, effectively overlapping the scheduling cost with the preceding computation.