Event Tensor

A Unified Abstraction for Compiling Dynamic Megakernel

Hongyi Jin*, Bohan Hou*, Guanjie Wang*, Ruihang Lai*, Jinqi Chen, Zihao Ye, Yaxing Cai, Yixin Dong, Xinhao Cheng, Zhihao Zhang, Yilong Zhao, Yingyi Huang, Lijie Yang, Jinchen Jiang, Gabriele Oliaro, Jianan Ji, Xupeng Miao, Vinod Grover, Todd C. Mowry, Zhihao Jia, Tianqi Chen

MLSys 2026

LLM decoding pays a tax at every kernel boundary

Each decode step

re-runs the whole model — hundreds of small kernels per output token.

One decode step

~ hundreds

of kernel launches

LLM decoding pays a tax at every kernel boundary

Each decode step

re-runs the whole model — hundreds of small kernels per output token.

Each kernel boundary costs **5–10 µs**.

The fastest kernels finish in just **1–2 µs**.

One decode step

~ hundreds

of kernel launches

× 5–10 µs per boundary

≈ 1+ ms / token

of pure overhead

LLM decoding pays a tax at every kernel boundary

Each decode step

re-runs the whole model — hundreds of small kernels per output token.

Each kernel boundary costs **5–10 µs**.

The fastest kernels finish in just **1–2 µs**.

⇒ **Boundary overhead dominates compute.**

One decode step

~ hundreds

of kernel launches

× 5–10 µs per boundary

≈ 1+ ms / token

of pure overhead

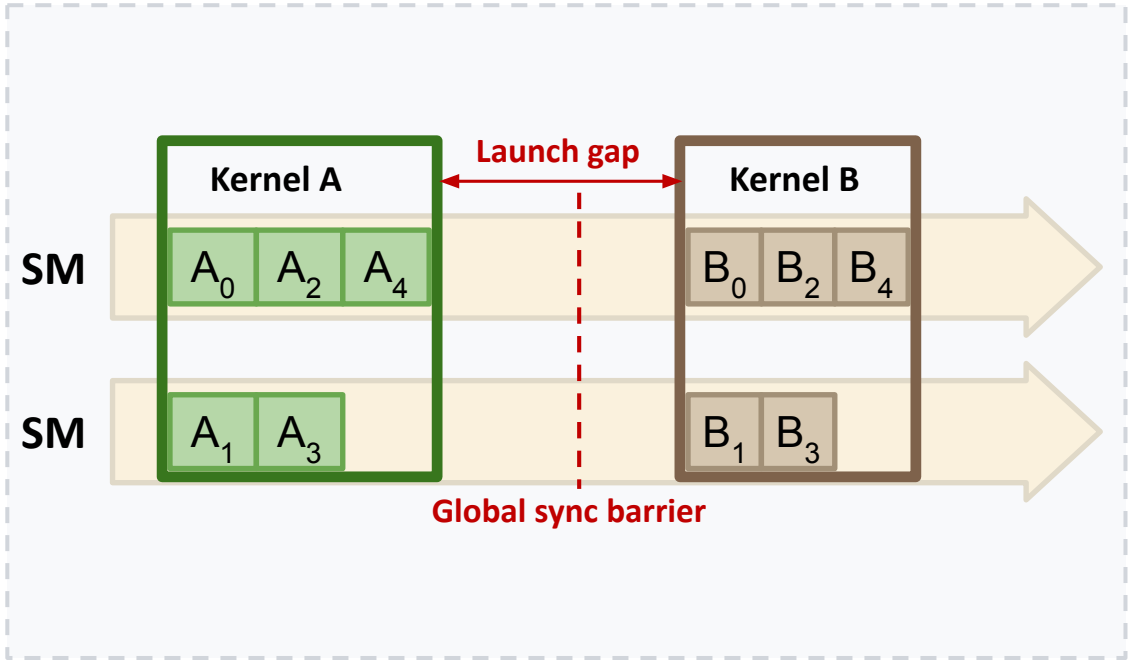
Two bottlenecks of kernel-by-kernel execution

1 **Launch gap**

Fixed 5–10 µs overhead paid at every kernel boundary.

2 **Global sync barrier**

Every SM must finish kernel A before any SM starts kernel B.



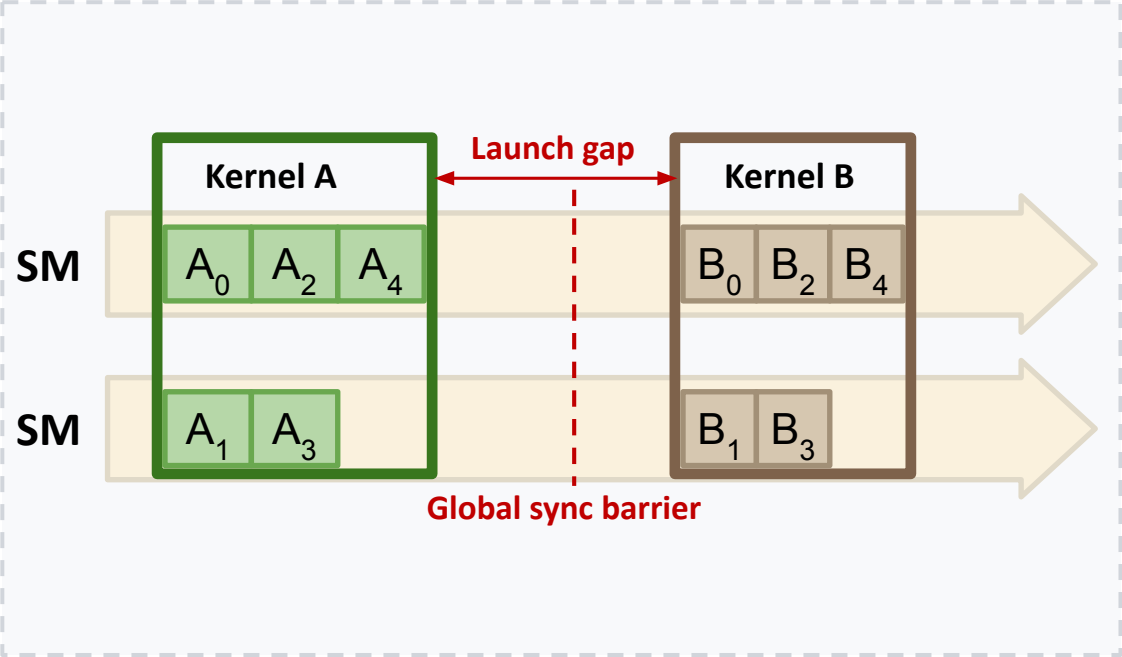
Two bottlenecks of kernel-by-kernel execution

1 **Launch gap**

Fixed 5–10 µs overhead paid at every kernel boundary.

2 **Global sync barrier**

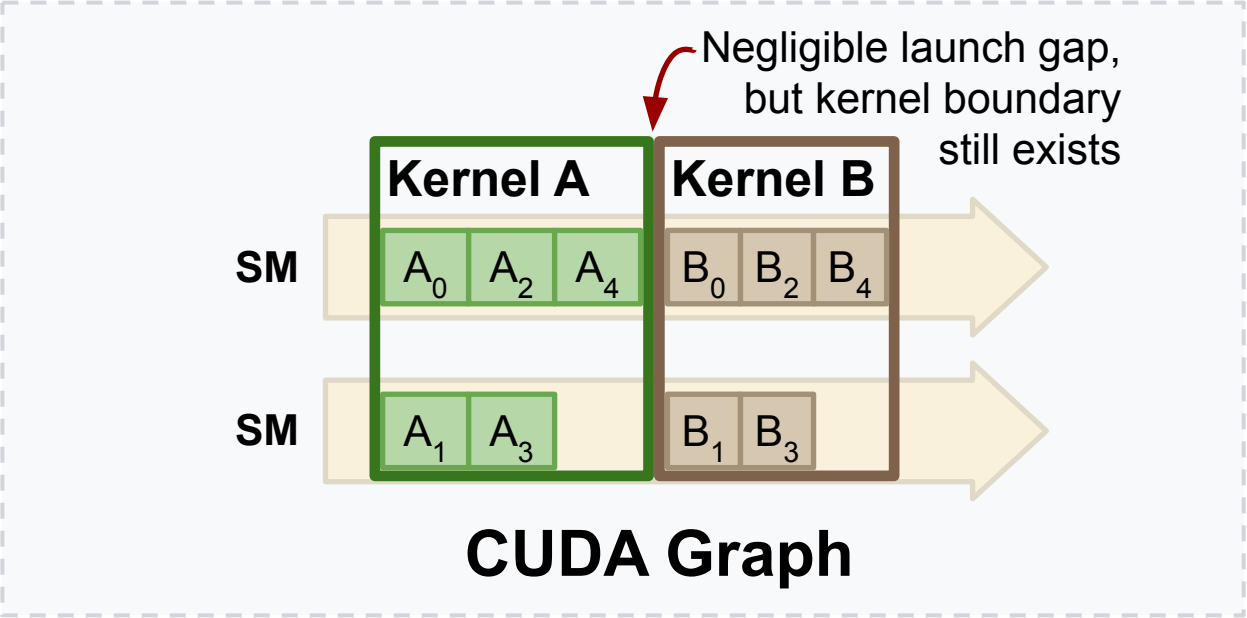
Every SM must finish kernel A before any SM starts kernel B.



Root cause: the GPU's smallest schedulable unit is the kernel, not the tile.

Doesn't CUDA Graph already solve this?

Yes — but only half of it



✓ Fixes the launch gap

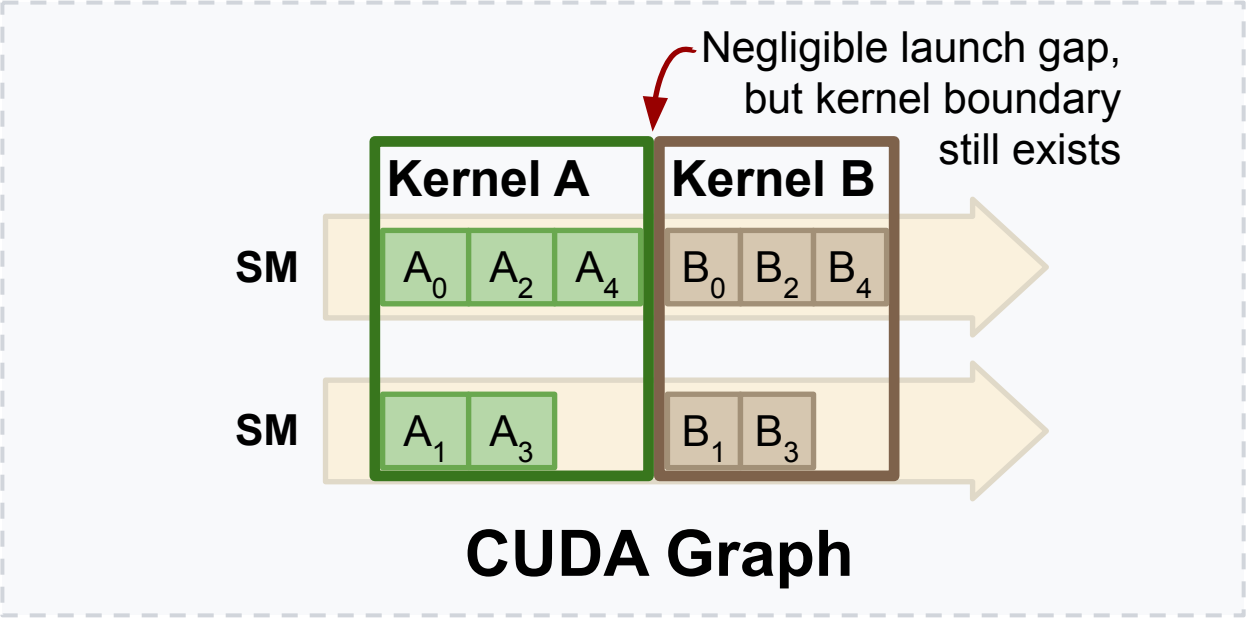
Capture a kernel sequence once, replay it many times — per-launch overhead drops to nearly zero.

✗ Keeps kernel boundaries

Global sync barrier between kernels remains.

Doesn't CUDA Graph already solve this?

Yes — but only half of it



✓ Fixes the launch gap

Capture a kernel sequence once, replay it many times — per-launch overhead drops to nearly zero.

✗ Keeps kernel boundaries

Global sync barrier between kernels remains.

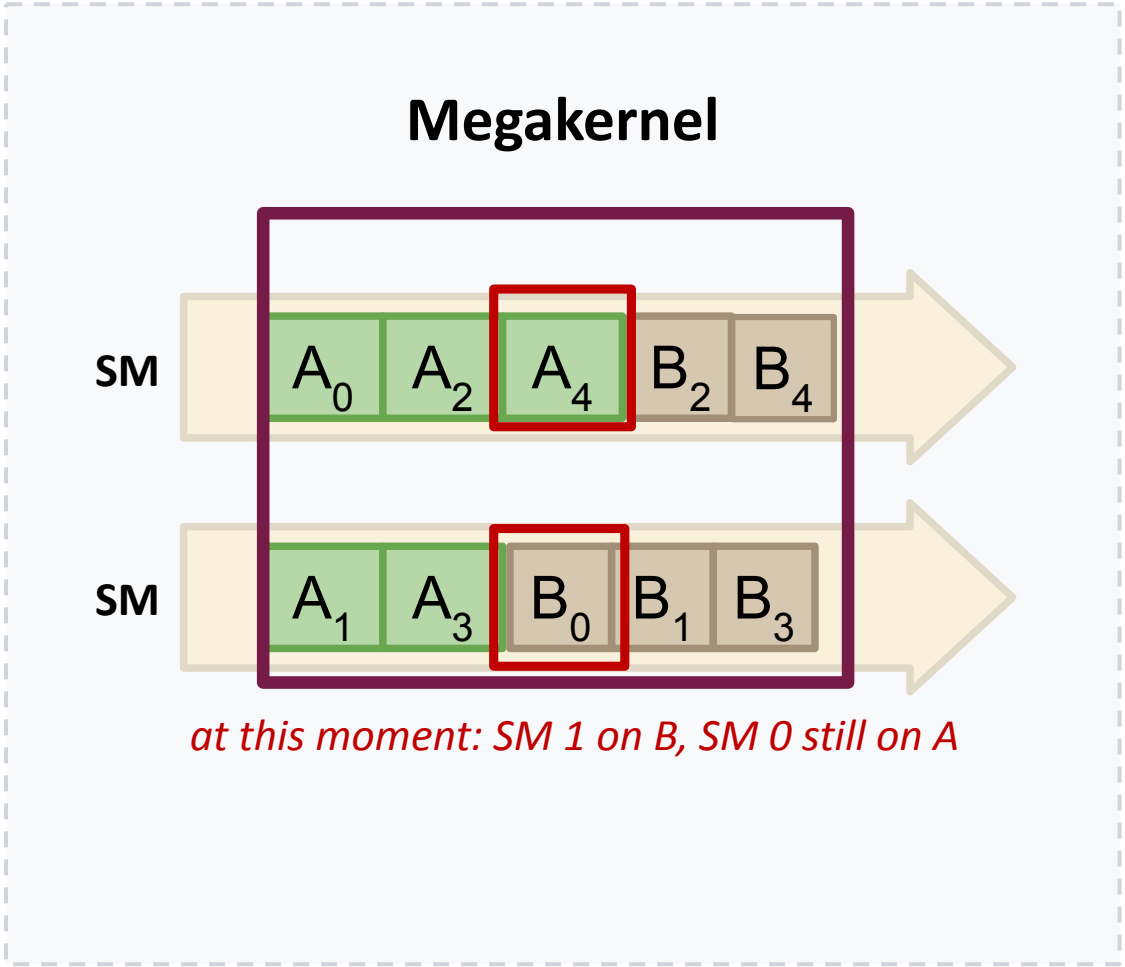
We need to **break kernels apart,** not just replay them.

Megakernel: schedule tiles, not kernels

Drop the kernel boundary.

Let SMs pick up tile-level tasks directly from a global pool.

Once tile 0 of op A is done, tile 0 of op B can start immediately — no waiting for op A to finish across all SMs.



Megakernel: schedule tiles, not kernels

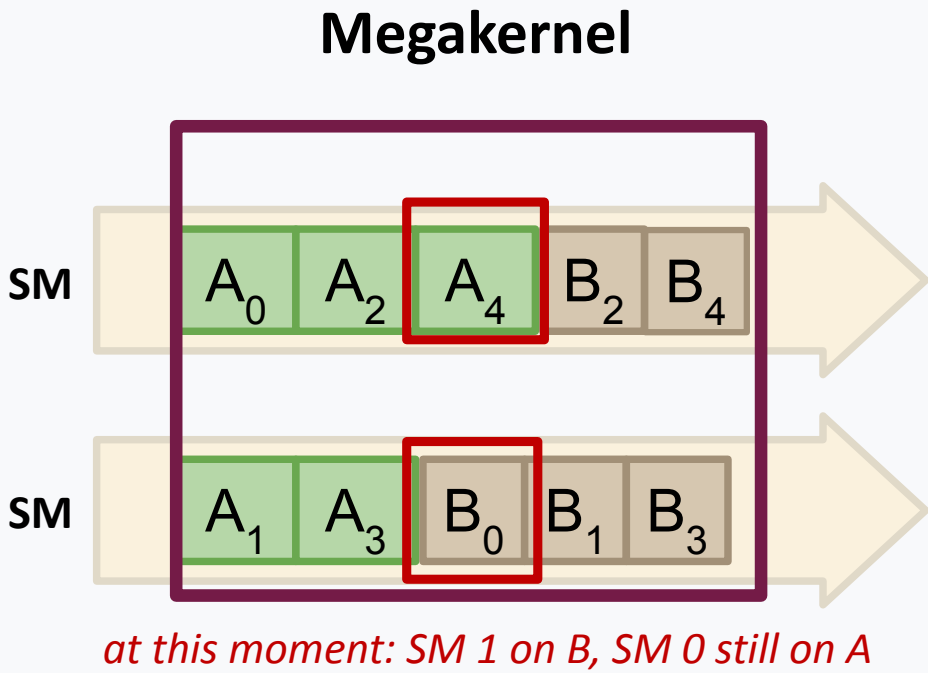
Drop the kernel boundary.

Let SMs pick up tile-level tasks directly from a global pool.

Once tile 0 of op A is done, tile 0 of op B can start immediately — no waiting for op A to finish across all SMs.

✓ **No launch gap**

✓ **No global sync — fine-grained pipelining across operators**



...but megakernels are hard to build

Challenge 1

Dynamism

- **Shape dynamism**

  Continuous batching makes batch size & sequence length change every iteration — recompiling per shape blows up warmup.

- **Data-dependent dynamism**

  MoE routing decides at runtime which tokens flow to which experts. Static task graphs can't express this.

...but megakernels are hard to build

Challenge 1

Dynamism

- **Shape dynamism**
  Continuous batching makes batch size & sequence length change every iteration — recompiling per shape blows up warmup.
- **Data-dependent dynamism**
  MoE routing decides at runtime which tokens flow to which experts. Static task graphs can't express this.

Challenge 2

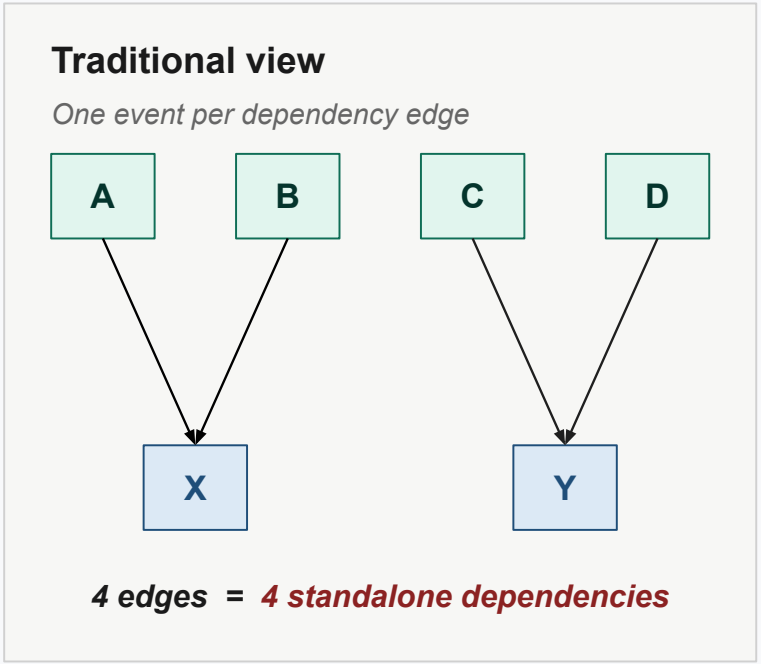
Programmability

- **Manual dependency wiring**
  Hand-coding notify/wait across thousands of tile-level tasks is error-prone and hard to maintain.
- **Multiple scheduling strategies**
  Static vs. dynamic scheduling suit different workloads — but switching today means rewriting the kernel.

We need a single abstraction that handles both.

Event Tensor: events as tensors

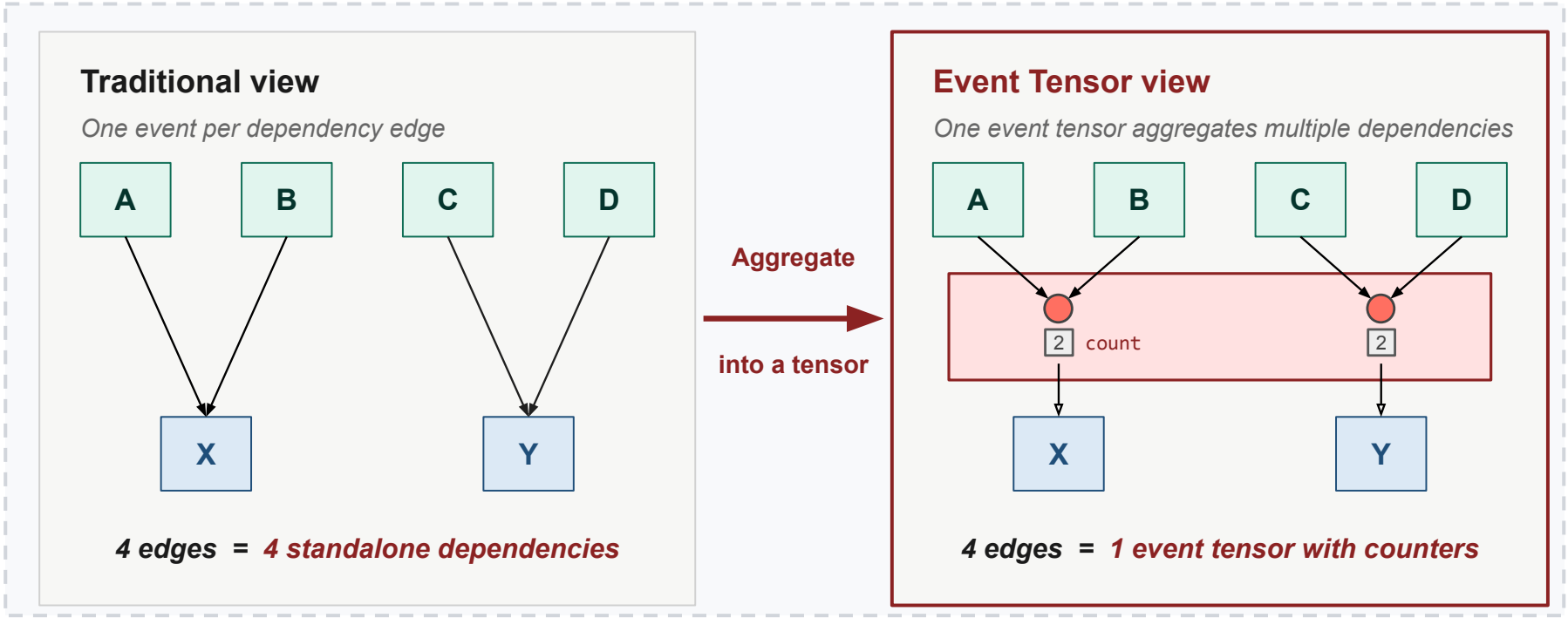
The core insight of the paper



Event Tensor: events as tensors

The core insight of the paper

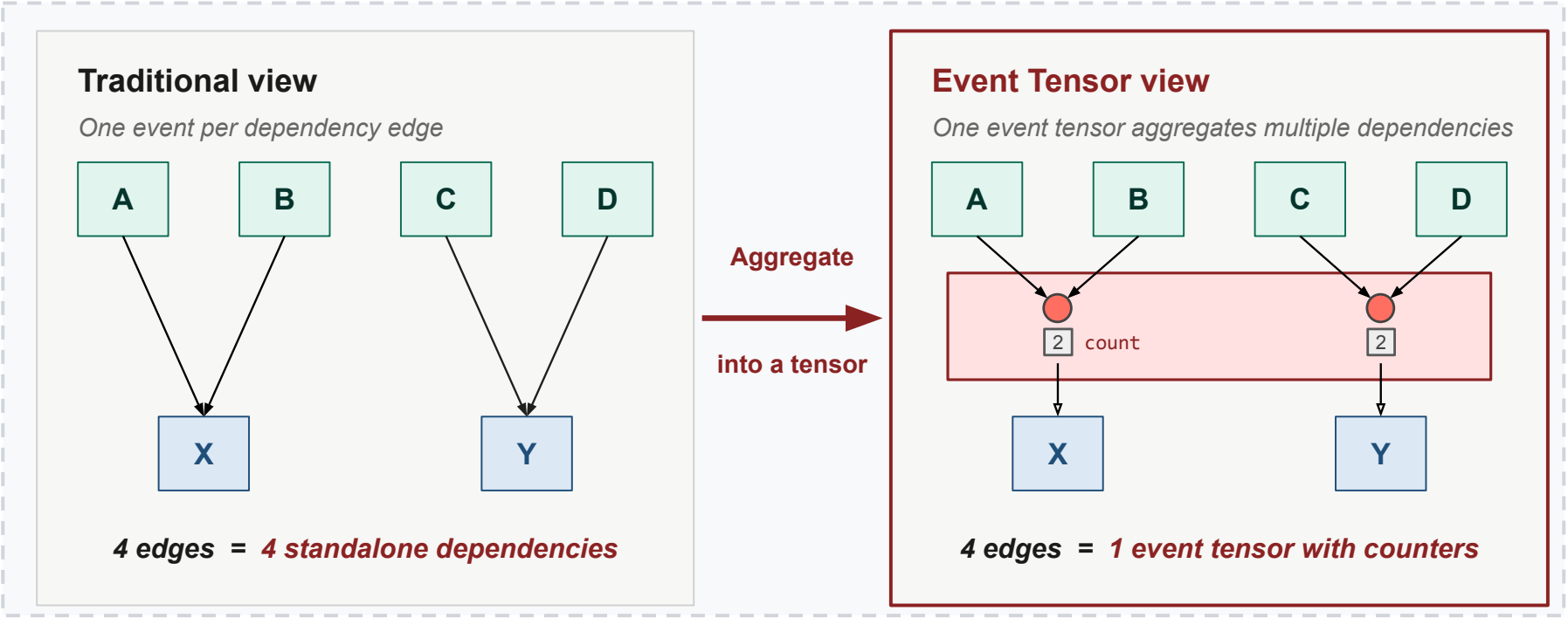
Aggregate many per-edge events into one tensor of events with wait-counts.



Event Tensor: events as tensors

The core insight of the paper

Aggregate many per-edge events into one tensor of events with wait-counts.




```
E = ETensor((2,), wait_count=2)
P: Tensor = call_device(
  producer,
  tile_num=(2, 2),
  out_edges={E: "ij->i"},
)
C: Tensor = call_device(
  consumer,
  tile_num=(2,),
  in_edges={E: "i->i"},
)
```

First-class IR object

Lives in the compiler IR like any tensor

Symbolic shape support

Inherits dynamic-shape machinery for free

Index expressions

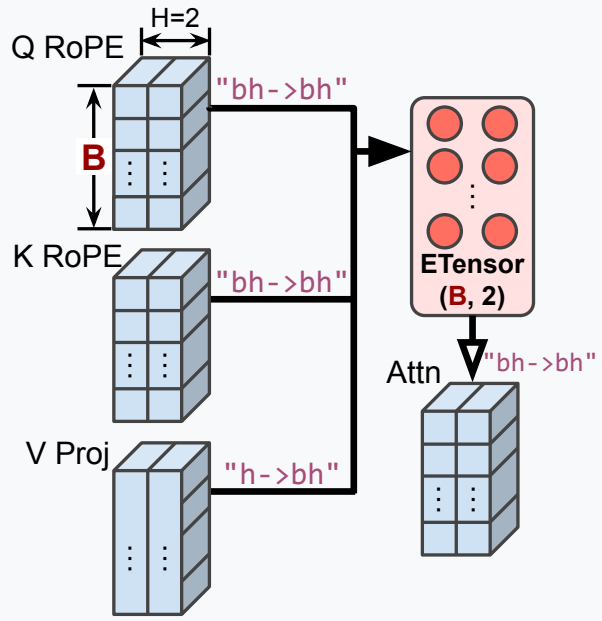
Map task coords ⟶ event coords for data dep.

Shape dynamism: one kernel, any batch size

Symbolic Event Tensor as template ⟶ materialized at runtime

`ETensor((B, 2))`

B stay symbolic in the compiler IR.



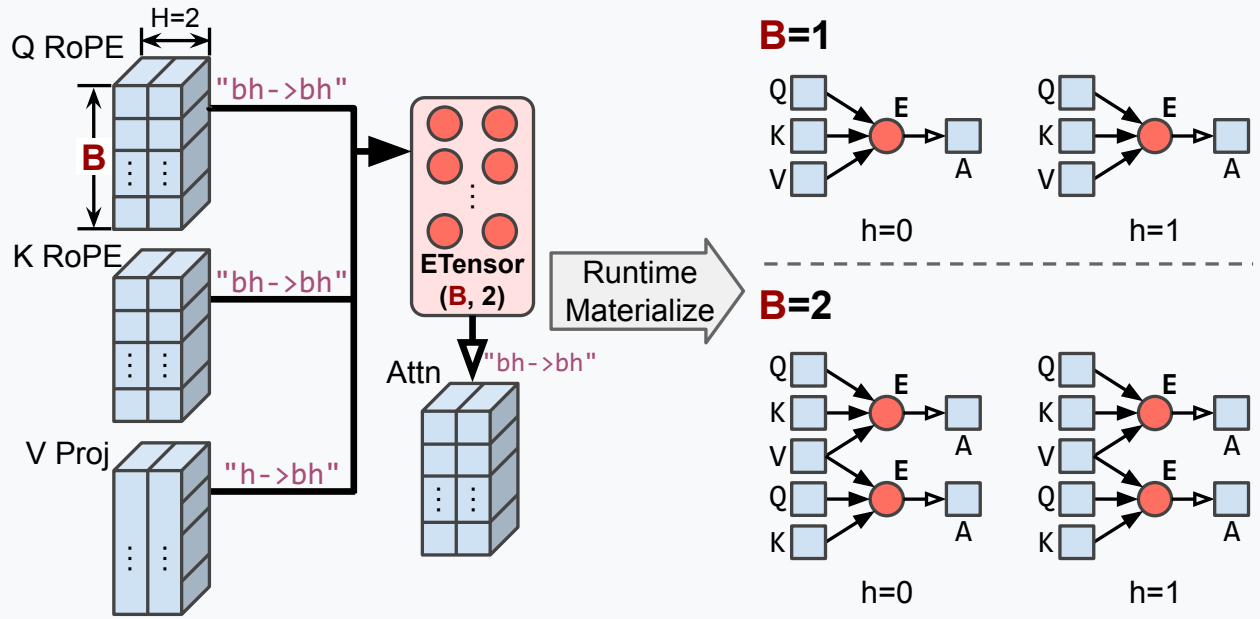
Shape dynamism: one kernel, any batch size

Symbolic Event Tensor as template → materialized at runtime

`ETensor((B, 2))`

B stay symbolic in the compiler IR.

At runtime, the same compiled kernel instantiates a different graph for B=1, B=2, …



Shape dynamism: one kernel, any batch size

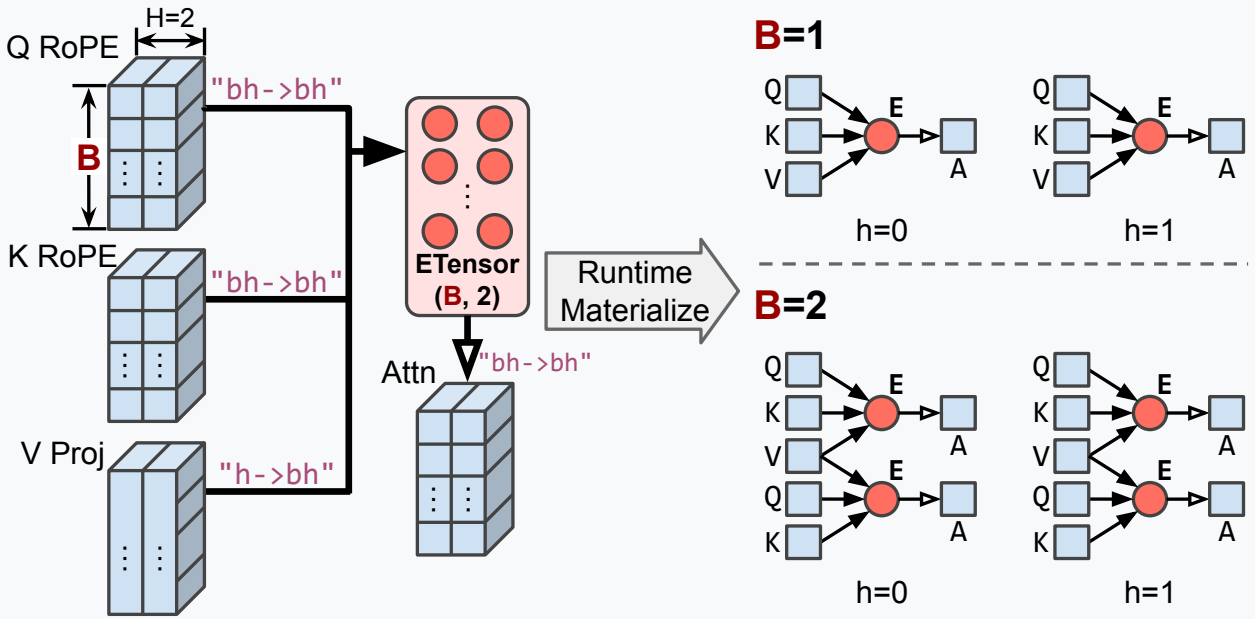
Symbolic Event Tensor as template → materialized at runtime

`ETensor((B, 2))`

B stay symbolic in the compiler IR.

At runtime, the same compiled kernel instantiates a different graph for B=1, B=2, …

Result: AOT-compile once, run on any shape — **no JIT, no CUDA Graph recapture.**



Data-dependent dynamism: the MoE case

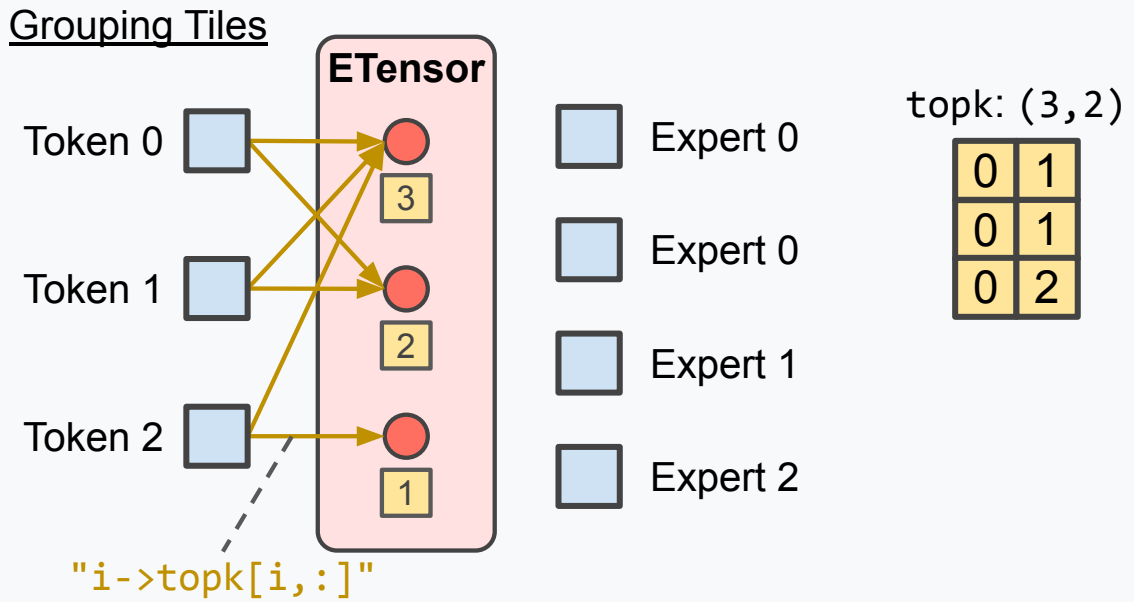
Dependencies themselves depend on runtime values

Mechanism 1

Data-dependent update

Runtime tensor topk decides which event each grouping tile notifies.

```
out_edges = { E: "i → topk[i, :]" }
```



Data-dependent dynamism: the MoE case

Dependencies themselves depend on runtime values

Mechanism 1

Data-dependent update

Runtime tensor topk decides which event each grouping tile notifies.

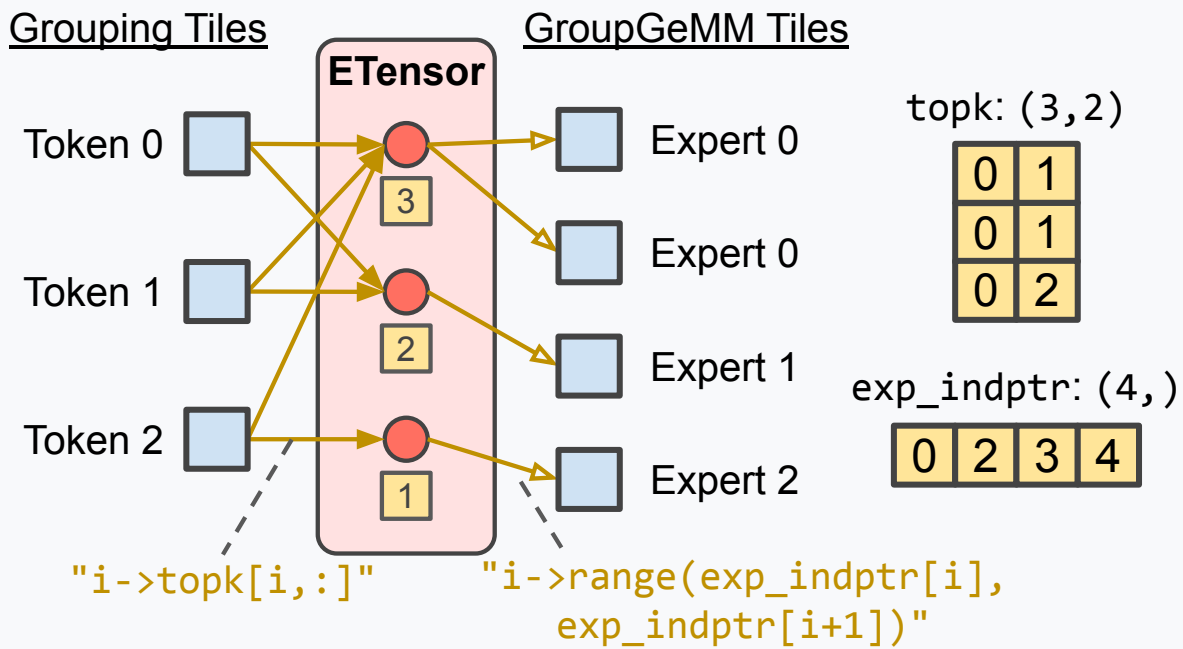
```
out_edges = { E: "i → topk[i, :]" }
```

Mechanism 2

Data-dependent trigger

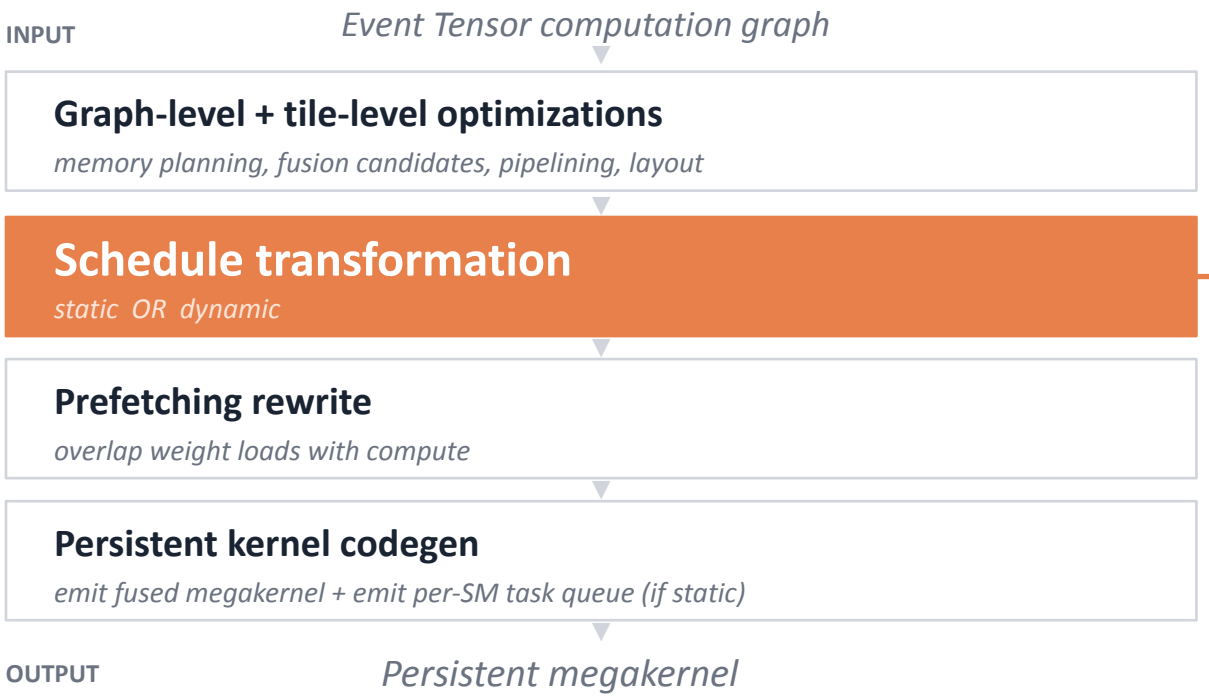
Runtime tensor exp_indptr decides how many GroupGEMM tiles each event triggers.

```
tiles ∈ [exp_indptr[i], exp_indptr[i+1])
```



ETC compiler: from graph to megakernel

A pipeline of compiler passes lowers Event Tensor graphs into persistent kernels



The key pass

Same Event Tensor IR **generates both** scheduling strategies — pick per workload, **no rewrite.**

Static scheduling: pre-assigned queues

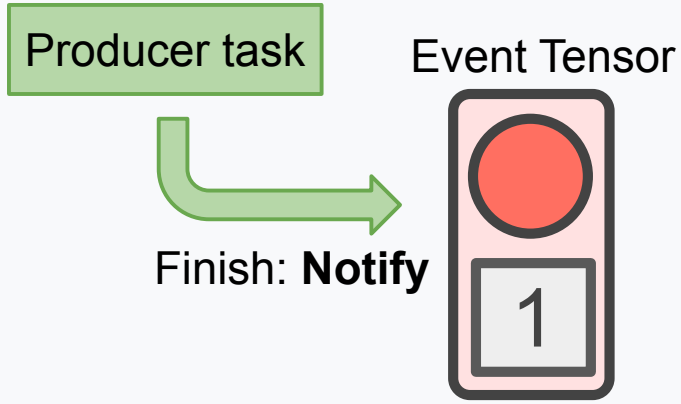
Compile-time SM assignment, sync via counter-based semaphores

Every task is assigned to a specific SM at compile time.
Each SM runs its own pre-computed queue. Sync between dependent tasks uses atomic counters: `notify()` decrements, `wait()` spin-loops until the counter hits zero.

Static scheduling: pre-assigned queues

Compile-time SM assignment, sync via counter-based semaphores

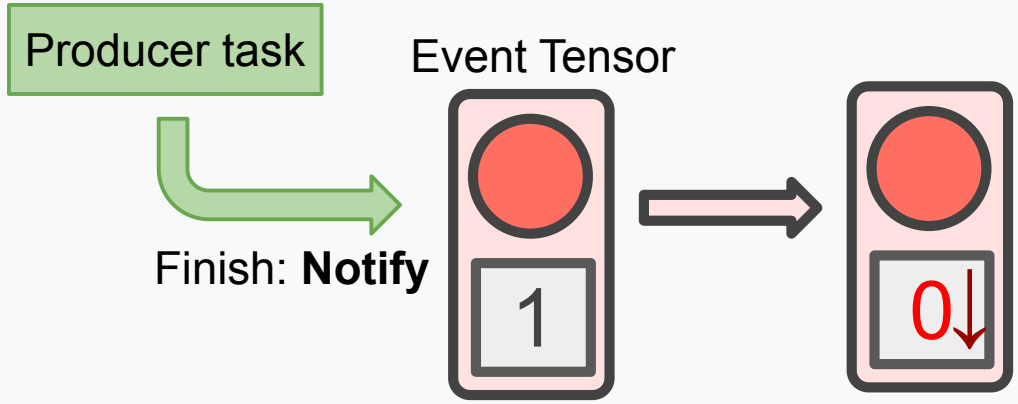
Every task is assigned to a specific SM at compile time.
Each SM runs its own pre-computed queue. Sync between dependent tasks uses atomic counters: `notify()` decrements, `wait()` spin-loops until the counter hits zero.



Static scheduling: pre-assigned queues

Compile-time SM assignment, sync via counter-based semaphores

Every task is assigned to a specific SM at compile time.
Each SM runs its own pre-computed queue. Sync between dependent tasks uses atomic counters: `notify()` decrements, `wait()` spin-loops until the counter hits zero.

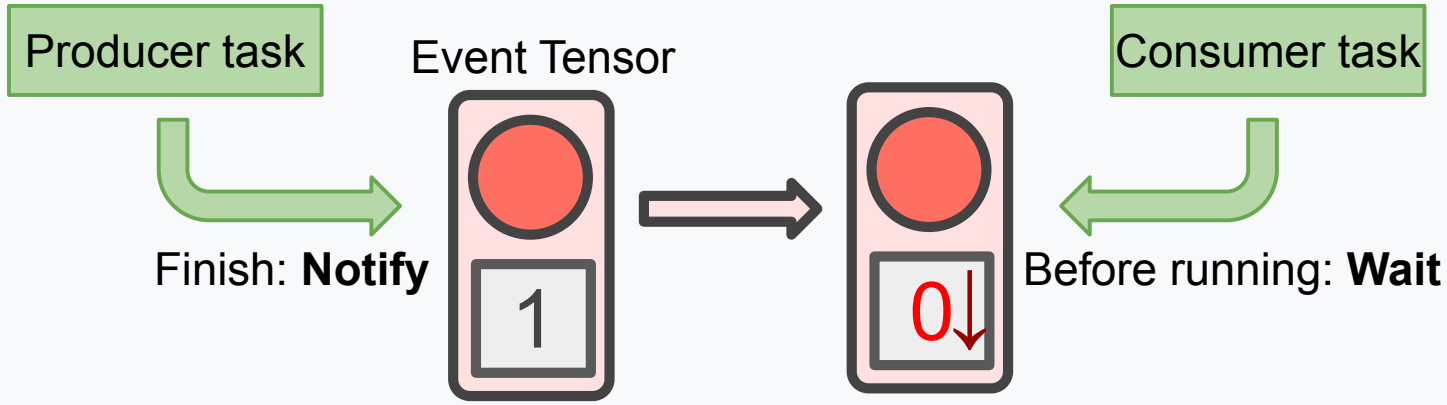


*Best for **predictable workloads** — near-zero scheduling overhead, but no load balancing.*

Static scheduling: pre-assigned queues

Compile-time SM assignment, sync via counter-based semaphores

Every task is assigned to a specific SM at compile time.
Each SM runs its own pre-computed queue. Sync between dependent tasks uses atomic counters: `notify()` decrements, `wait()` spin-loops until the counter hits zero.



*Best for **predictable workloads** — near-zero scheduling overhead, but no load balancing.*

Dynamic scheduling: on-GPU scheduler

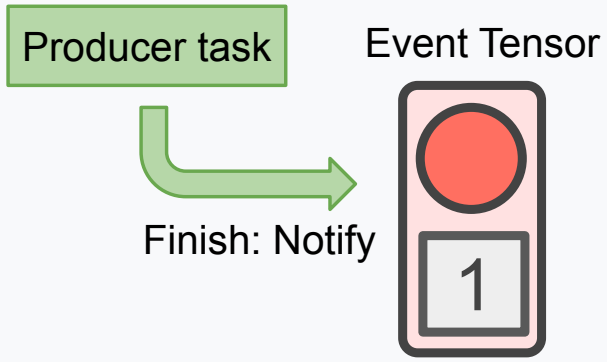
Counter triggers push; idle SMs pop from a shared ready queue

Tasks are not pre-assigned. A lightweight on-GPU scheduler dispatches them.
When an event's counter hits zero, all its consumer tasks are atomically `push()`ed to a global ready queue. Any idle SM `pop()`s a ready task and runs it.

Dynamic scheduling: on-GPU scheduler

Counter triggers push; idle SMs pop from a shared ready queue

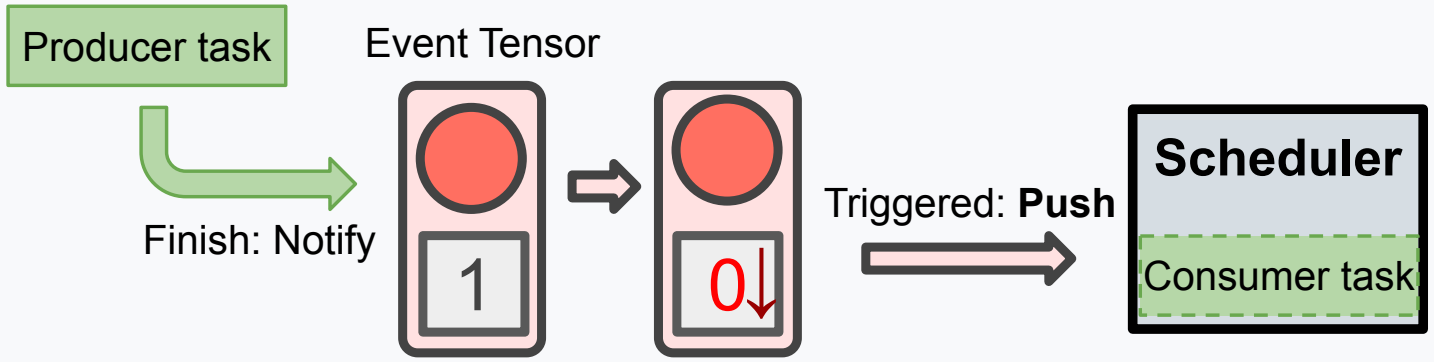
Tasks are not pre-assigned. A lightweight on-GPU scheduler dispatches them.
When an event's counter hits zero, all its consumer tasks are atomically `push()`ed to a global ready queue. Any idle SM `pop()`s a ready task and runs it.



Dynamic scheduling: on-GPU scheduler

Counter triggers push; idle SMs pop from a shared ready queue

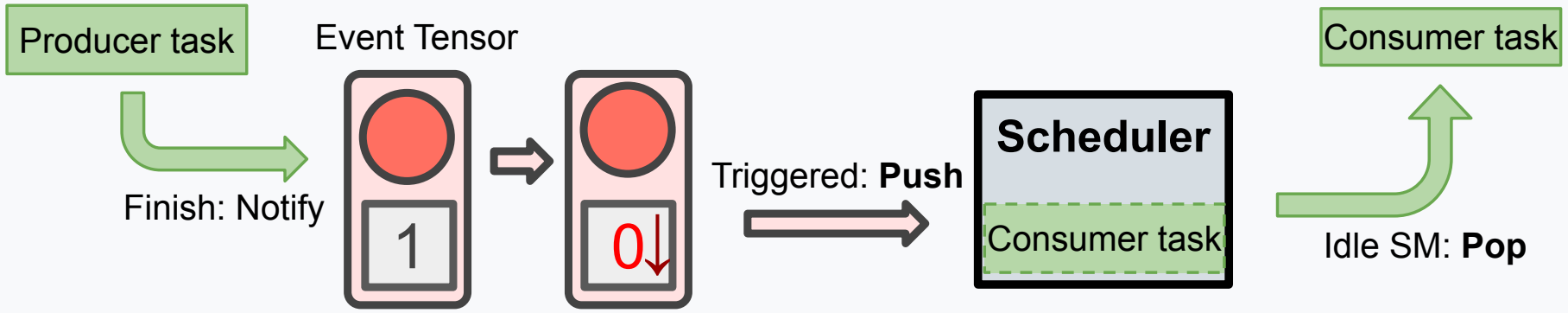
Tasks are not pre-assigned. A lightweight on-GPU scheduler dispatches them.
When an event's counter hits zero, all its consumer tasks are atomically `push()`ed to a global ready queue. Any idle SM `pop()`s a ready task and runs it.



Dynamic scheduling: on-GPU scheduler

Counter triggers push; idle SMs pop from a shared ready queue

Tasks are not pre-assigned. A lightweight on-GPU scheduler dispatches them.
When an event's counter hits zero, all its consumer tasks are atomically `push()`ed to a global ready queue. Any idle SM `pop()`s a ready task and runs it.



*Best for **irregular / data-dependent workloads** — adaptive load balancing at the cost of push/pop overhead.*

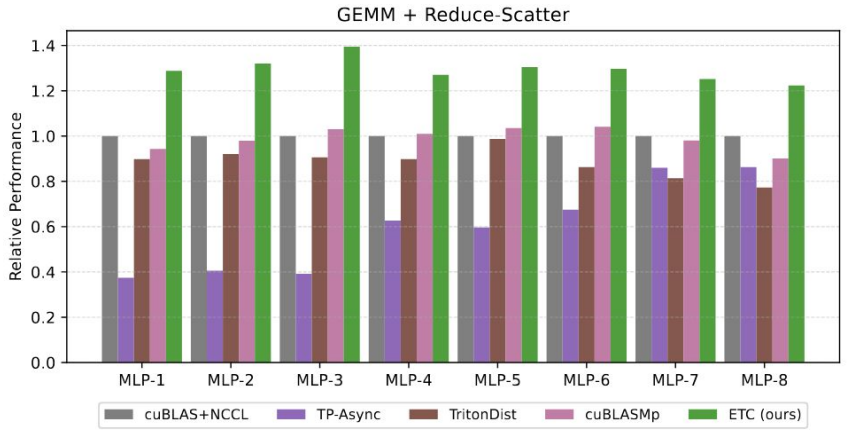
Subgraph evaluations

Validating Event Tensors on static and dynamic task graphs

STATIC TASK GRAPHS

TP comm + GEMM fusion

GEMM+RS on 8 B200s

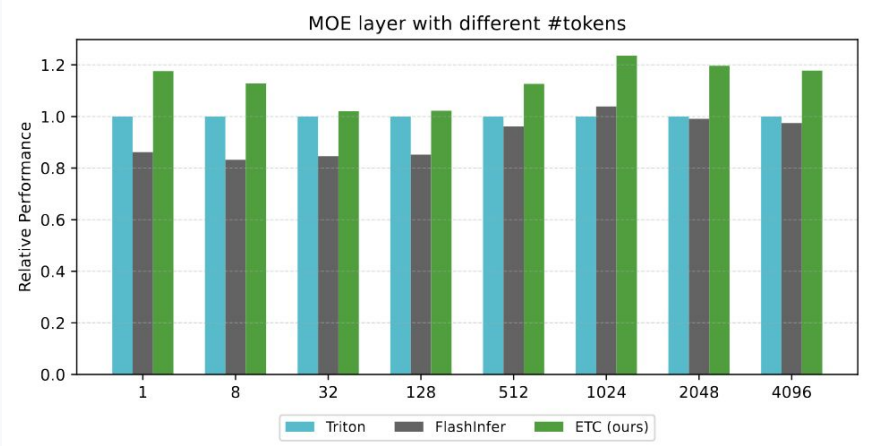


up to 1.40× vs cuBLAS + NCCL & fused baselines

DYNAMIC TASK GRAPHS

Mixture-of-Experts layer

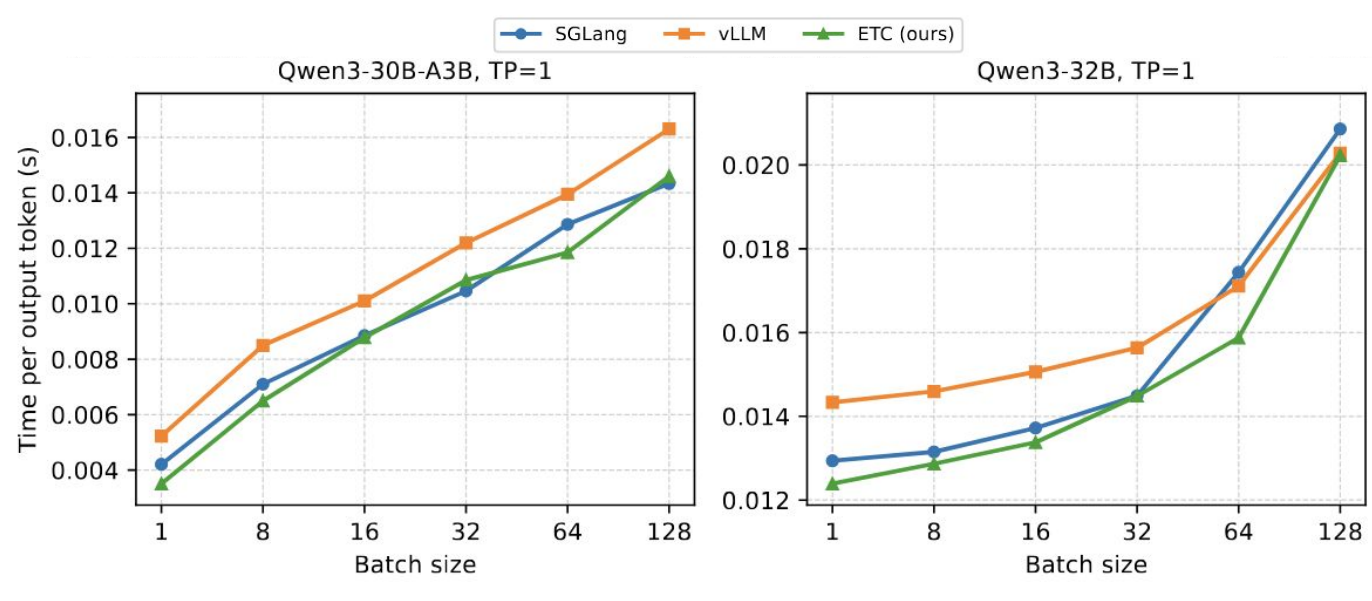
Qwen3-30B-A3B · 128 experts, top-8



up to 1.23× vs Triton & FlashInfer

Low-batch end-to-end serving on B200

Decoding latency + engine warmup vs vLLM and SGLang



DECODE LATENCY

Matches or outperforms SOTA serving engines (SGLang, vLLM)

ENGINE WARMUP

16× faster vs SGLang

— AOT-compiled, zero graph capture.

SGLang	583 s	51 graphs
vLLM	123 s	67 graphs
ETC (AOT)	**35 s**	**0 graphs**

Where does the speedup come from?

Four mechanisms unlocked by Event Tensors + megakernel compilation

1 Parallel execution across ops

Independent operators run concurrently — e.g., Q's Norm+RoPE in parallel with K's Norm+RoPE+CacheAppend and V's CacheAppend inside attention.

2 Reduced wave quantization

Pipelining GroupGEMMs in MoE and GEMMs in MLP smooths SM allocation — no half-empty wave at each kernel's tail.

3 Weight prefetching

Each tile prefetches its weights before activations are ready — hiding memory latency under the previous op's compute.

4 On-chip load balancing

Dynamic scheduler keeps SMs busy under irregular routing (e.g., MoE token dispatch) — no straggler SMs.

Conclusions

1. **Event Tensor Abstraction**

   Unified, first-class support for dynamic shapes and data-dependent computation.

2. **Event Tensor Compiler**

   Use static/dynamic scheduling to generate high-performance kernels automatically

3. **Achieved state-of-the-art serving latency while substantially reducing warmup overhead.**

Future work: Agentic Megakernel Generation

Thank you · Questions?