

# **GPU Architecture and Kernel Programming**

CSE 291A · October 6, 2026

Yiying Zhang

# From Inference to GPU Execution

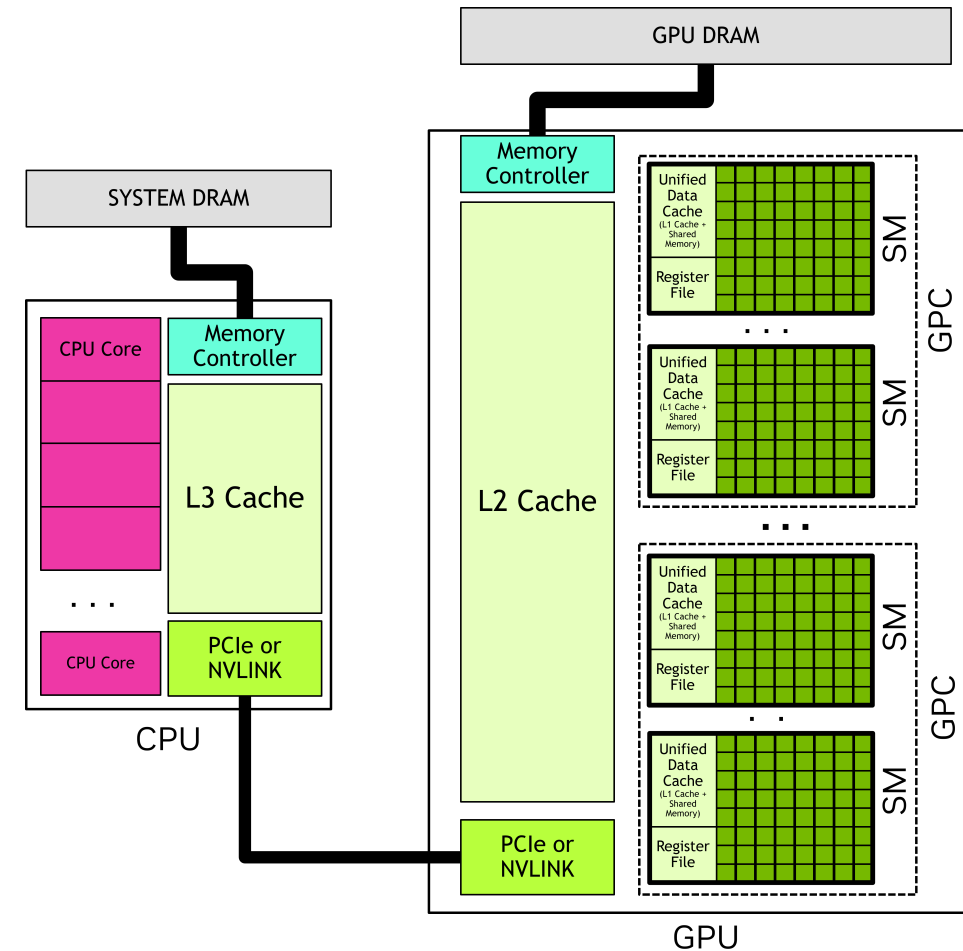
- GPU execution and memory hierarchy
- FlashAttention: tiling attention to reduce memory traffic
- From PyTorch to kernels: the programming stack
- ThunderKittens and megakernels
- Project connection: profiling GPU inference

# CPU Host and GPU Device

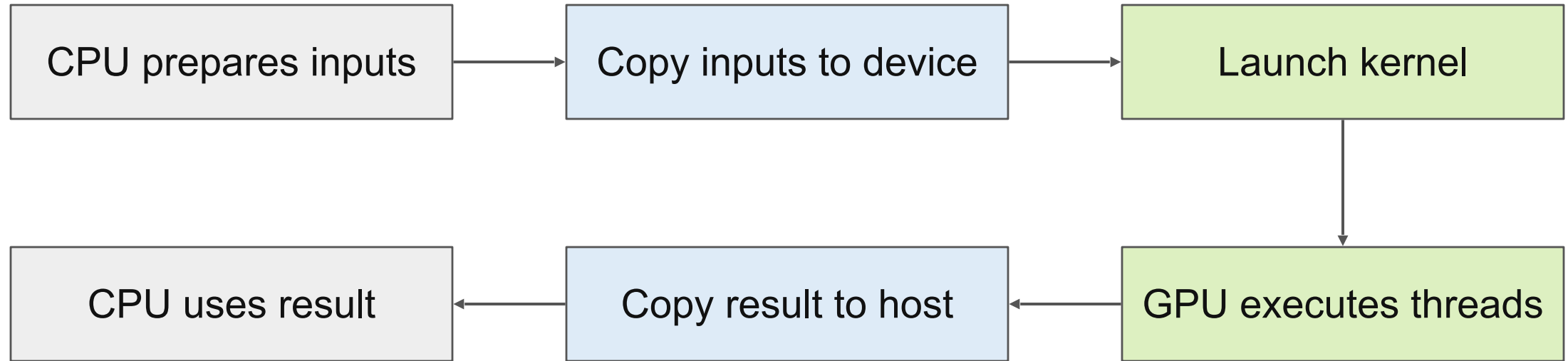
The CPU runs the application and launches GPU kernels.

The GPU runs many threads across streaming multiprocessors.

Data placement and transfers are part of the program.



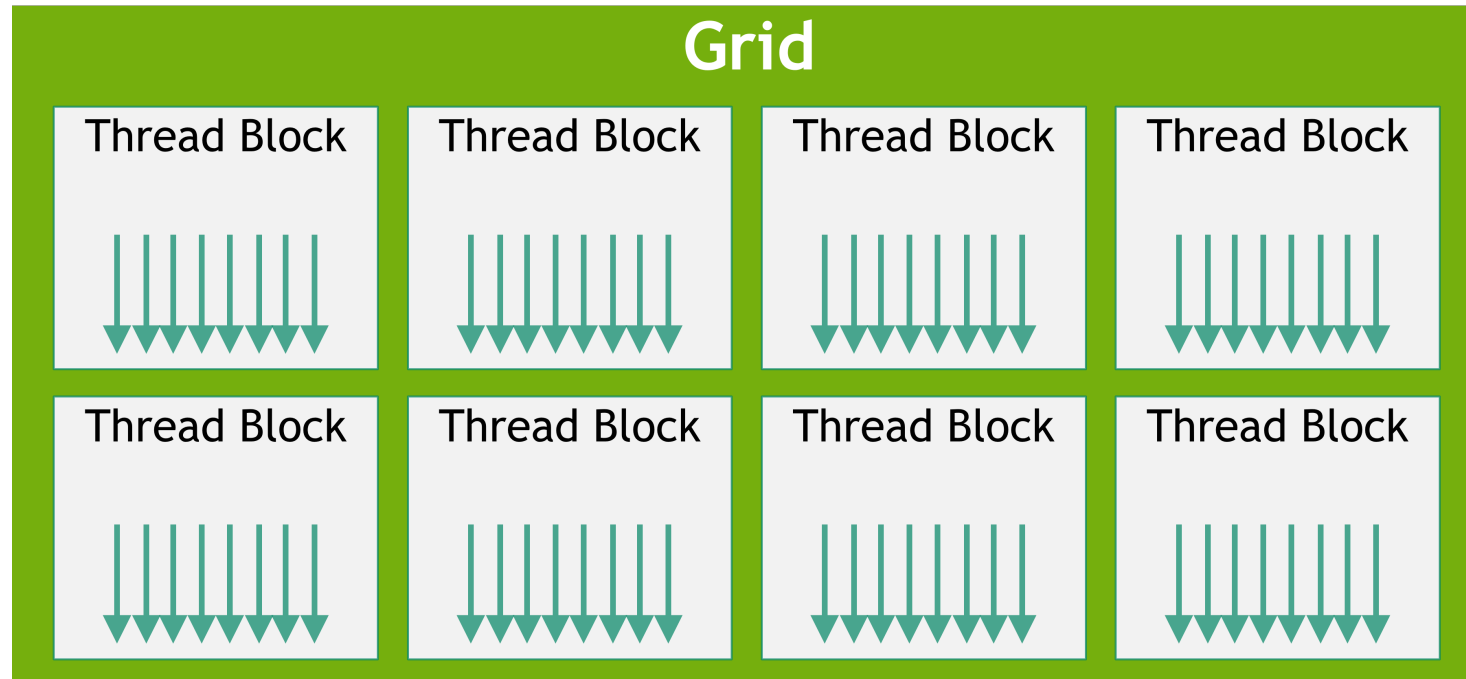
# A GPU Operation: Copy, Launch, Use



A launch is asynchronous: the CPU can continue while the GPU works.

Keep intermediate tensors on the GPU to avoid repeated transfers.

# Threads, Blocks, and a Grid



Kernel launch: `add<<<number_of_blocks, threads_per_block>>>(...)`

`threadIdx` identifies a thread within its block; `blockIdx` identifies the block.

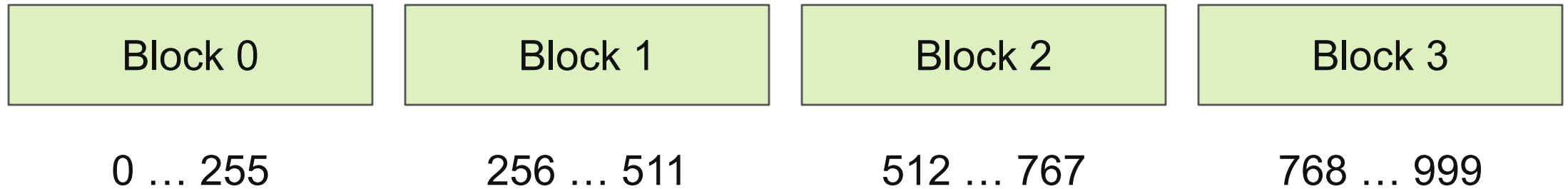
# Vector Addition: One Element per Thread

```
__global__ void add(const float* a, const float* b,  
                   float* c, int n) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < n) c[i] = a[i] + b[i];  
}  
  
int threads = 256;  
int blocks = (n + threads - 1) / threads;  
add<<<blocks, threads>>>(a, b, c, n);
```

The same kernel code runs in every thread, with a different index  $i$ .

# Mapping Threads to Array Elements

Example:  $n = 1,000$  elements, 256 threads per block



$$i = \text{blockIdx.x} \times 256 + \text{threadIdx.x}$$

$4 \times 256 = 1,024$  threads. The last 24 fail the  $i < n$  guard.

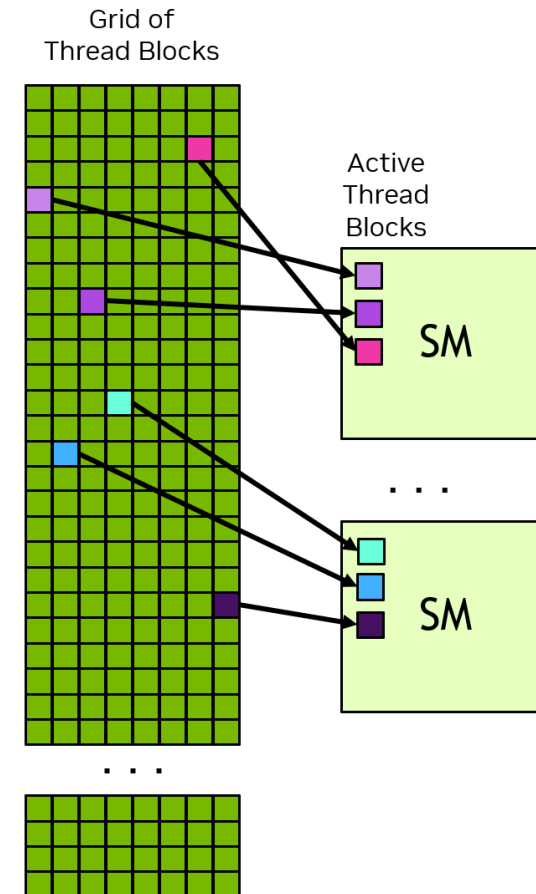
# How Blocks Reach the Hardware

One block executes on one SM.

An SM can hold several blocks.

Registers and shared memory limit how many blocks fit.

A grid can contain far more blocks than there are SMs.



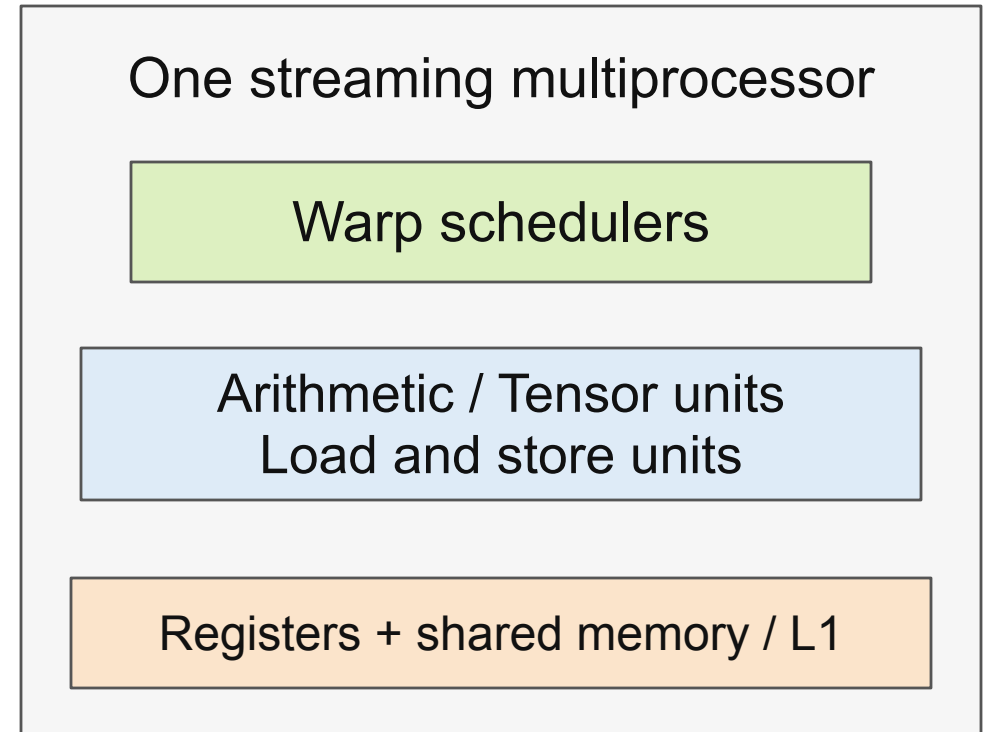
# What Is an SM?

SM = Streaming Multiprocessor

A hardware execution engine inside the GPU.

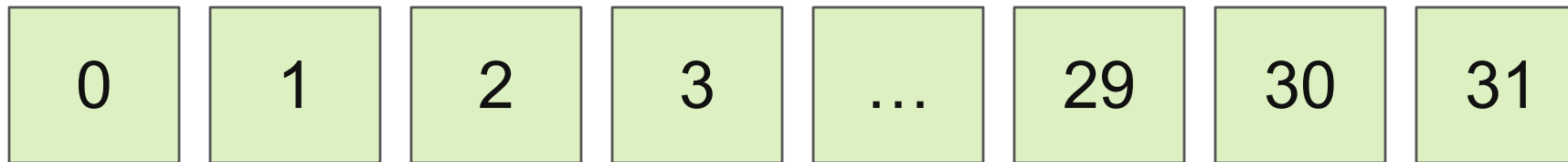
Runs resident thread blocks by scheduling their warps.

Contains compute units and fast on-chip storage.



# What Is a Warp?

A warp groups 32 threads from the same thread block.



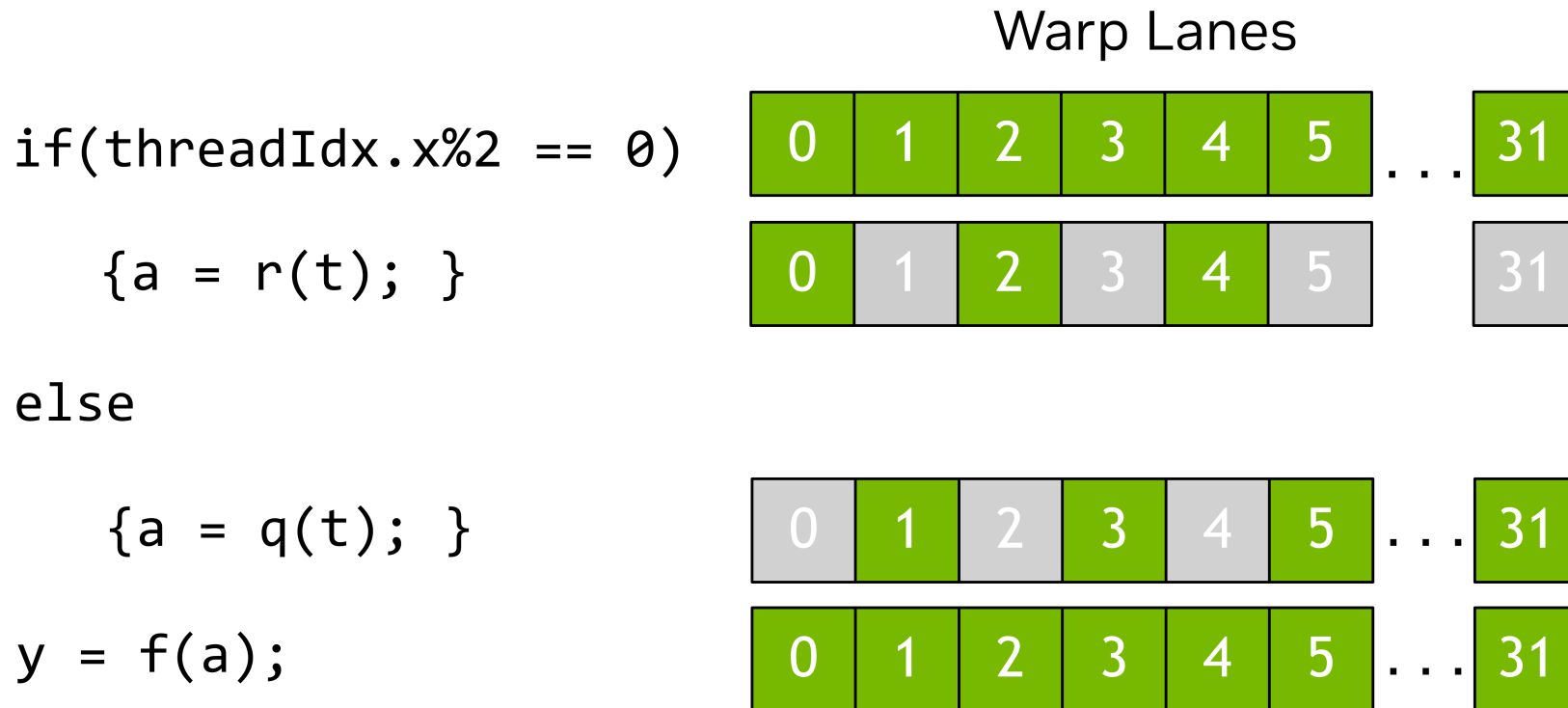
Lane IDs within one warp

The SM schedules instructions for ready warps.

128 threads per block = 4 warps per block.

Threads keep their own registers and may take different branches.

# Warps and Divergent Branches

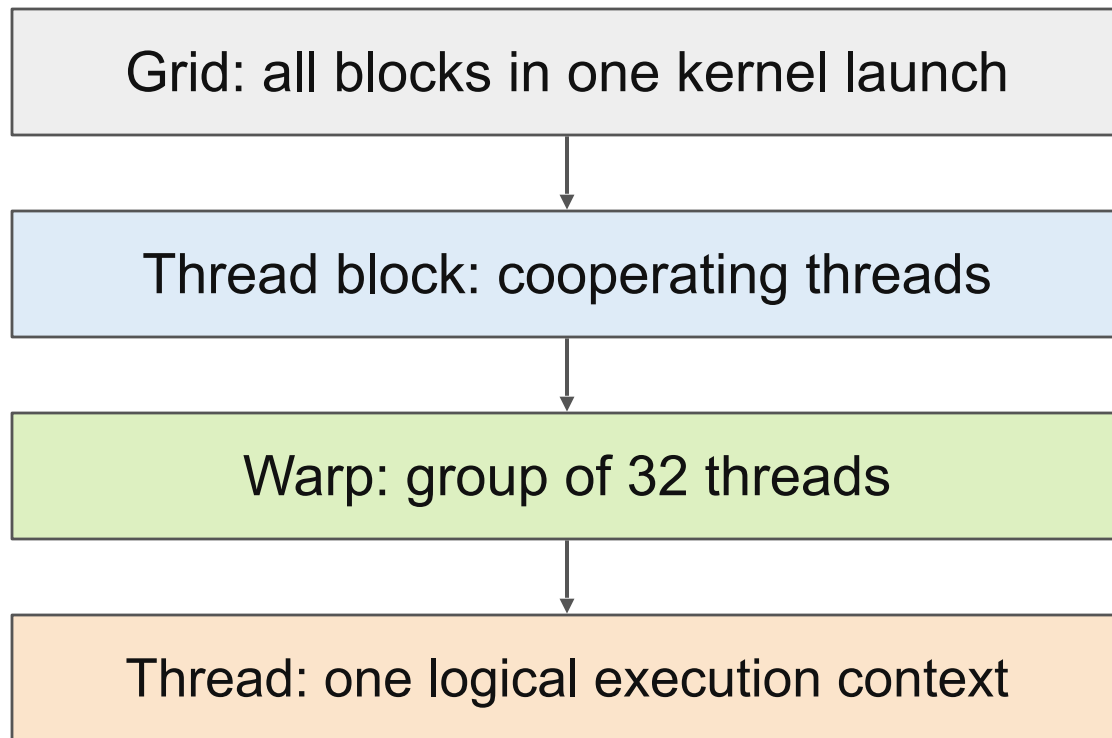


The SM switches among ready warps to keep execution units busy.

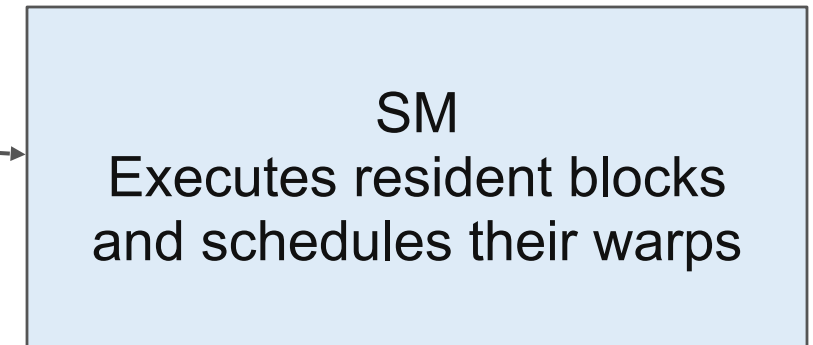
**A warp is not part of CUDA Programming Interface, but is an important CUDA implementation detail on modern NVIDIA GPUs**

# Grid, Block, Warp, Thread, and SM

## Logical organization



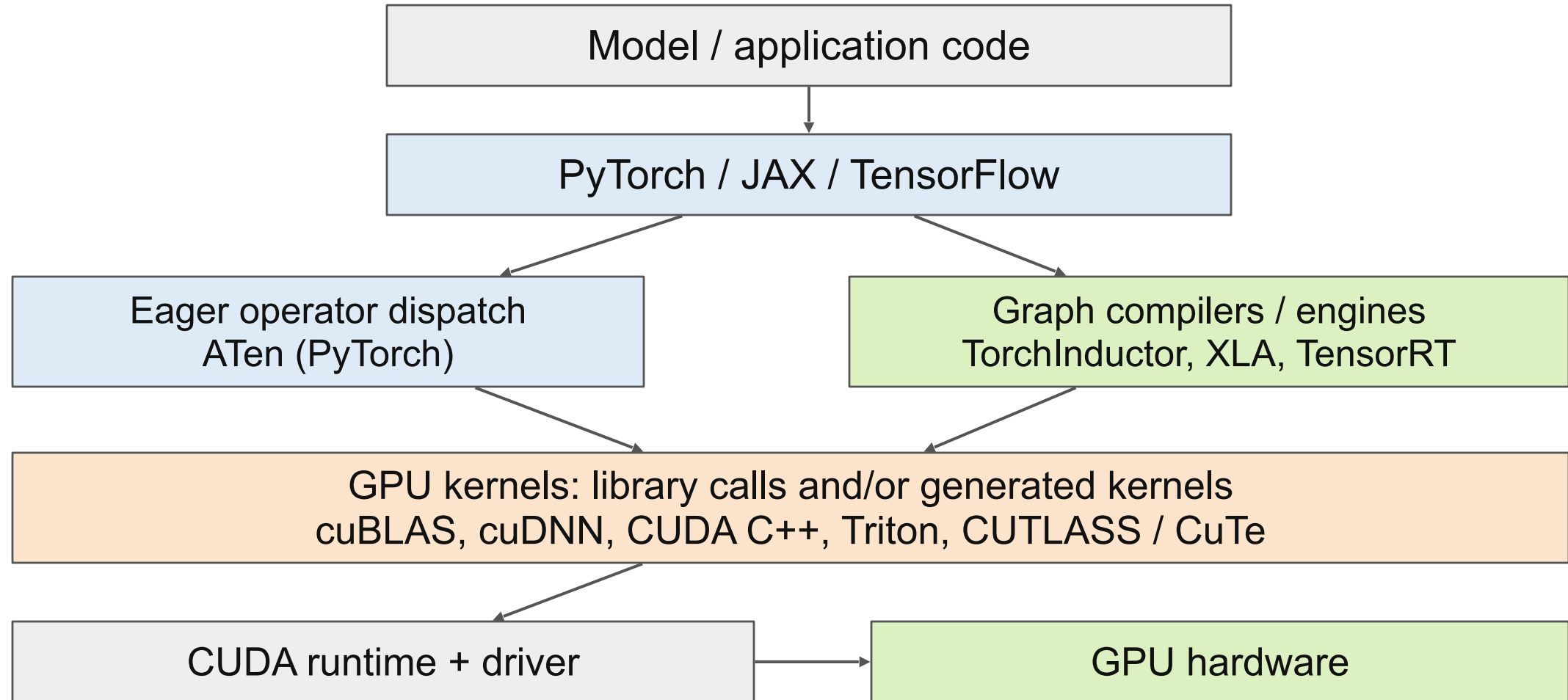
## Hardware execution



One block stays on one SM.  
An SM can host several blocks.

256 threads/block = 8 warps/block. More blocks than SMs can run in waves.

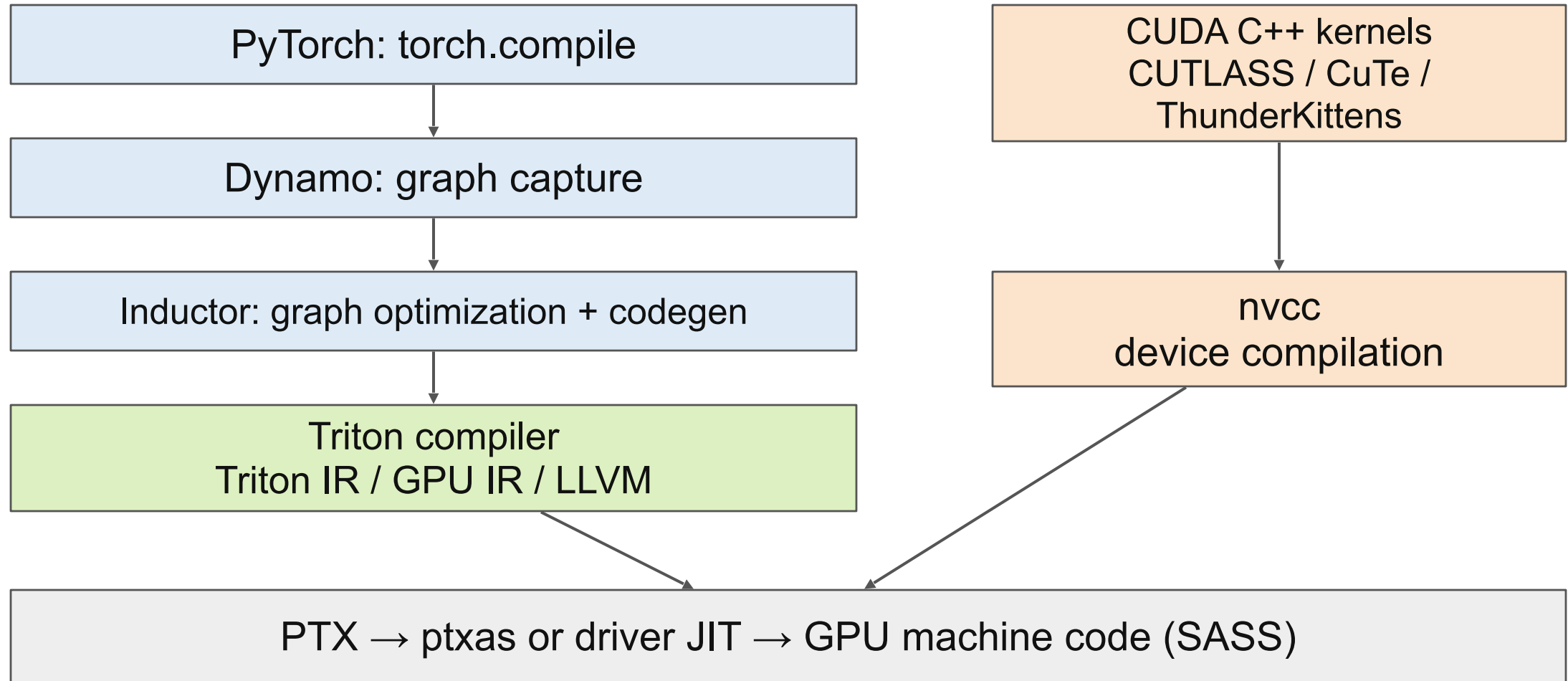
# GPU Software Ecosystem



Across layers: Nsight profiling

Between GPUs: NCCL communication

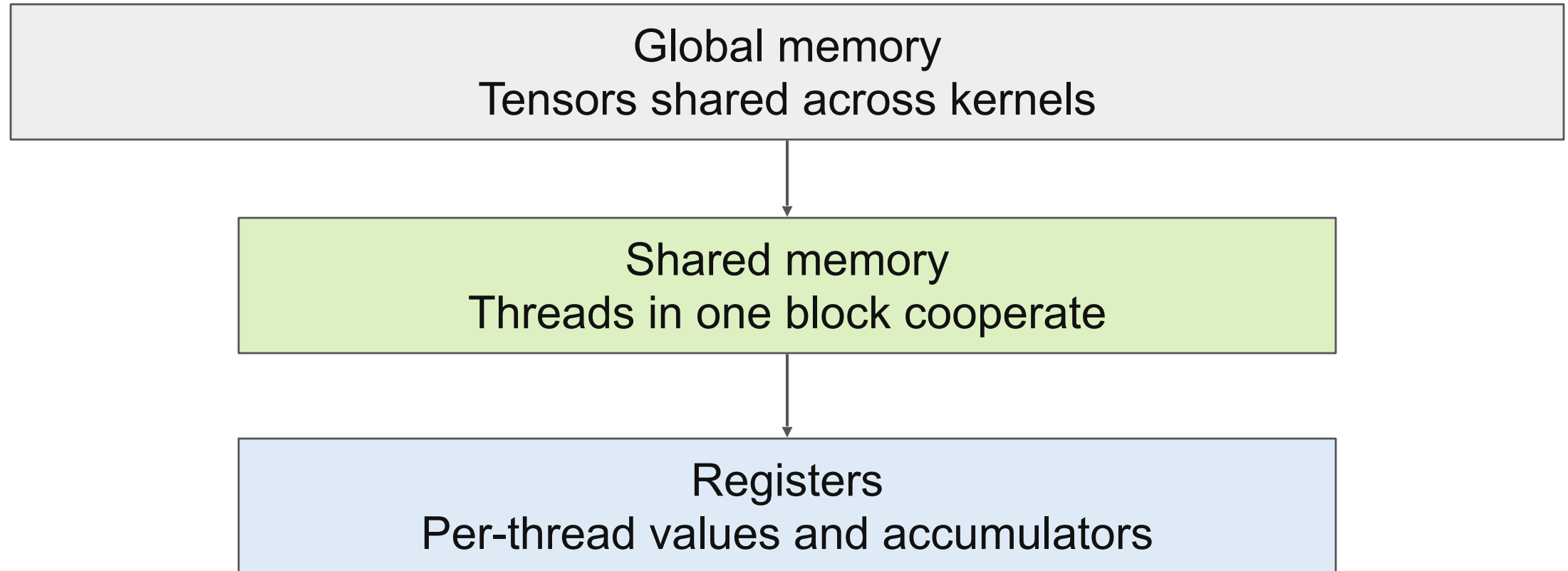
# Two Paths to GPU Machine Code



# GPU Execution Model - Summary

- Many execution units, organized into streaming multiprocessors (SMs)
- Each thread block maps to one SM
- Threads execute in groups called warps
- Residency and parallel execution depend on resources and scheduling

# Memory and Cooperation Within a Block



Example: load a tile, synchronize the block, then reuse the tile.

# CUDA Memory Hierarchy

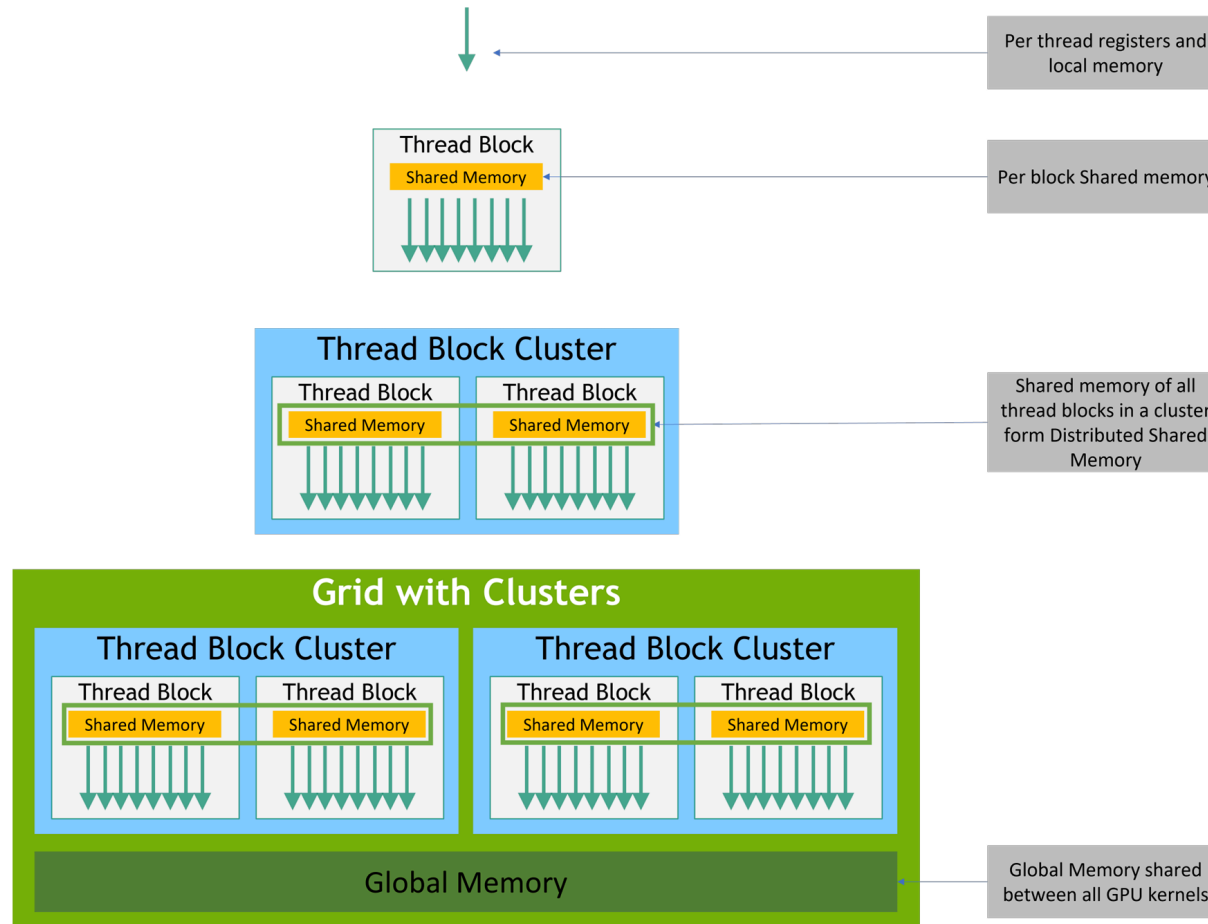


Figure 6: Memory Hierarchy

Thread: registers and thread-private local memory.

Block: shared memory on its SM.

Cluster (optional): access other blocks' shared memory.

Global: device HBM accessible across kernels.

**“Local” means thread-private. Local memory is in device memory, not registers.**

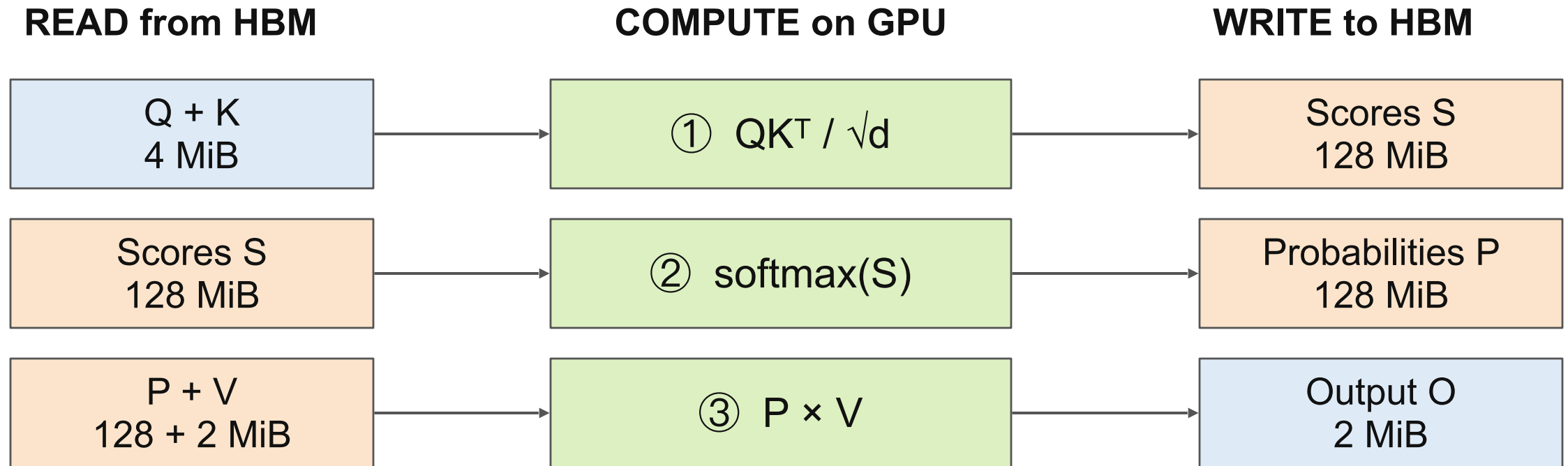
# Memory Hierarchy Sets the Cost

- HBM: large capacity, but every transfer costs time
- Shared memory and registers: small, fast, and local to an SM
- Tensor cores accelerate matrix tiles when operands are ready
- More arithmetic can be faster if it avoids memory traffic

# Why Attention Moves So Much Data

Naïve attention: three separate kernels · one head · FP16

$N = 8,192$  tokens,  $d = 128 \rightarrow N \times d = 2 \text{ MiB}$ ;  $N \times N = 128 \text{ MiB}$



**S and P alone:  $4 \times 128 \text{ MiB} = 512 \text{ MiB}$  moved (write + read each)**

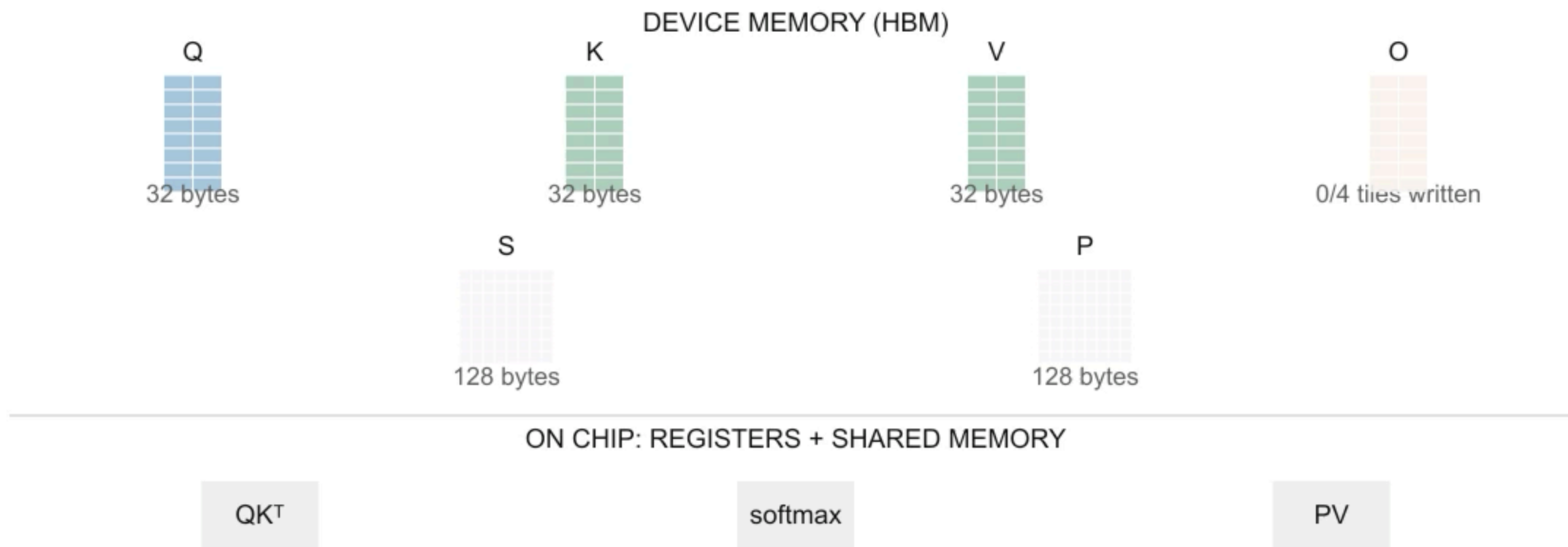
Idealized tensor traffic; cache effects and internal rereads omitted. Total: 520 MiB.

Source: <https://arxiv.org/abs/2205.14135>

# Naïve Attention: Memory Movement

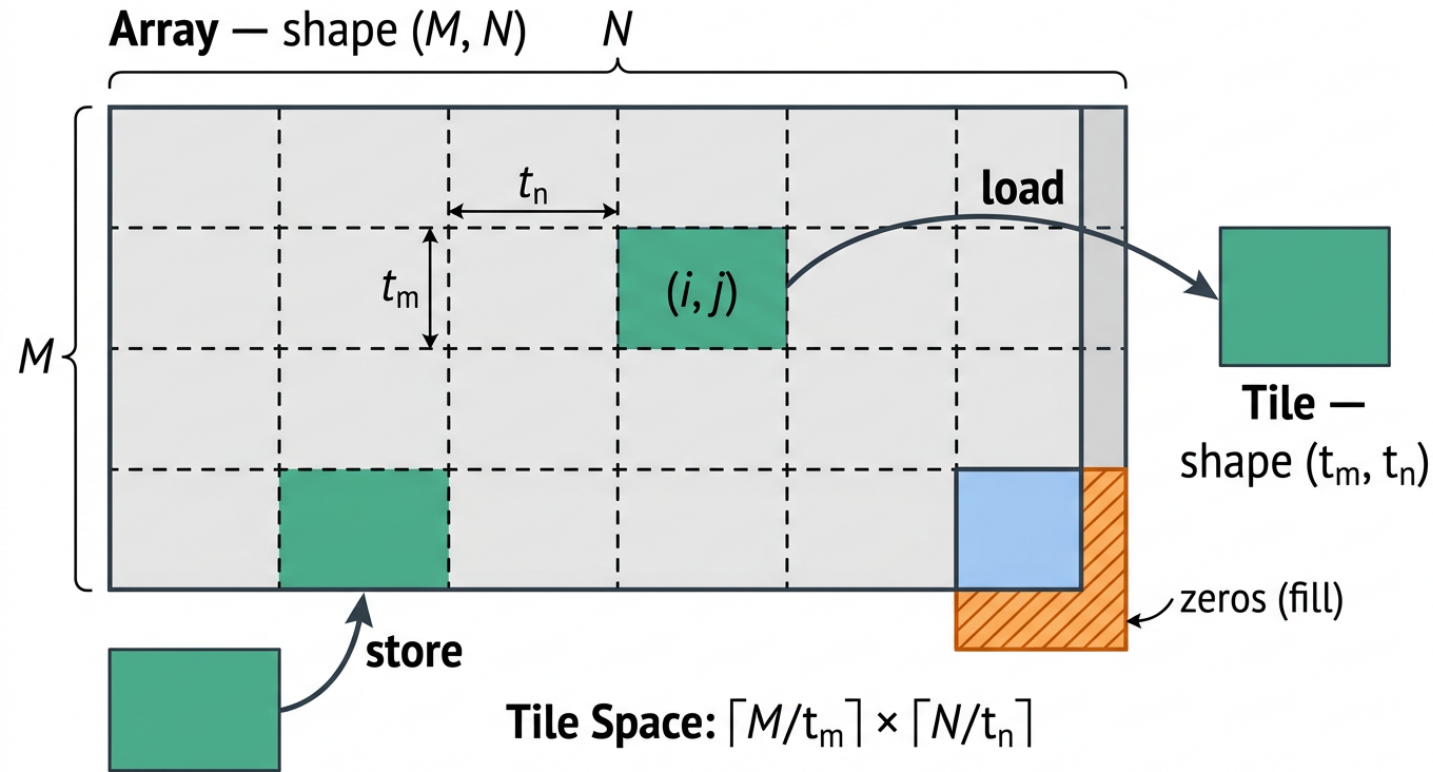
$N = 8$ ,  $d = 2$ , FP16, 2 rows per tile

Transferred: 0 / 640 bytes



Q, K, V are in HBM; S and P are not computed yet.

# What Is Tiling?



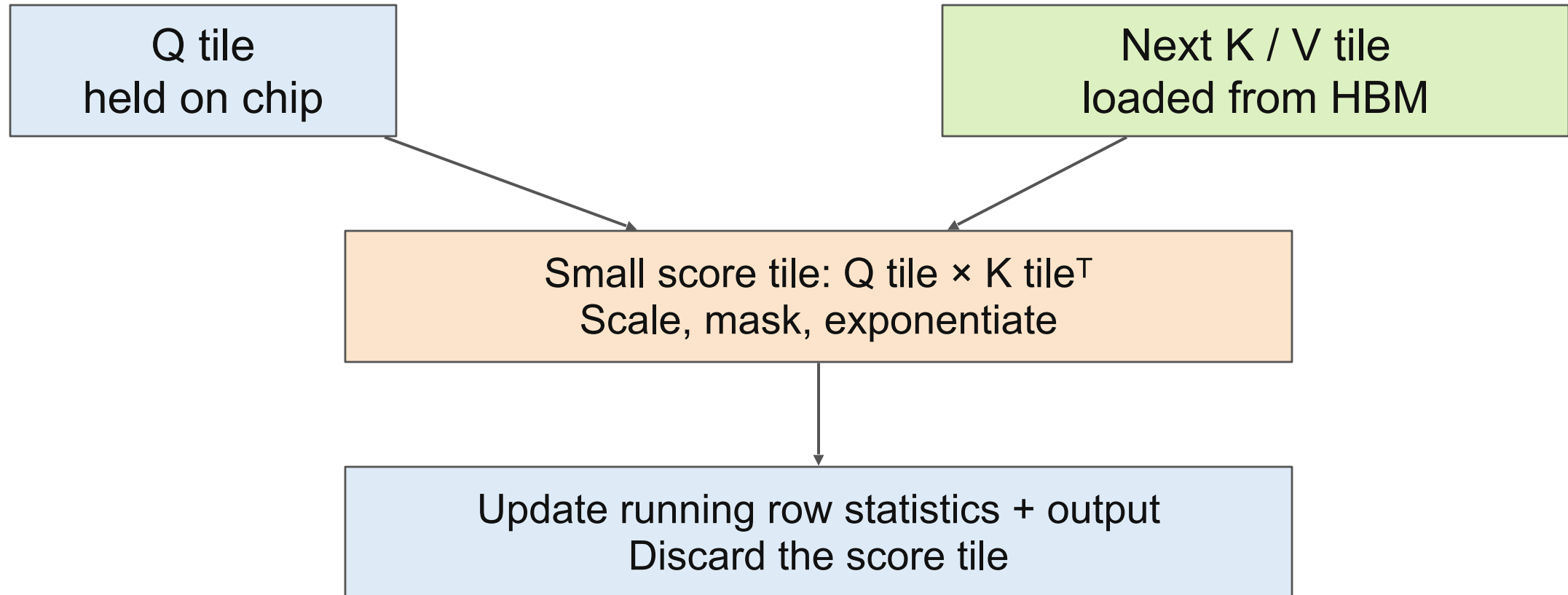
A tile is a small region of a larger tensor.

Load a tile into fast on-chip storage.

Reuse its values before loading another tile.

A tile describes data. A thread block describes the threads working on it.

# FlashAttention: One Query Tile at a Time

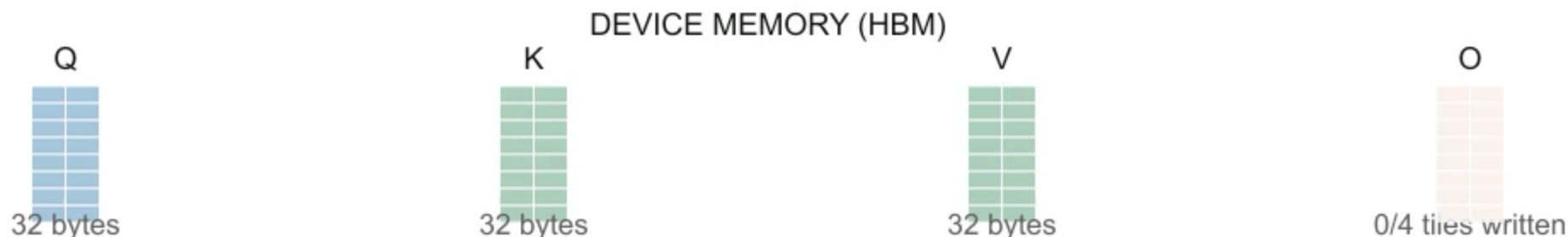


Repeat for all K/V tiles, then write the final output tile to HBM.

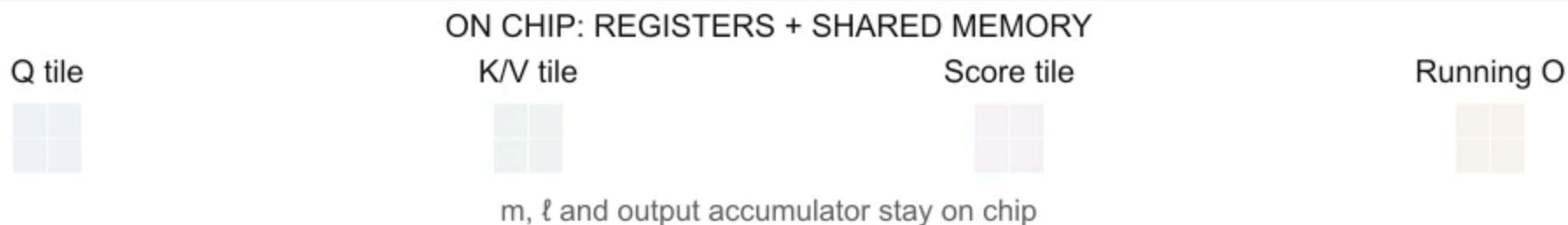
# FlashAttention: Memory Movement

$N = 8$ ,  $d = 2$ , FP16, 2 rows per tile (FA2 loop order)

Transferred: 0 / 320 bytes



S and P: no full matrices in HBM  
K/V are read again for each new Q tile



One query tile will remain on chip while all K/V tiles pass through.

# Why Softmax Makes Tiling Harder

One query, four keys:  $\exp(\text{scores}) = [1, 1, 3, 3]$

Tile A  
 $[1, 1] / 2 = [\frac{1}{2}, \frac{1}{2}]$

Tile B  
 $[3, 3] / 6 = [\frac{1}{2}, \frac{1}{2}]$

Separate tile softmaxes lose the relative weight of the two tiles.

Full row:  $[1, 1, 3, 3] / 8 = [\frac{1}{8}, \frac{1}{8}, \frac{3}{8}, \frac{3}{8}]$

# Combining Tiles Without Keeping Scores

One query, four keys:  $\exp(\text{scores}) = [1, 1, 3, 3]$

Keep each tile's total weight and weighted value sum.

$$oA = (v_1 + v_2) / 2 \quad oB = (v_3 + v_4) / 2$$

$$o = (2oA + 6oB) / 8 = \frac{1}{4}oA + \frac{3}{4}oB$$

Per query row, retain a running maximum, normalizer, and output sum.

Rescale old state when the maximum changes to keep exponentials stable.

# HBM Traffic: Naïve vs. FlashAttention

One head:  $N = 8,192$ ,  $d = 128$ , FP16, full attention

**Naïve: 520 MiB**

Write + read S: 256 MiB

Write + read P: 256 MiB

Read Q, K, V + write O: 8 MiB

**FlashAttention: 260 MiB**

S/P transfers: 0 MiB

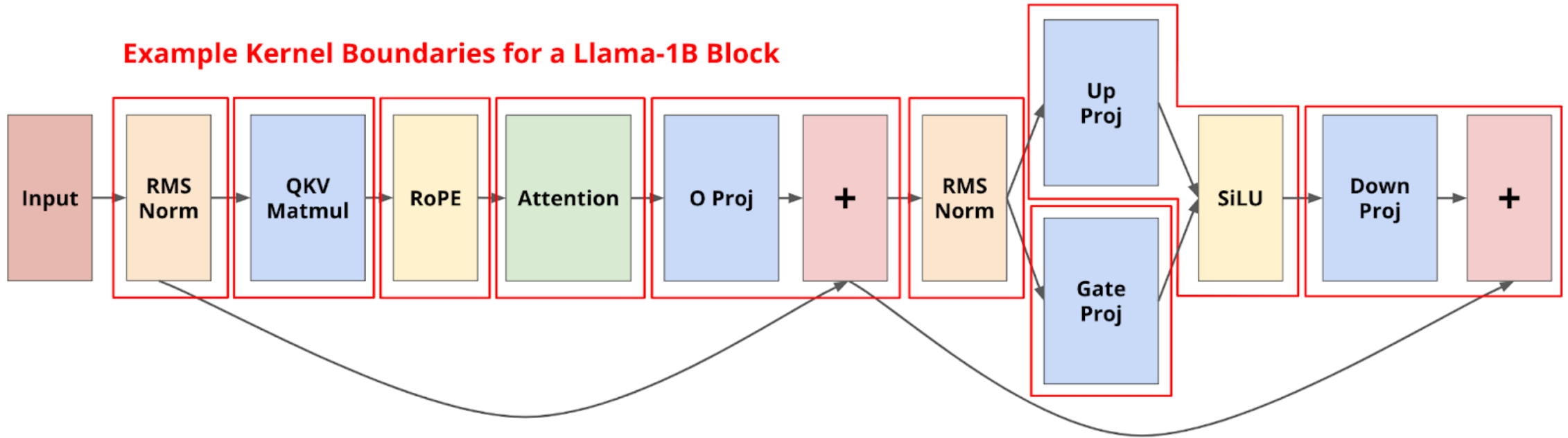
64 Q tiles  $\times$  4 MiB K/V: 256 MiB

Read Q + write O: 4 MiB

# BREAK TIME

- 5 minutes

# Separate Kernels' Boundaries



- Red boxes show separate kernels in a Llama-1B block

# Kernel Fusion

**Kernel fusion combines operations into one GPU kernel.**

## **Separate: two kernels**

Kernel 1:  $t = x + b$   
Write  $t$  to global memory.

Kernel 2:  $y = \text{ReLU}(t)$   
Read  $t$ , then write  $y$ .

Two launches + temporary  
write/read of  $t$ .

## **Fused: one kernel**

Load  $x$  and  $b$ . Compute  $t$ .  
Keep  $t$  in registers.

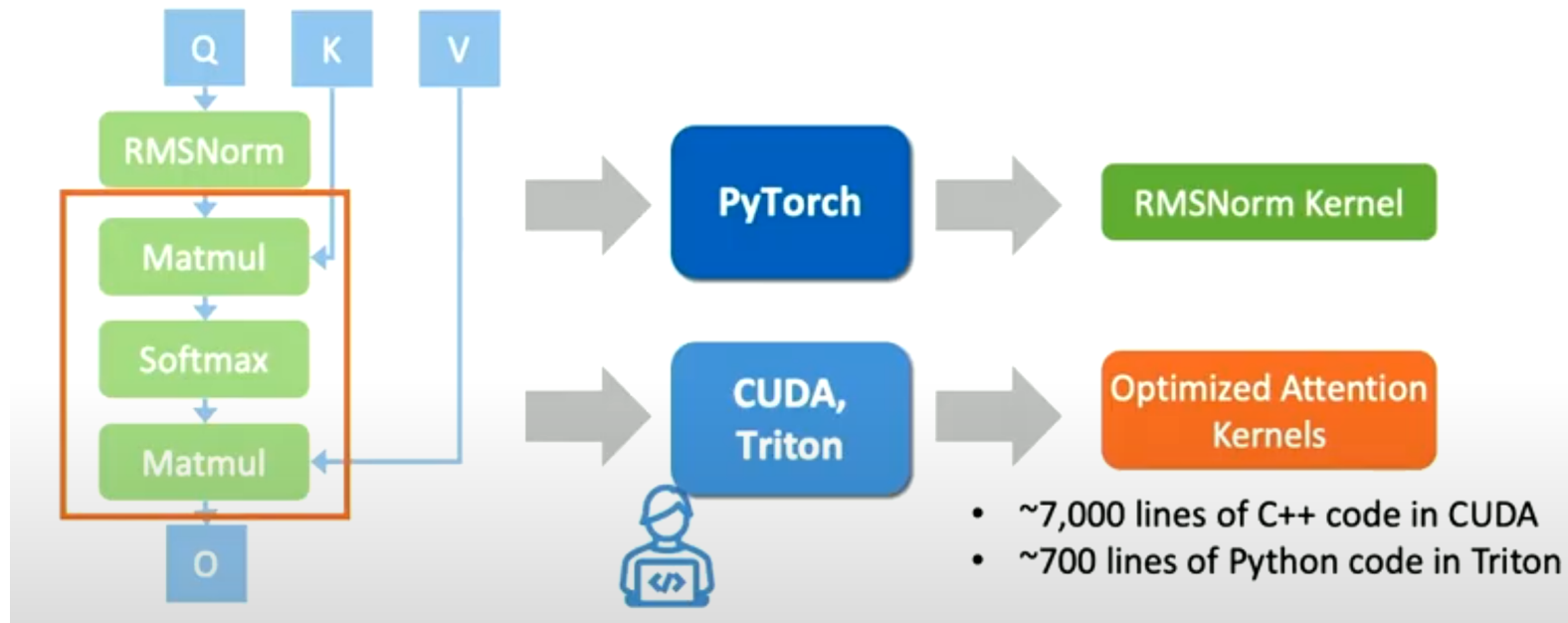
Apply ReLU to  $t$ .  
Write  $y$  once.

One launch. No global-memory  
write/read of  $t$ .

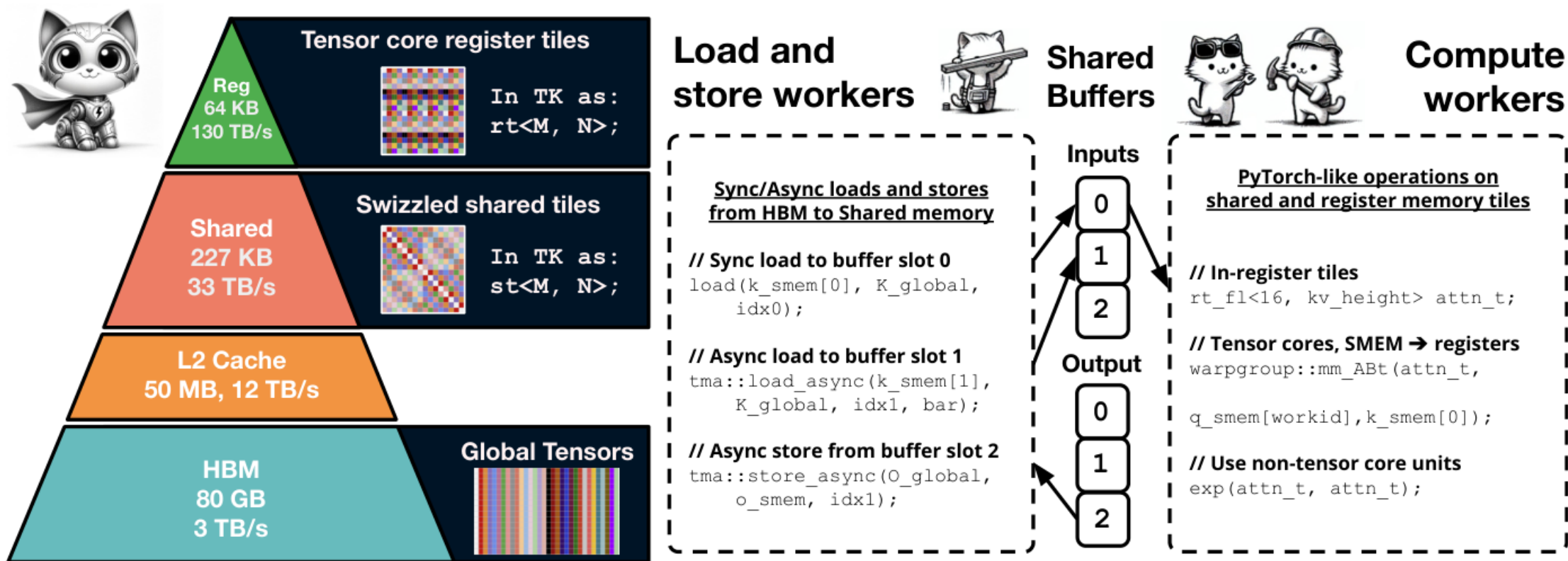
**Motivation: fewer launches and less intermediate data movement.**

# Megakernels

Combining multiple GPU kernel launches and inter-kernel communication, into one large, persistent kernel with GPU-controlled instruction/dispatch scheduling.

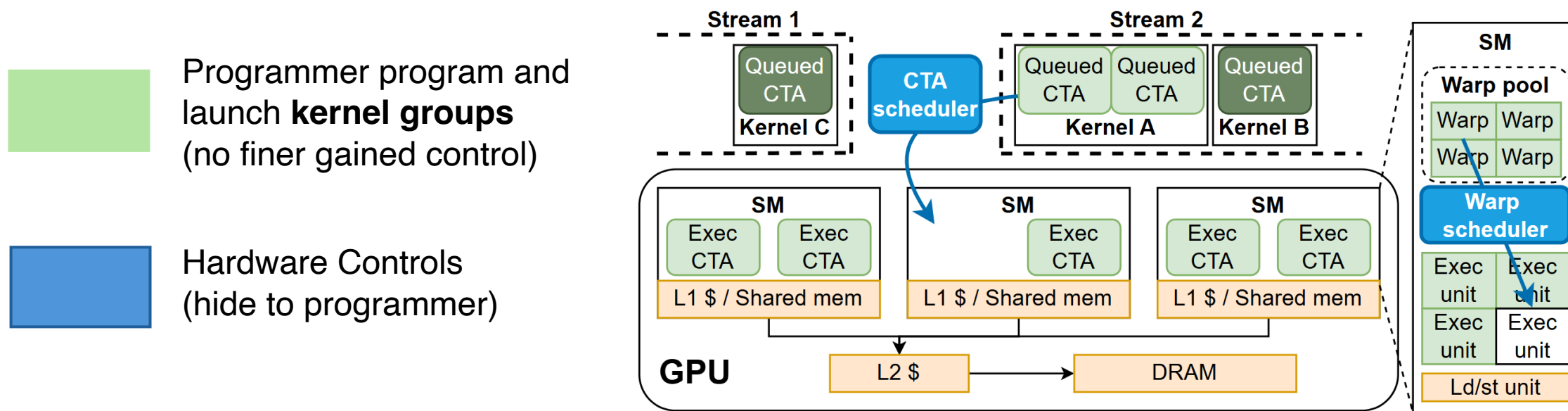


# ThunderKittens: Tiles and Pipelines



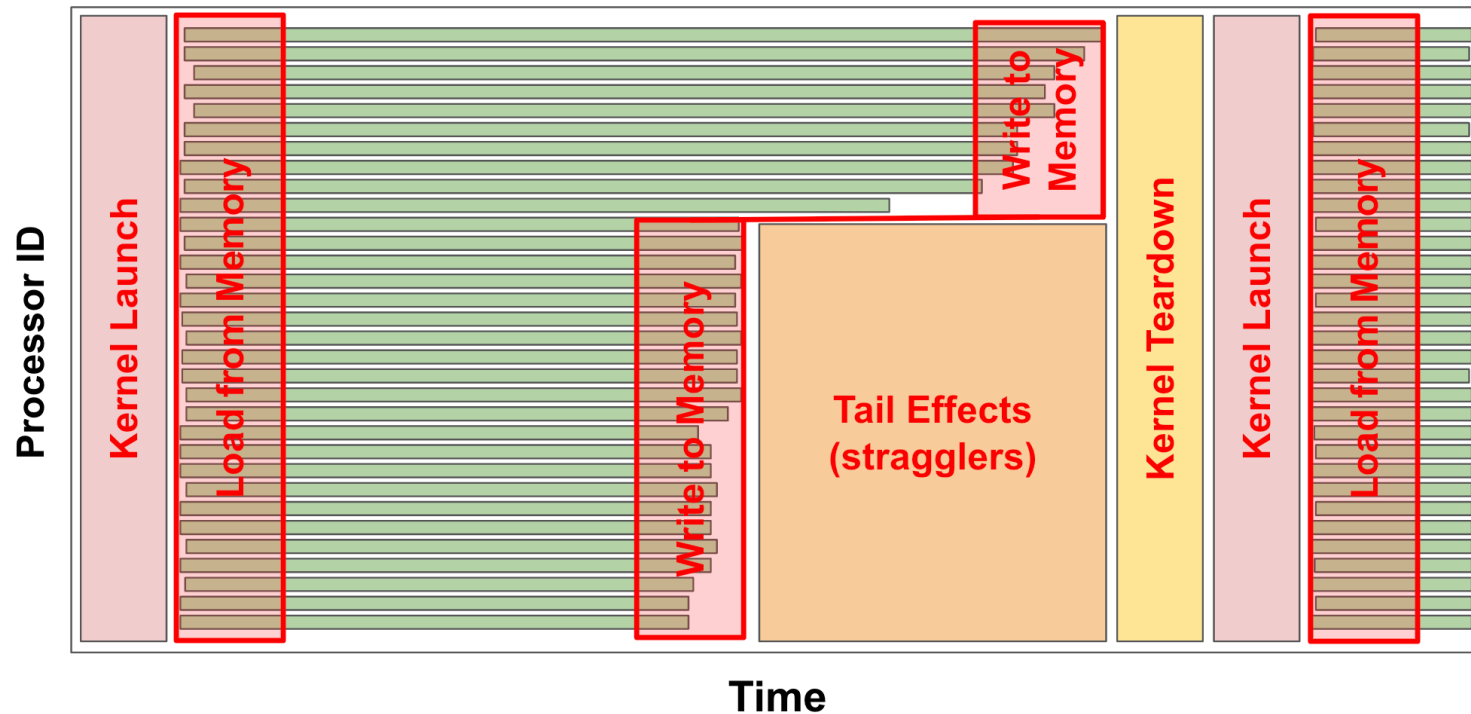
# Programming and Execution Misaligned

- Key motivation: Higher utilization of hardware units vs. CPU-based control overhead and opaque hardware scheduling



# Programming and Execution Misaligned

- Key motivation: Higher utilization of hardware units vs. CPU-based control overhead and opaque hardware scheduler



# Coarse Dependencies and Tail Effects

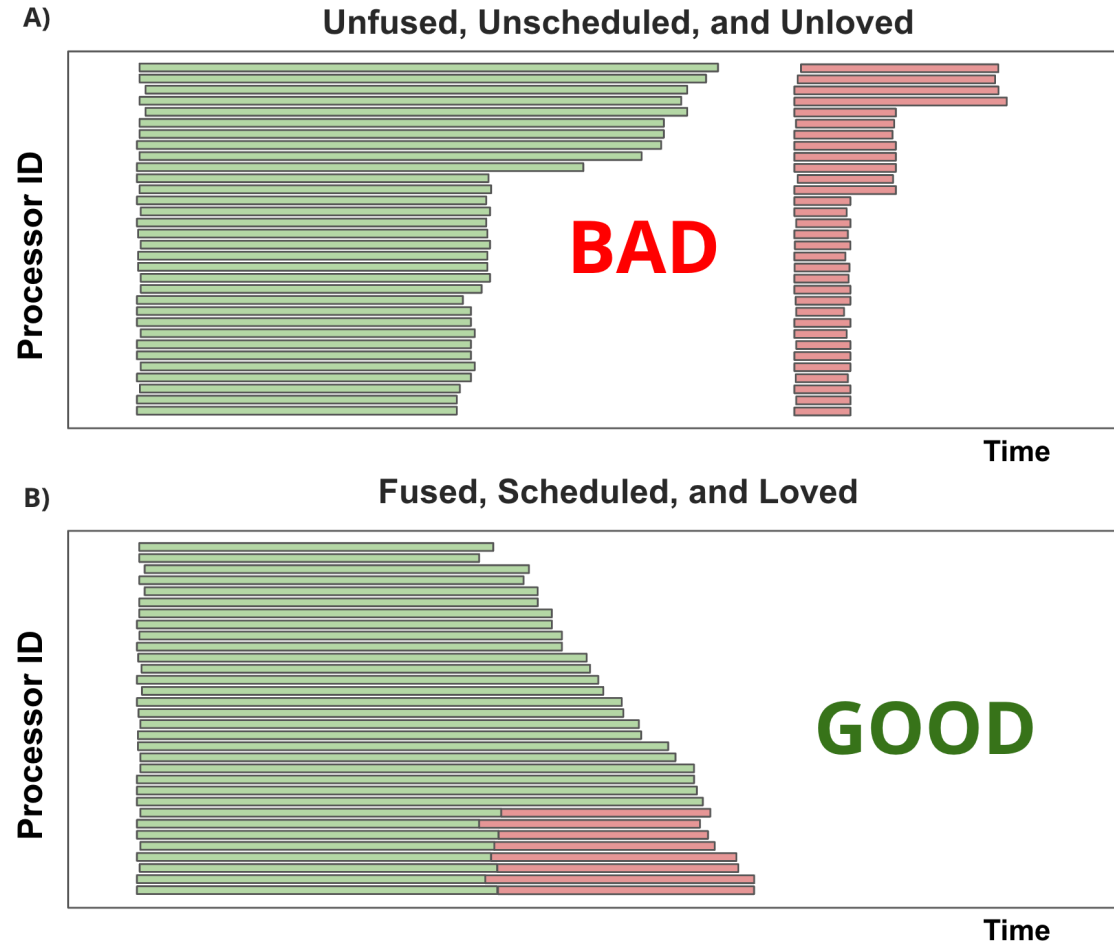
## One stream: A then B

B waits for all of A, even if some inputs for B are ready.

Early-finishing SMs sit idle while stragglers finish.

## Megakernel tasks

Start a consumer when its own required inputs are ready.



# Hardware Support for Megakernels

**Persistent kernels existed earlier. New hardware improves overlap.**

**Ampere: asynchronous copies**

Global-to-shared copies overlap with computation.

**Hopper: Tensor Memory Accelerator (TMA) + async tensor cores**

Loader warps issue bulk transfers while compute warps use tensor cores.

**Hardware barriers and on-chip buffers**

Track completion and hold several pipeline stages safely.

**Warp specialization uses these features. Thread divergence still exists.**

# Issue 1: Fusing Many Operations

## **Solution: an on-GPU interpreter**

Represent operations as instructions with a common load/compute/store template.

Each SM executes a precomputed instruction stream inside one kernel.

Example instruction: RMSNorm + QKV + RoPE.

Build schedules on the CPU and reuse them across forward passes.

# Issue 2: Sharing Shared Memory

**Solution: page the SM's shared memory**

Prefetching needs free space while earlier operations still run.

Instructions request pages and release them when finished.

Freed pages go to the next instruction. Its weight loads overlap earlier computation and stores.

# Issue 3: Synchronizing Tasks

**Solution: counters track producer completion**

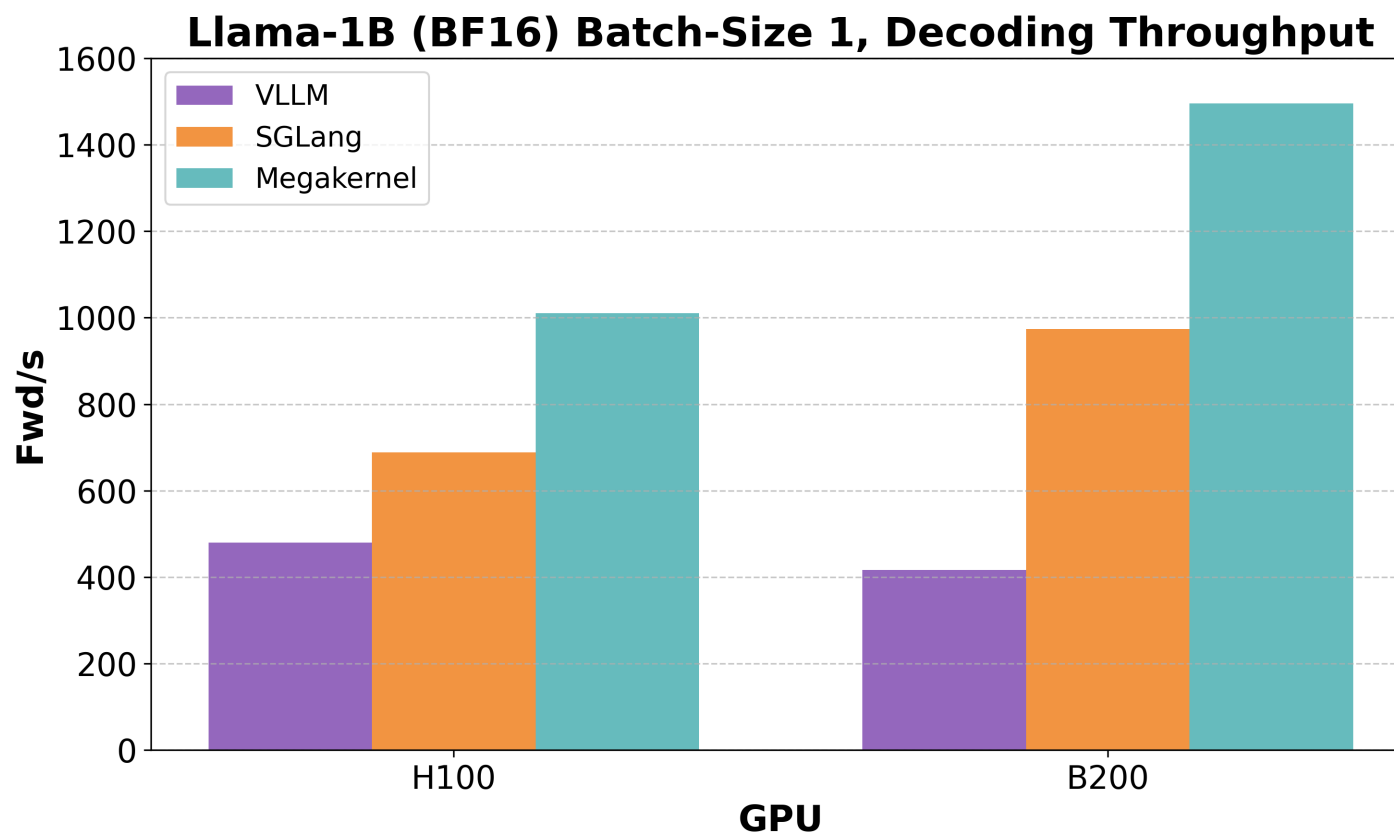
Initialize dependency counters in GPU global memory.

Producers write outputs, then increment completion counters.

Consumers wait for their required counters to reach target values.

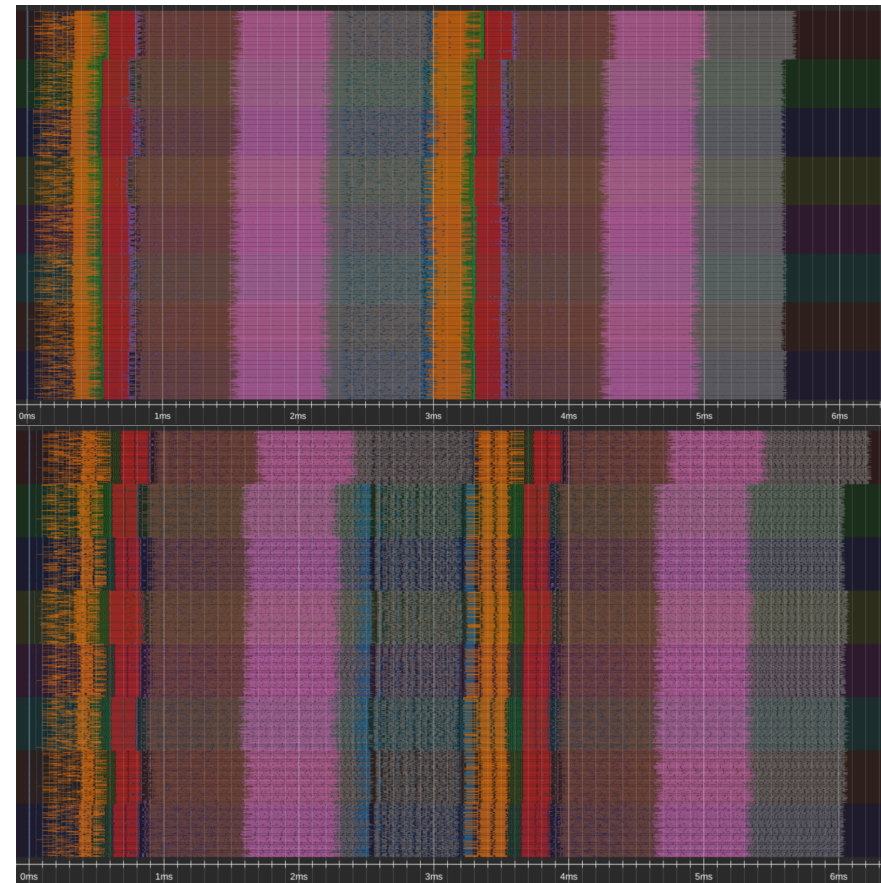
# TK Megakernel: Batch-1 Decode (2025)

- Llama-3.2-1B
- BF16 weights
- 32 prompt tokens
- 128 output tokens
- May 2025 baselines



# Understanding the benefits of Megakernels

- Inter-task parallelism can keep hardware resources busy across operation boundaries

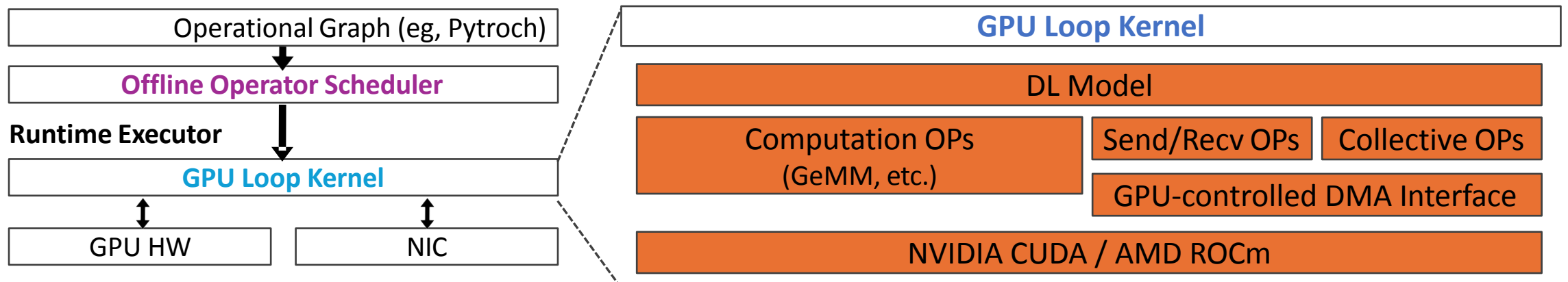


Less gaps in the graph with inter-task parallel (up)

# How to build a Megakernel system?

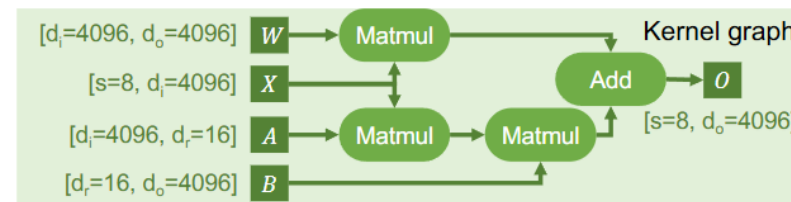
Essential components:

- **Task Set**: Internal supported operations
- **Compiler**: Convert and optimize program to internal tasks
- **Scheduler**: Dispatch internal task to workers
- **Worker**: Fused kernel, one kernel to execute all tasks

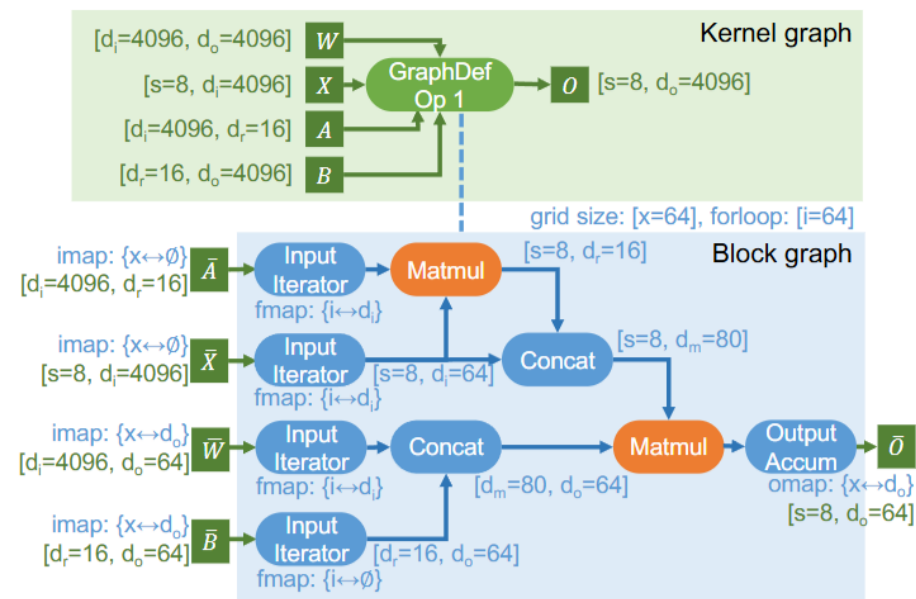


# Megakernel Task Set and Compiler

- Translate the computation graph (e.g, a pytorch graph or a cuda stream/graph) into internal task graph.



(a) The kernel graph for LoRA in existing systems.



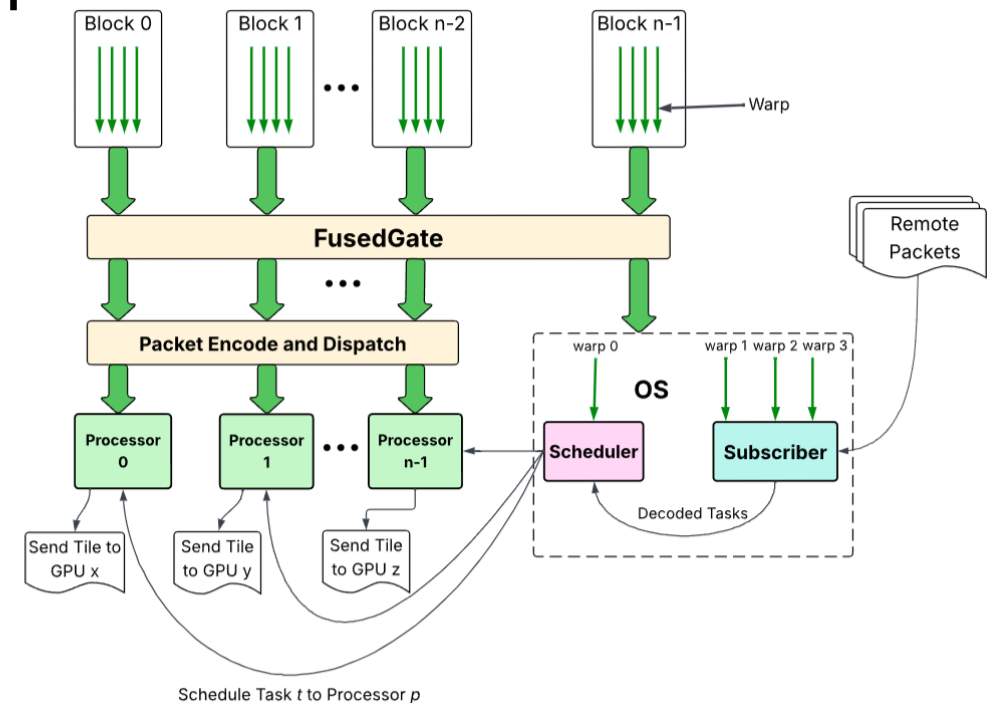
(b) The best  $\mu$ Graph discovered by Mirage for LoRA.

# MegaKernel Scheduler

- Distributed co-routine scheduler vs. separated scheduler
- Scheduling granularity – CTA, Warp or Thread?

```
1  if (threadIdx.x == 0) { // Leader thread finds assignment
2      int sm_id; // Find which SM this CTA is on
3      asm volatile("mov.u32 %0, %%smid;" : "=r"(sm_id));
4      // For this SM, what do we want to run?
5      const int ratio = (prefill_ratio + decode_ratio);
6      int op, ticket = (atomicAdd(&sm_ctr[sm_id], 1) % ratio);
7      if(ticket < prefill_ratio) op = PREFILL;
8      else op = DECODE;
9      // Get the next CTA for operation
10     int cta_id = atomicAdd(&cta_assign[op], 1);
11     // If the CTA exceeds the max CTA for that op switch ops
12     if (op == PREFILL && cta_id >= prefill_ctas) {
13         op = DECODE;
14         cta_id = atomicAdd(&cta_assign[op], 1);
15     } else if (op == DECODE && cta_id >= decode_ctas) {
16         op = PREFILL;
17         cta_id = atomicAdd(&cta_assign[op], 1);
18     }
19     // Write the CTA ID and operation to shared memory
20     shared_mem[0] = cta_id;
21     shared_mem[1] = op;
22 }
23 __syncthreads(); // Barrier: waits for scheduling to finish
24 // Fetch the assigned CTA and operation.
25 int cta_id = shared_mem[0];
26 const int op = shared_mem[1];
27 __syncthreads();
28 // Perform the appropriate operation
29 if (op == PREFILL) prefill_op(cta_id);
30 else decode_op(cta_id)
```

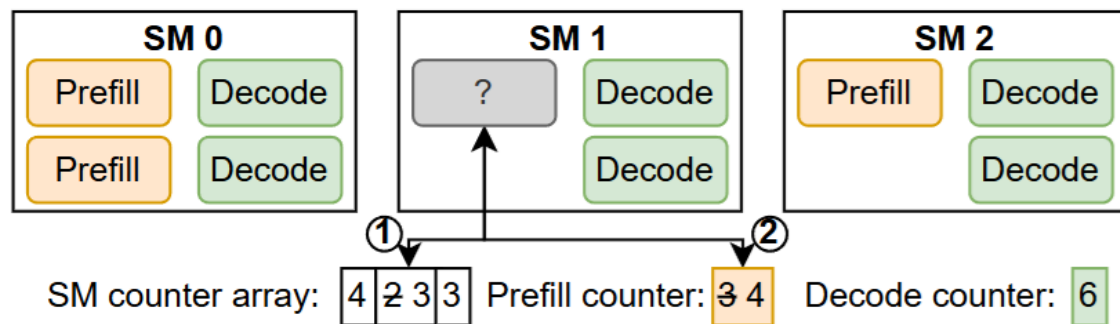
Co-routine scheduler



Separated scheduler (warp-level)

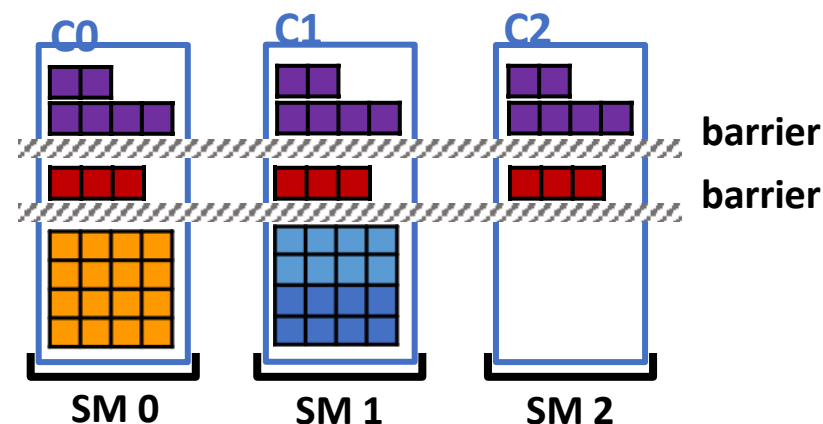
# MegaKernel Scheduler

- Distributed co-routine scheduler vs. separated scheduler
- Scheduling granularity
- Runtime vs. static scheduling



Runtime scheduling (based on the current system status, in this example, the number running P/D instances)

(Offline) Scheduling (Example)



Static scheduling (dispatch statically or at offline)

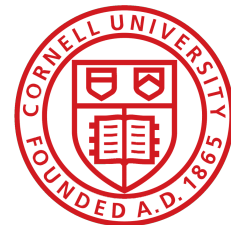
# VD Cores: Virtual Decoupled Cores for Modern Asynchronous GPUs

---

Zhiyuan Guo, Zijian He, Adrian Sampson, **Yiying Zhang**



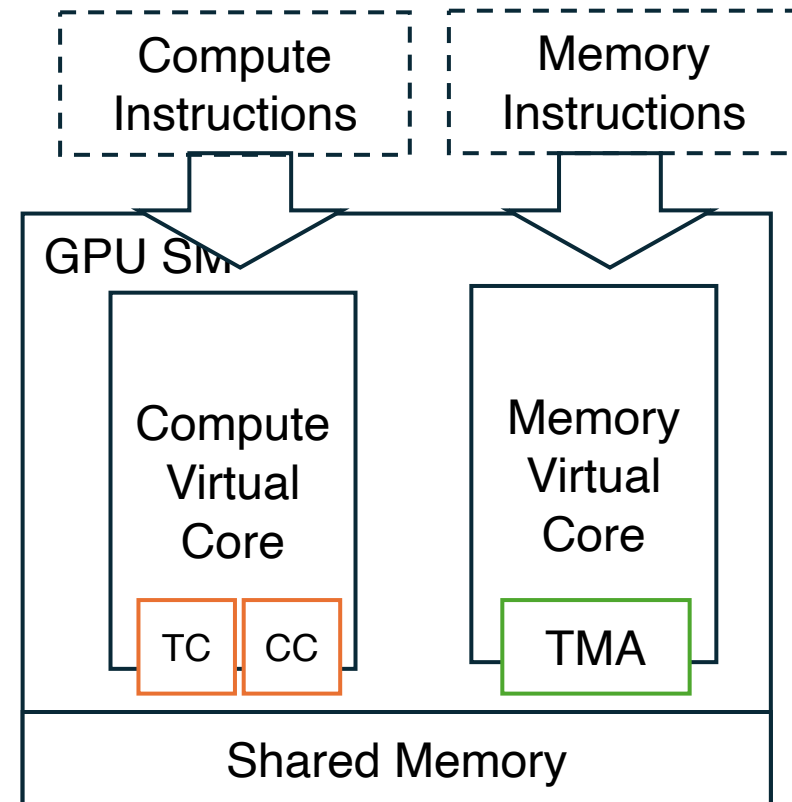
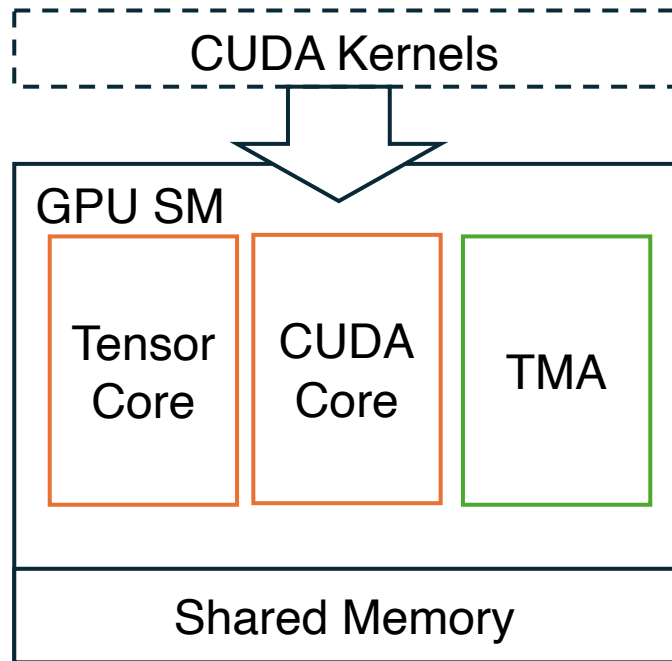
**UCSD CSE**  
Computer Science and Engineering



Cornell University®

# Decoupling: A Better Way to Handle Asynchrony

Decompose kernels into independent, asynchronously executing units, **Decoupled Cores**.

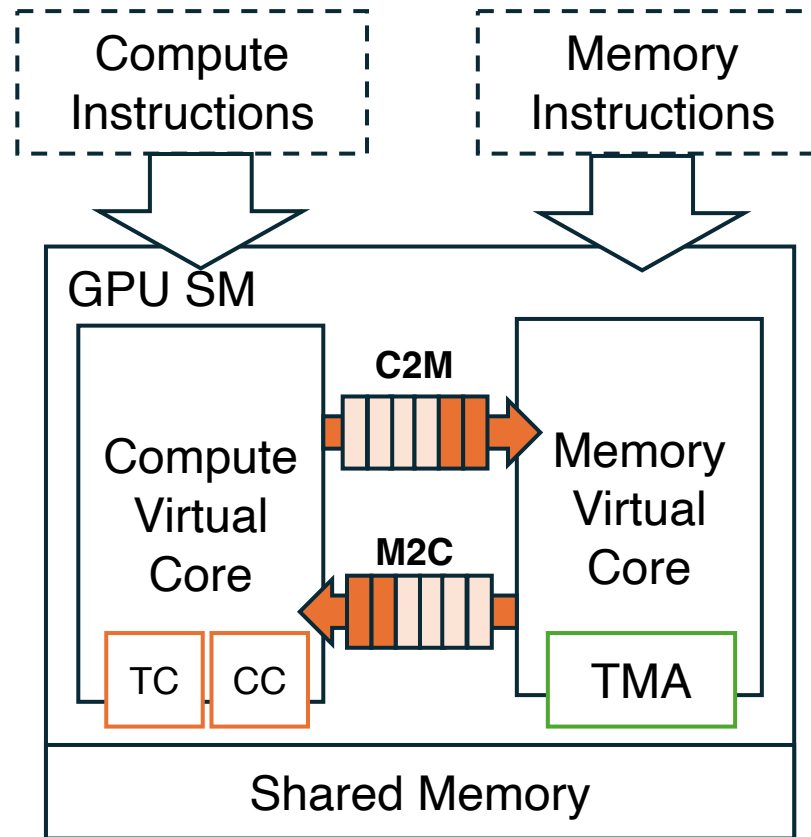


# Principles of Decupled Cores Model

1. **Isolation**: only execute based on local state
  - Separate *what* happens from *when* it happens
2. **Asynchrony**: Handle explicit dependency messages.
  - Data dependency, not control flow or kernel launch dependency
  - Parallelize when no dependency arises.

# Decoupled Cores in Action

- Pre-compiled CUDA tasks
- Manages compute Tensor&CUDA cores
- The compute task pops the address of input from M2C and push the used memory to C2M



- Manages shared memory resource
- Allocate for load/store when
  - 1. memory available
  - 2. dependency satisfied
- Pass allocated memory to compute and collect used memory

# DeepSeek-V4: A Production Megakernel

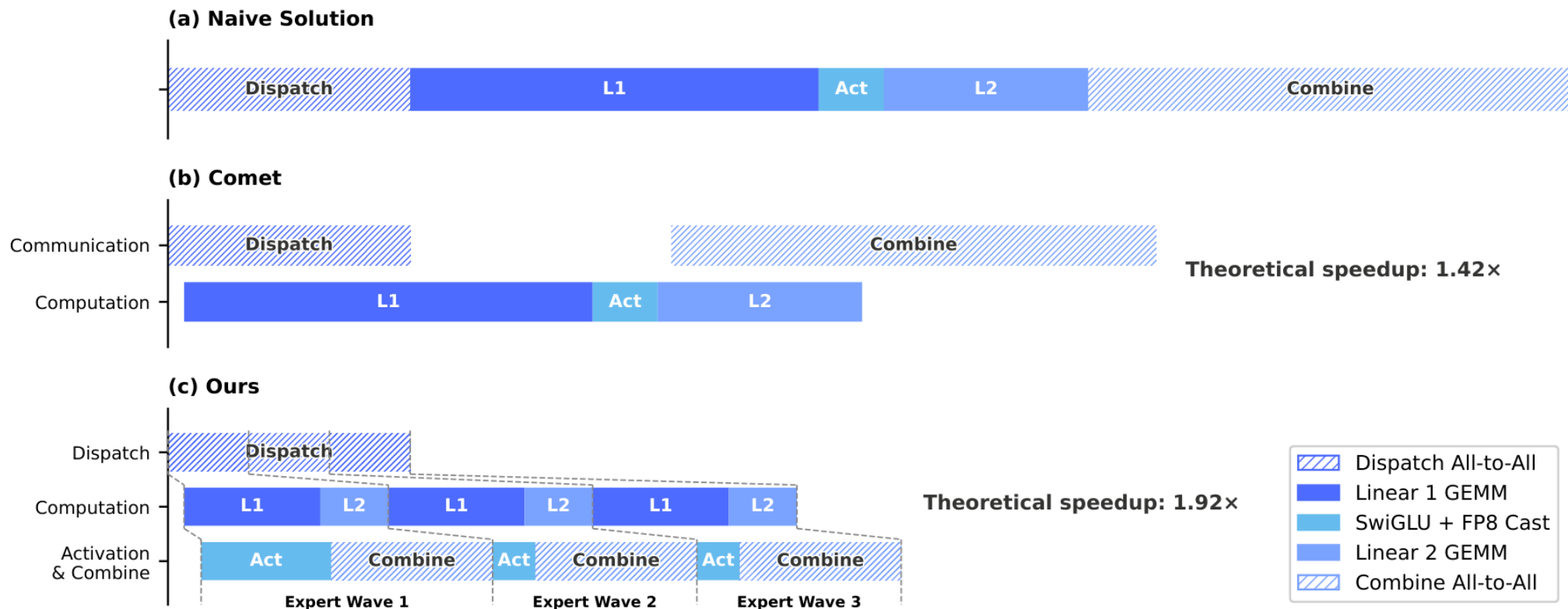


Figure 5: schedule expert waves so communication overlaps computation.

# DeepSeek-V4: Beyond Kernel Fusion

- TileLang replaces many small framework operators with fused kernels
- Generated host launchers reduce repeated Python dispatch and checks
- Batch-invariant and deterministic kernels make numerical behavior a systems requirement
- Discussion: what should a kernel benchmark check besides elapsed time?

# AI for GPU Kernel Generation and Optimization

## **2024–25: Measuring kernel-writing ability**

KernelBench: replace PyTorch operations with kernels.  
Evaluate numerical correctness and speed together.

## **2025: Search and learning from GPU feedback**

Agents generate, compile, test, profile, and revise candidates.  
Stanford explores branching search. Kevin trains this loop with RL.

## **2025–26: Production use across hardware**

Meta's KernelEvolve searches and optimizes production kernels  
for NVIDIA, AMD, and MTIA using hardware-specific feedback.

**Speedups depend on shape, precision, and hardware. Verification remains essential.**

# Project Connection: Profile GPU Inference

- Compare machines using the same model, precision, and workload
- Sweep prompt and output length; separate prefill and decode
- Measure latency, memory use, and sustained bandwidth
- Use profiling to decide whether to optimize kernels or placement

# Next: Edge AI and NPUs

- Hardware speed depends on data movement and execution mapping
- FlashAttention reduces attention traffic; megakernels reduce boundaries
- Next: local inference, resource constraints, and NPU execution
- Required reading: PowerInfer (SOSP '24)
- What changes when model weights do not fit on the accelerator?