Modern AI Systems: Agents and System Optimizations

LLM Serving Basics II: Intro to GPUs

Juncheng Yang



Harvard John A. Paulson
School of Engineering
and Applied Sciences



HARVARD UNIVERSITY
MaDSys
Measurement and Design of Systems Lab

Logistics

- Assignment 4 will go out today and due in two and half weeks
  - **Goal: reason about performance, bottlenecks and improve iteratively, how real-world workloads differ from synthetic benchmarks**
  - Write-up is important, you only need to write a small number of lines of code
  - NO LATE DAY for leaderboard submission. DO NOT grab more than one GPU
  - Hard if you have limited prior experience in systems programming
- Proposal due next Wednesday (Oct 28)
  - No limit on format or page
  - DO NOT upload AI-generated content

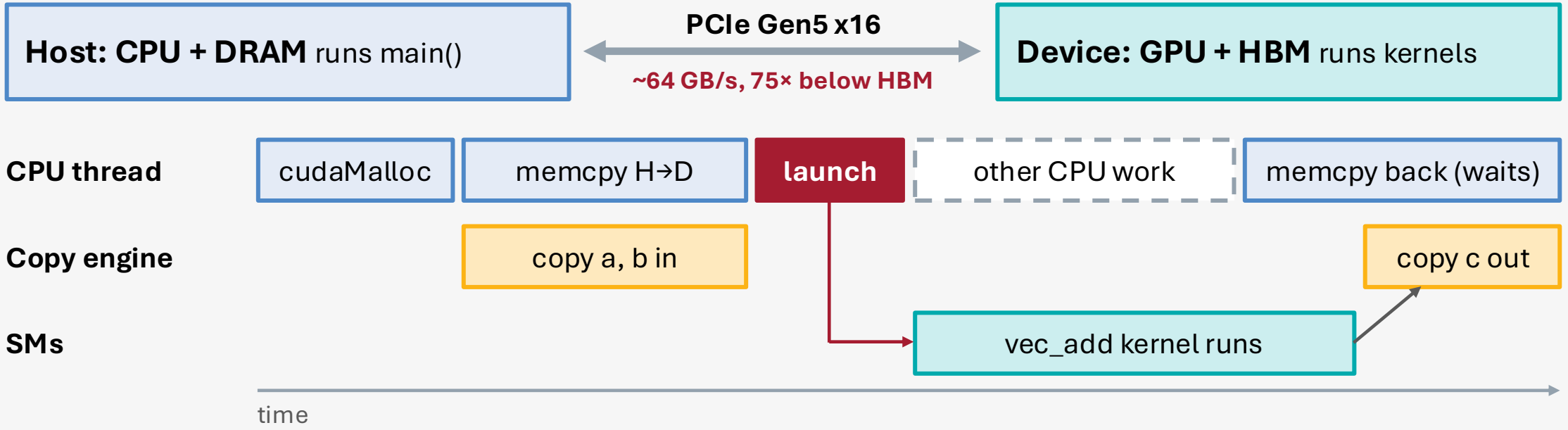
Plan for today

- Introduction to GPU for LLM serving[1]
  - GPU basics
  - GPU architecture
  - GPU programming and recent kernel advances

[1] We will not go into details; the goal is to help you build intuition

GPU Programming

GPU code runs asynchronously



Launch is asynchronously
CPU does not wait for GPU kernel to finish.

Errors surface late
A kernel fault shows up at the next sync call.

NVIDIA, CUDA C++ Programming Guide, 2024; PCI-SIG, PCIe 5.0, 2019

A kernel is written for one thread (SIMT)

```
// one thread computes one output element
__global__ void add(float* x, float* y,
                    float* out, int n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) out[i] = x[i] + y[i];
}

// launch: n = 1M elements
int threads = 256;
int blocks  = (n + threads - 1) / threads;
add<<<blocks, threads>>>(x, y, out, n);
```

Qualifier	Runs on	Called from
`__global__`	device	host
`__device__`	device	device
`__host__ __device__`	both	either

Global index

Each thread finds its element from block and thread IDs.

4,096 blocks

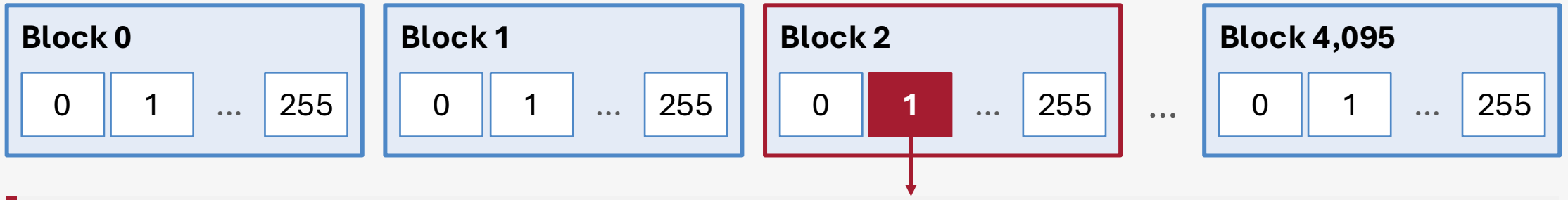
1M ÷ 256 threads; the GPU spreads them over 132 SMs.

Pick block sizes in multiples of 32

128–256 threads is typical; warps come from the linear tid.

Each thread's IDs pick its array element

1 LAUNCH: 4,096 BLOCKS × 256 THREADS

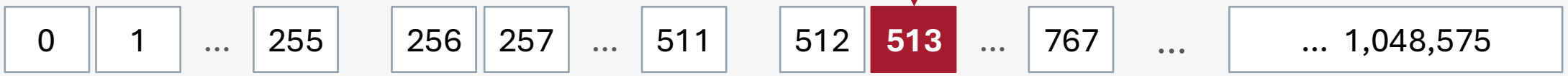


2 EACH THREAD COMPUTES ITS INDEX

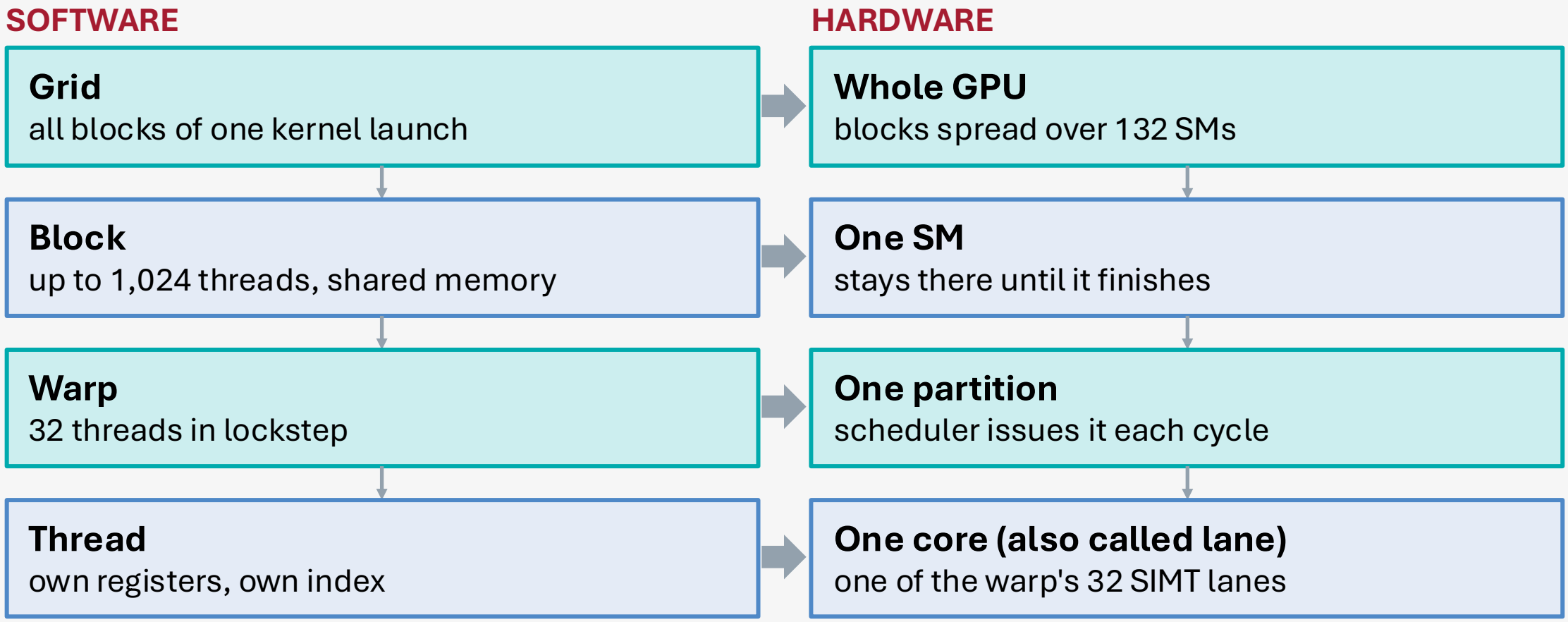
```
i      =   blockIdx.x   *   blockDim.x   +   threadIdx.x
513    =   2            ×   256           +   1
```

element = which block × block size + slot in block

3 ...AND WRITES ITS OWN ELEMENT out[i]



Blocks run on SMs as warps of 32



Graph compilers (torch.compile): code the model, the compiler writes the kernels

torch.compile

```
import torch

def block(x, w, b):
    h = x @ w + b
    return gelu(h) * 0.5

# capture the graph, fuse, generate kernels
fast = torch.compile(block,
                     mode="max-autotune")
y = fast(x, w, b)  # first call compiles

# JAX: jax.jit(block) lowers to XLA
```

You decide

The math, in ordinary PyTorch or JAX. No threads, tiles or memory.

The compiler decides

Which ops fuse, tile sizes, and the generated Triton or XLA kernels.

Stops working when

Control flow breaks the graph, or the slow op cannot be fused.

Ansel et al., PyTorch 2, ASPLOS 2024; OpenXLA documentation, 2026

Tile languages: describe the work of a whole block, not one thread

	CUDA C++	Tile language
You write	the work of one thread	the work of one tile of data
Indexing	`blockIdx × blockDim + threadIdx`	program_id + a range over the tile
Memory staging, sync	you stage tiles, call __syncthreads()	the compiler stages and syncs
Tensor Cores	warp-level MMA instructions	one dot product on whole tiles

Tile languages in use

Triton
Python; NVIDIA and AMD GPUs

Pallas
JAX kernels for TPU and GPU

TileLang
Python DSL built on TVM

ThunderKittens
C++ tile templates

cuTile
NVIDIA's tile model in CUDA

Tillet et al., MAPL 2019; Spector et al., ThunderKittens, 2024; NVIDIA CUDA 13.1, 2025

Triton: each program handles one tile of data

```python
# each program handles one BLOCK
@triton.jit
def add(x, y, out, n, BLOCK: tl.constexpr):
    pid = tl.program_id(0)
    offs = pid * BLOCK + tl.arange(0, BLOCK)
    mask = offs < n
    a = tl.load(x + offs, mask=mask)
    b = tl.load(y + offs, mask=mask)
    tl.store(out + offs, a + b, mask=mask)

# launch: one program per tile
grid = (triton.cdiv(n, 1024),)
add[grid](x, y, out, n, BLOCK=1024)
```

One program per tile

A program = one CUDA thread block; it owns BLOCK elements.

Compiler does the rest

Coalescing, shared-memory staging and Tensor Core MMA via tl.dot.

Autotuning picks the tile

@triton.autotune tries BLOCK sizes and warp counts per shape.

TileLang: you place each tile in shared memory, the compiler maps threads

TileLang matmul

```
@T.prim_func
def mm(A, B, C):
  with T.Kernel(N // bn, M // bm,
                threads=128) as (bx, by):
    As = T.alloc_shared((bm, bk), "float16")
    Bs = T.alloc_shared((bk, bn), "float16")
    Cl = T.alloc_fragment((bm, bn), "float")
    T.clear(Cl)
    for k in T.Pipelined(K // bk, num_stages=3):
      T.copy(A[by * bm, k * bk], As)
      T.copy(B[k * bk, bx * bn], Bs)
      T.gemm(As, Bs, Cl)
    T.copy(Cl, C[by * bm, bx * bn])
```

You decide

Tile shapes, shared vs. register storage, and stages (tiles loaded ahead).

TileLang decides

Thread mapping, copy instructions, swizzles and the MMA calls.

Use it for

GEMM and attention variants that need explicit staging.

ADVANCED — skim, or read more on your own

CuTe, CUTLASS and ThunderKittens: you choose layouts, async copies and warp roles

CuTe layouts (CuTe DSL, Python)

```
# a layout maps a coordinate to an offset
row = cute.make_layout((8, 4), stride=(4, 1))
col = cute.make_layout((8, 4), stride=(1, 8))

# (i, j) -> i*4 + j  versus  i + j*8

# the same algebra describes:
#  thread -> data in a warp
#  swizzled shared-memory tiles
#  TMA boxes and MMA fragments
```

You decide

Layouts, TMA and async pipelines, and which warps load or compute.

The library gives

Tested building blocks for copies, MMAs and pipelines per GPU.

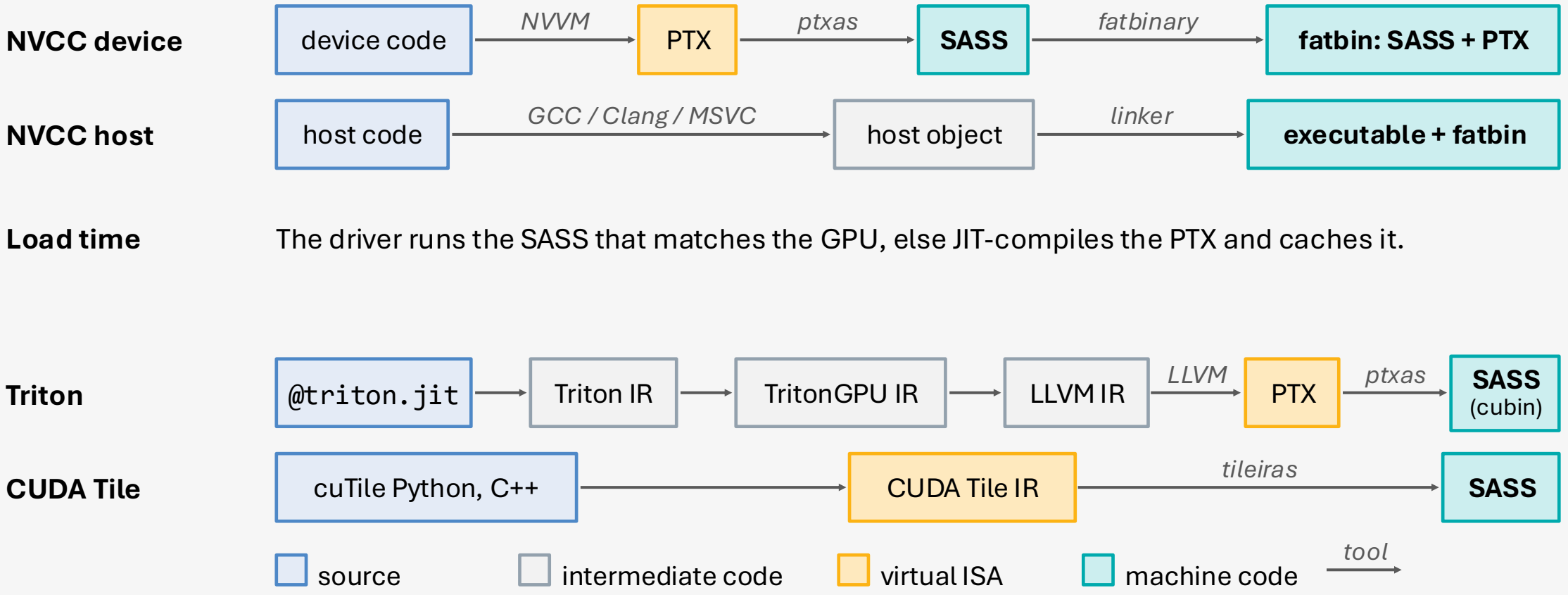
Use it for

The last stretch to peak on GEMM and attention.

Different levels of abstractions

Level	Reach for it when	Cost
Graph compiler torch.compile, XLA	Default for the whole model	Little say over the slow op
Tile language Helion, Triton, Pallas	One op is unfused or missing	Hours; ports across GPUs
Explicit tiles TileLang	You need to control staging	Days
Layouts, warp roles CuTe, CUTLASS, ThunderKittens	GEMM or attention must reach peak	Weeks; per GPU generation
Threads CUDA C++	Irregular work: sort, sparse, graphs	Everything is on you

Every GPU program compiles down to SASS



GPU efficiency and utilization

MFU in inference: model FLOPs utilization

Model FLOPs utilization

MFU = achieved model FLOP/s ÷ peak FLOP/s

Model FLOPs: ≈ 2N per generated token, forward pass only

Peak: the GPU's dense Tensor Core rate

Llama 2 70B on 8 H100s

Model FLOPs per token

2 × 70B ≈ 140 GFLOP

Achieved per GPU

3,040 tok/s × 140 GFLOP ≈ 426 TFLOP/s

MFU

426 ÷ 1,979 TFLOP/s ≈ 22%

Rule of thumb

Training 35–45%;

inference prefill 40–60%, decode 10–40%

Four reasons why MFU is low

Memory-bound kernels

norms, softmax and the optimizer step run at bandwidth, not Tensor Core peak

Communication

tensor-, pipeline- and data-parallel transfers not hidden behind compute

Pipeline bubbles

stages sit idle while each batch fills and drains the pipeline

Recomputation (training)

activation checkpointing reruns forward work that MFU does not count

MFU separates hardware capability from software maturity

Chowdhery et al., PaLM, JMLR 2023; Korthikanti et al., Activation Recomputation, MLSys 2023

MBU: decode efficiency is bytes moved per second, not FLOPs

Model bandwidth utilization

MBU = achieved bytes/s ÷ peak HBM bandwidth

Bytes per decode step: all weights plus the KV cache read

Llama 3 8B, BF16, batch 1, H200

16 GB read per token

At 200 tok/s: 3.2 TB/s ÷ 4.8 TB/s ≈ 67%

Rule of thumb: report MFU for prefill and training, MBU for decode

Other measures of a serving system

Tokens/s per GPU

Throughput at a stated latency target, e.g. p50 time per output token

Energy per token

Joules = average board power ÷ tokens/s

Cost per million tokens

$ per GPU-hour ÷ (tokens/s × 3,600) × 10^6

Profile the timeline, then the kernel

Nsight Systems (nsys)

```
nsys profile -o run ./app
```

Shows: one timeline of CPU threads, CUDA API calls, copies, kernels and streams.

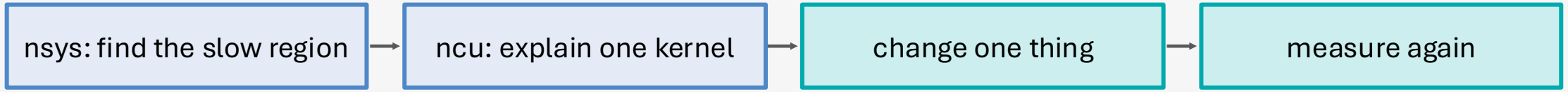
Finds: idle gaps, serialized streams and launch overhead.

Nsight Compute (ncu)

```
ncu --set full -k vec_add ./app
```

Shows: each kernel replayed for counters: speed of light, memory traffic, warp stalls, hot lines.

Finds: why one kernel is slow. Replay timings are for comparison only.



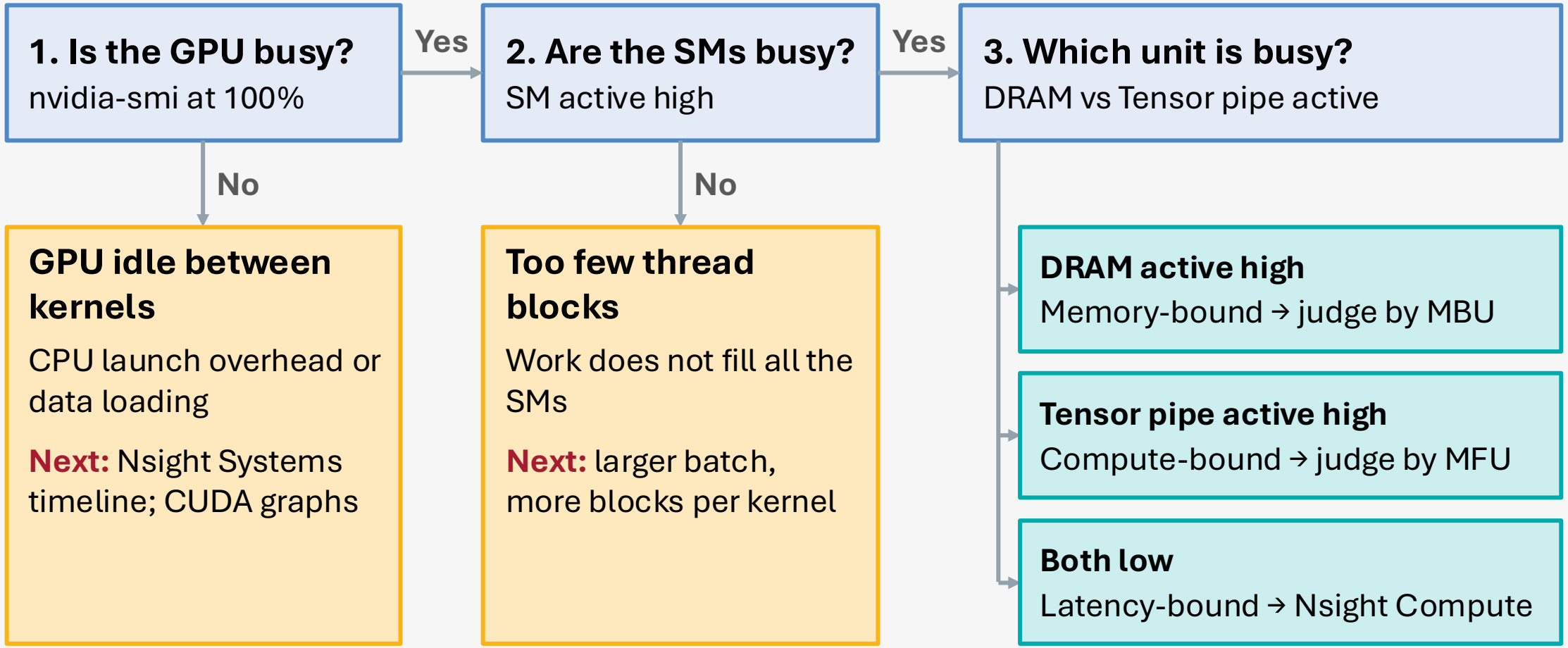
Intuition: scan the whole body (timeline) to find where time goes, then zoom in (kernel) to see why

GPU utilization

Counter	What it measures
nvidia-smi utilization	Share of time at least one kernel is running
SM active	Share of SM cycles with a warp resident
SM occupancy	Resident warps ÷ max warps, all SMs
Tensor pipe active	Share of cycles the Tensor Cores are busy
DRAM active	Share of cycles HBM is moving data

```
dcgmi dmon -e 203,1002,1003,1004,1005
```
one column per counter above

Reading the counters: find which unit is busy



NVIDIA, DCGM User Guide: profiling metrics, 2026

GPU efficiency: what to take away

1 **MFU:** achieved model FLOP/s ÷ dense peak; judge prefill and training by it.

2 **Low MFU:** communication, pipeline bubbles, memory-bound kernels and recomputation eat the peak.

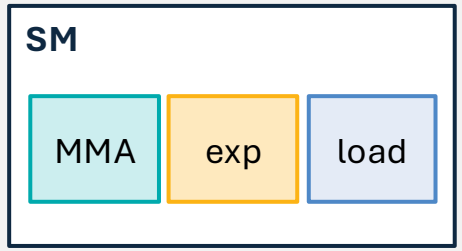
3 **MBU:** achieved bytes/s ÷ peak HBM bandwidth; judge decode by it.

4 **Counters:** nvidia-smi at 100% only means a kernel is running; read SM, Tensor and DRAM activity.

Recent Advances in GPU Kernel Optimizations

Kernels lose GPU time in three places

1 Inside a kernel



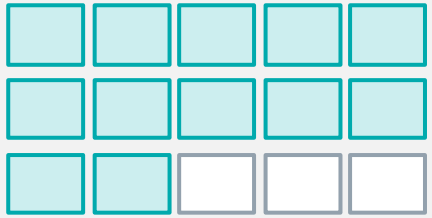
Lost time

Tensor Cores wait on loads, softmax and exp.

Advances

FlashAttention, warp specialization, FlashAttention-4

2 Across SMs



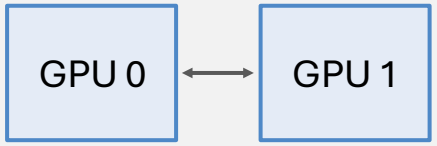
Lost time

SMs sit idle in uneven rounds and between kernel launches.

Advances

Persistent kernels, split-KV, CUDA Graphs, PDL, megakernels

3 Across GPUs



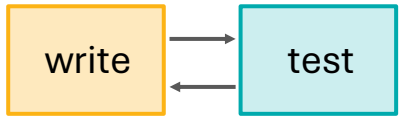
Lost time

GPUs wait for each other's data.

Advances

Fused collectives, MoE all-to-all overlap

Writing kernels



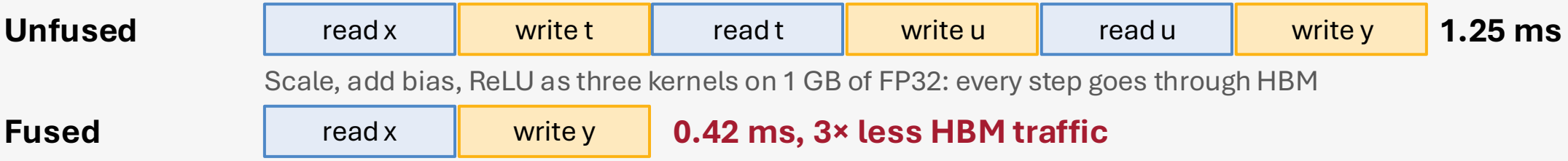
Lost engineer time

Each new GPU needs new hand-tuned kernels.

Advances

Agents that write and test kernels

Kernel fusion
Vertical: chain the steps, horizontal: merge parallel ops



Vertical fusion

A producer feeds its consumer inside one kernel.



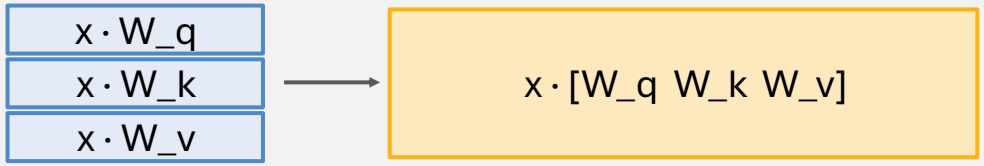
one kernel: each result stays in registers

Saves: HBM round trips

Examples: GEMM epilogues, LayerNorm, softmax with mask

Horizontal fusion

Ops that share an input merge into one kernel.



one GEMM: x is read once, one launch

Saves: launches and repeated reads of x

Examples: QKV projection, FFN gate + up

Compilers do both: torch.compile fuses vertical chains into one Triton kernel; model code often fuses QKV and gate + up weights by hand (concatenating them).

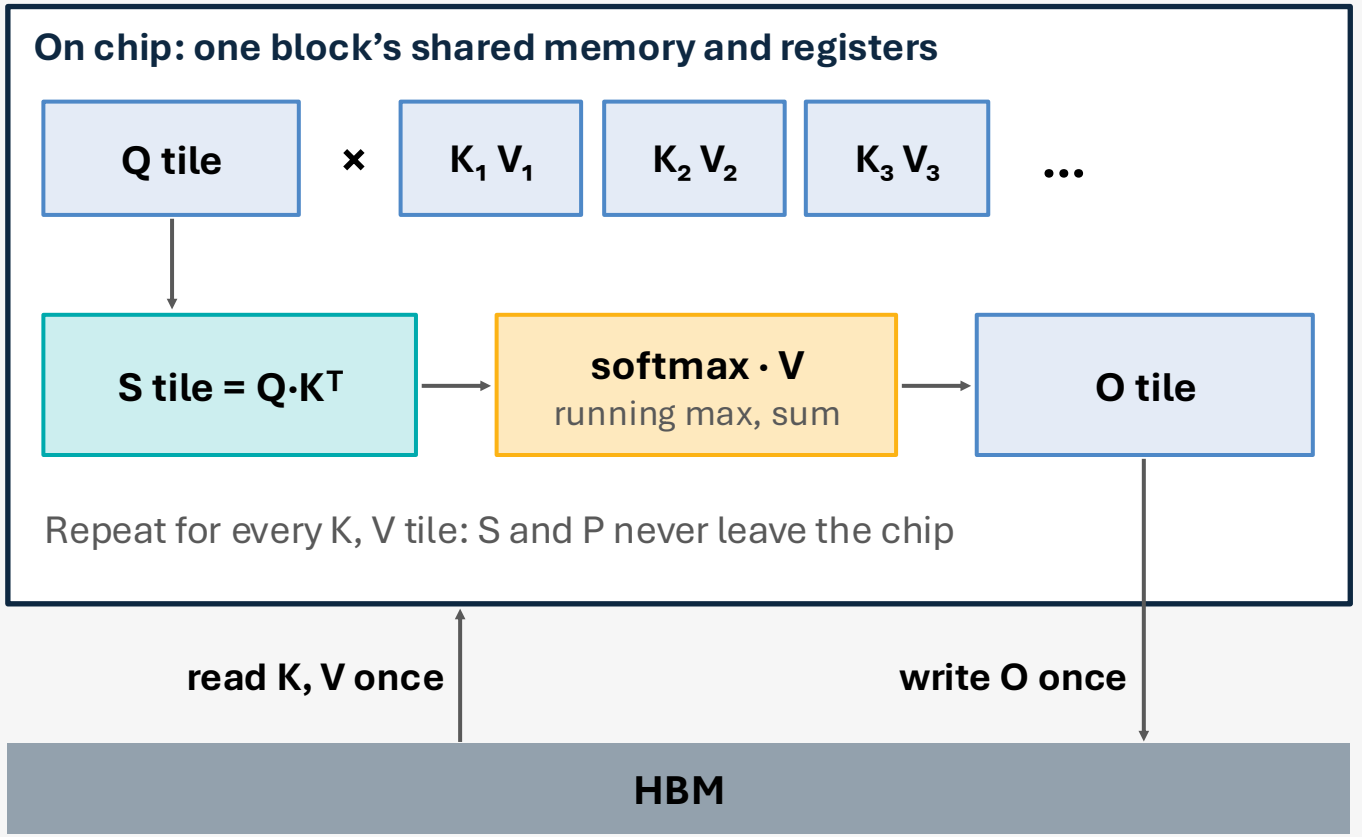
FlashAttention: compute attention tile by tile so S never reaches HBM

The problem

Standard attention writes the n × n score matrix S and its softmax P to HBM, then reads them back.

At 8K tokens that is 67M scores per head in every layer, all moved through HBM.

Solution



Dao et al., FlashAttention, NeurIPS 2022; Dao, FlashAttention-2, ICLR 2024

From FlashAttention to FlashAttention-4

FlashAttention 2022 · A100	FlashAttention-2 2023 · A100	FlashAttention-3 2024 · H100	FlashAttention-4 2025 · B200
Bottleneck HBM traffic: the n × n score matrix is written and read back	**Bottleneck** Low occupancy: one block per head, many non-matmul FLOPs	**Bottleneck** Tensor Cores wait while warps load and compute softmax	**Bottleneck** Tensor Cores doubled, exp unit did not: softmax is the bottleneck
Fix Tile Q, K, V in shared memory; online softmax	**Fix** Split work across the sequence; fewer rescales	**Fix** TMA + warp specialization; overlap softmax with GEMM; FP8	**Fix** TMEM accumulators; part of exp on FMA units
2–4× faster attention; memory linear in length	**About 2× FA1; up to ~70% of A100 peak**	**About 75% of H100 peak in FP16**	**Faster than cuDNN attention on B200**

Dao et al. 2022; Dao 2023; Shah et al. 2024; Dao et al. 2025

Persistent kernels: one thread block per SM works through many tiles

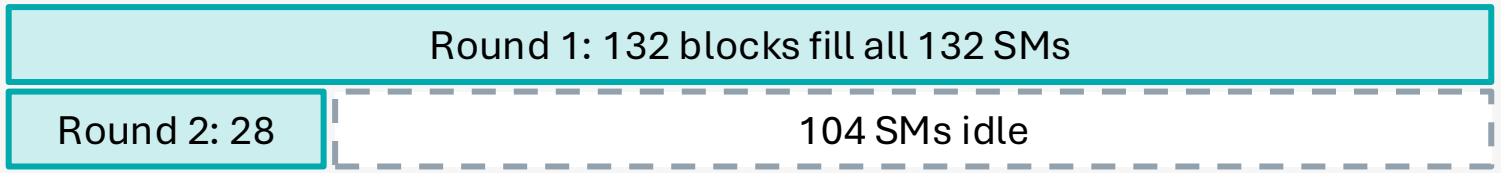
Problem

A matrix multiply is cut into 160 output tiles, one thread block per tile. The GPU has 132 SMs, so the blocks run in two rounds.

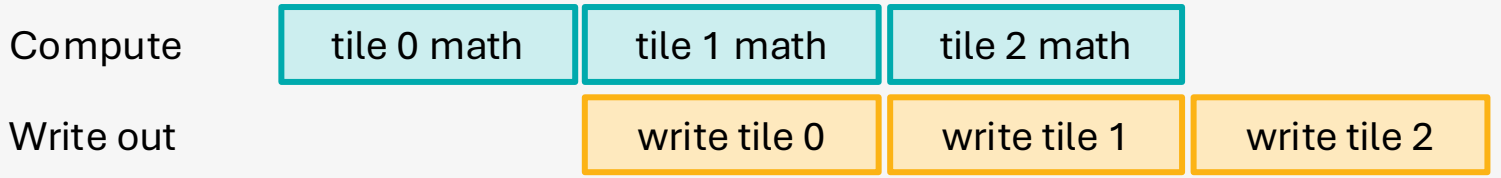
Round 2 has only 28 blocks: 104 SMs wait. Each block also redoes its setup for one tile.

Solution

Normal launch: one block per tile, run in rounds



Persistent kernel: 132 blocks stay put and take tiles one by one



Stream-K: cut tiles into smaller pieces so all SMs finish together

Takeaway: launch one block per SM and let it loop over tiles: no idle round, and setup runs once. CUTLASS and DeepGEMM do this.

Osama et al., Stream-K, PPoPP 2023; NVIDIA CUTLASS 3.x, 2024

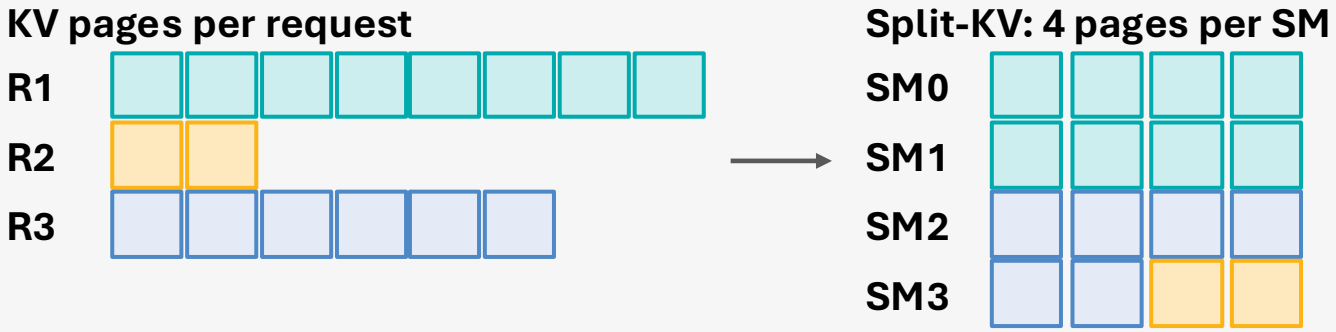
Split-KV balances decode across SMs

Problem

Decode has one query token per request, so splitting only over batch × heads leaves SMs idle.

KV lengths differ by 100× across requests, and their pages sit scattered in HBM.

Solution



Merge partial (m, ℓ, O) per request, as in online softmax

Paged KV as block-sparse rows

Fixed grid: CUDA Graph-safe

Takeaway: FlashInfer (MLSys 2025 Outstanding Paper) pairs block-sparse KV with JIT-specialized kernels and this scheduler; vLLM and SGLang use it.

Dao et al., Flash-Decoding, 2023; Ye et al., FlashInfer, MLSys 2025

ADVANCED — skim, or read more on your own

CUDA Graphs: record the launches once, replay them with one call

Problem

Decoding one token runs dozens of small kernels.

Each launch costs a few µs of CPU and driver time, so the GPU waits between kernels.

Solution

Normal stream: one CPU launch per kernel



CUDA Graph: one call replays all of them



no gaps, done sooner

```
cudaStreamBeginCapture(s, mode);   // record one step
  …launch the kernels…;  cudaStreamEndCapture(s, &g);
cudaGraphInstantiate(&e, g, 0);  cudaGraphLaunch(e, s);
```

Takeaway: capture a decode step once and replay it every token; vLLM and SGLang do this.

Programmatic Dependent Launch (PDL) starts the next kernel early

Problem

In a stream, kernel B launches only after kernel A has fully drained and flushed memory.

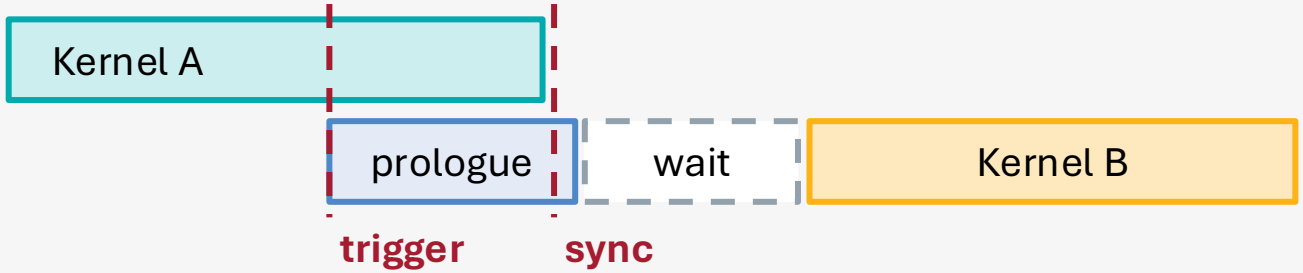
Decode runs many µs-scale kernels per token, so launch latency, prologue and each kernel's tail add up.

Solution

Default stream order



With PDL




```
A: cudaTriggerProgrammaticLaunchCompletion();
B: cudaGridDependencySynchronize();
```

Takeaway: PDL needs Hopper or newer (SM 9.0); it cuts per-kernel gaps in decode, and works inside CUDA Graphs.

A megakernel runs the model in one launch

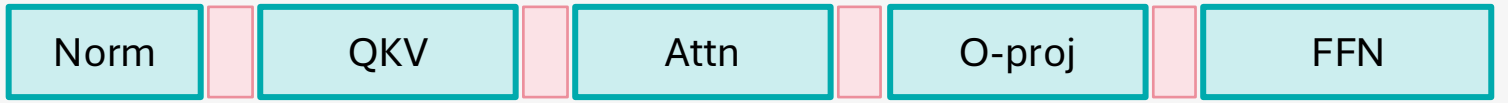
Problem

Batch-1 decode of Llama-3.2-1B runs ~7 kernels per layer × 16 layers for every token.

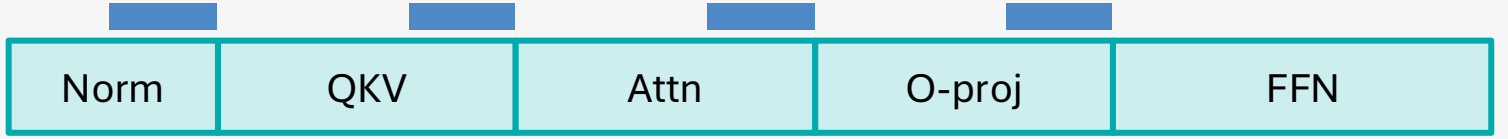
Each boundary drains the GPU before the next kernel loads weights; vLLM and SGLang use ~50% of H100 bandwidth.

Solution

Kernel per op: bubbles at every launch boundary



Megakernel: next op's weights load during the current op



On-GPU interpreter

- Per-SM instruction queue
- Paged SMEM for weights
- Global counters for dependencies

Takeaway: the Hazy Research megakernel reaches 78% of H100 bandwidth at batch 1, ~1.5× SGLang per-user throughput. Mirage MPK compiles such task graphs automatically.

Fused collectives hide TP communication

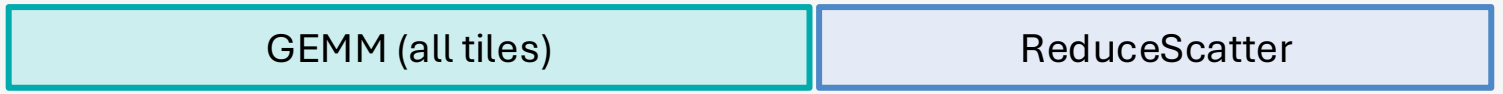
Problem

Tensor parallelism splits each layer across 8 GPUs, adding an AllGather before and a ReduceScatter after every sharded GEMM.

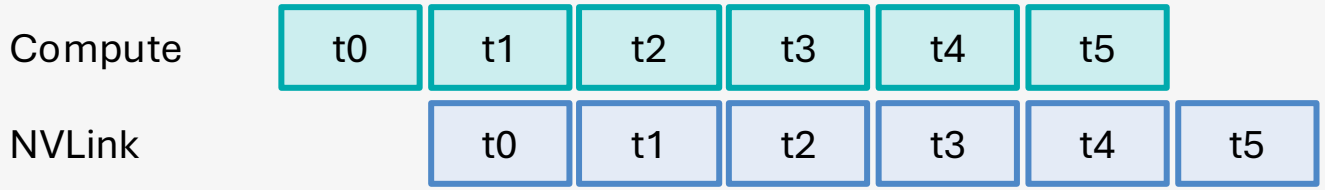
Run as separate kernels, the GEMM waits for the whole collective.

Solution

Separate kernels: GEMM, then ReduceScatter



Fused: each tile's epilogue writes to its owner GPU



AllGather + GEMM: tiles start as their rows arrive

Peer writes via NVLink or NVSHMEM, no extra kernel

Takeaway: Flux overlaps up to 96% of the communication and runs up to 1.66× faster prefill and 1.30× faster decode than vLLM on 8 GPUs.

Chang et al., FLUX, arXiv:2406.06858, 2024

MoE all-to-all hides behind compute

Problem

Each MoE layer sends every token to its top-8 experts on other GPUs, then gathers the outputs back.

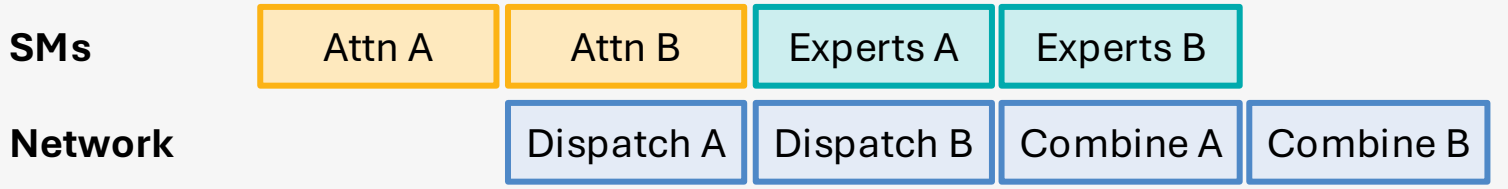
Run back to back, dispatch and combine leave Tensor Cores idle, and NCCL kernels take SMs from compute.

Solution

Serial: dispatch, experts, combine, then the next layer



DeepEP: two micro-batches overlap



Hook-based transfers run on the NIC and use no SMs



Takeaway: DeepEP sustains ~153 GB/s dispatch over NVLink within a node and ~43 GB/s over RDMA across nodes, close to link peak.

DeepSeek-AI, DeepEP, 2025; DeepSeek-V3, arXiv:2412.19437

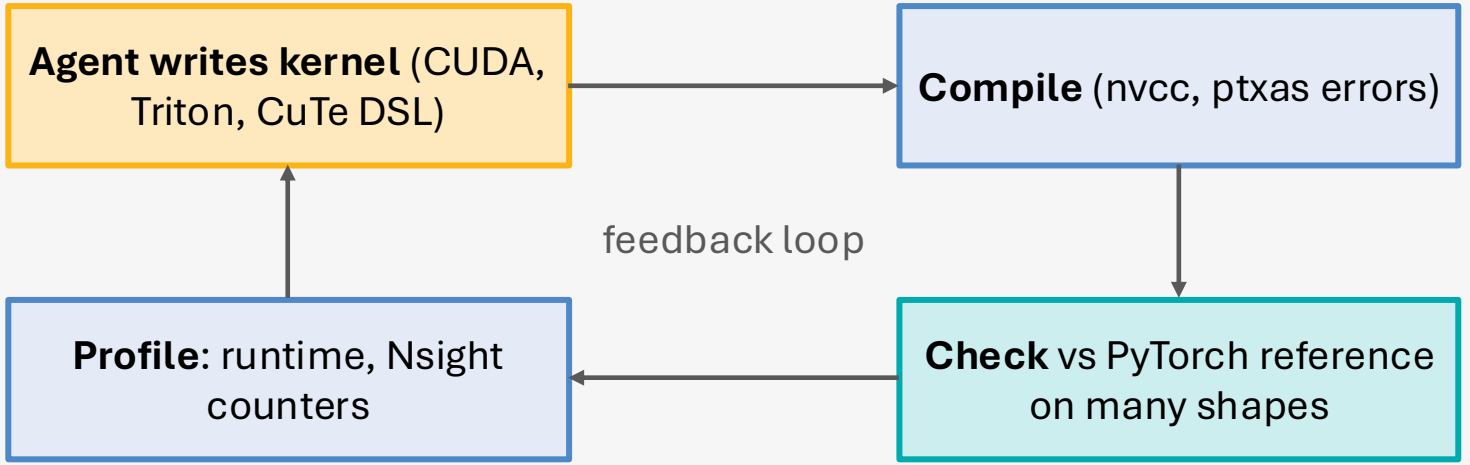
Agents become the new compiler

Problem

Each GPU generation (TMA, TMEM, FP4) and each attention variant needs new hand-tuned kernels, and few engineers write them.

Generated kernels can look fast by passing a weak test or exploiting the harness.

Solution



fast$_p$: share of tasks that are correct and more than p× faster than PyTorch. Static checks reject racy schedules.

Ouyang et al., KernelBench, 2025; AutoMegaKernel, arXiv:2606.09682, 2026

Kernel advances: what to take away

1 Inside a kernel: feed the Tensor Cores. Fuse epilogues, scale per block in FP8/FP4, split warps into producers and consumers, and move work off the slowest unit.

2 Across SMs and launches: remove idle time. Persistent CTAs and split-KV balance the SMs; PDL and megakernels close the gaps between kernels.

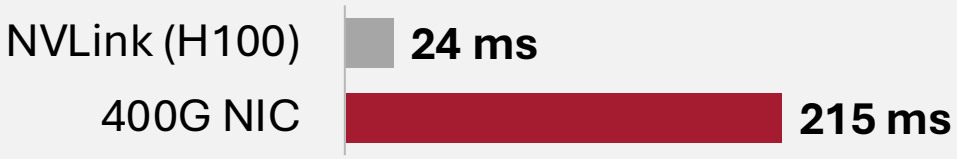
3 Across GPUs: hide communication. Fuse tensor-parallel collectives into GEMM tiles and overlap MoE all-to-all with expert compute.

4 Writing kernels: automate the search. Agents write and profile candidate kernels; correctness checks on many shapes decide which survive.

Challenges in AI hardware

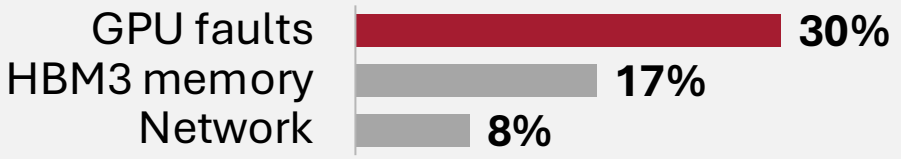
Networking — 9× slower off-node

KV-cache handoff, 32K-token Llama 3 70B prompt



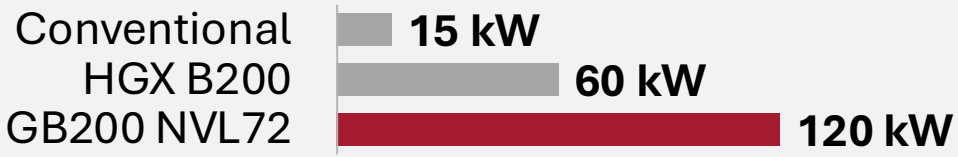
Reliability — 1 failure every 3 hours[1]

Causes of 419 interruptions, Llama 3 training



Power — 8× a conventional rack

Power per rack[2]



Cooling and water

GB200 NVL72 racks require liquid cooling[2]

700,000 L
freshwater to train GPT-3[3]

4.2–6.6B
m^3 withdrawn by AI, 2027E[3]

[1]The Llama 3 Herd of Models [2]NVIDIA HGX Platform: Data Center Physical Requirements Guide
[3]Making AI Less "Thirsty"

Next class

Optional reading

Optimize GPU code

Optimizing GPU code: four goals that combine in one kernel

Move less data

Fewer bytes moved per result

e.g. Coalescing, tiling, fusion

Hide waiting time

Keep units busy while loads finish

e.g. Async copies, warp specialization

Expose more parallelism

Give every SM enough independent work

e.g. Split-K, independent accumulators

Use faster instructions

Same math on cheaper hardware paths

e.g. Tensor Cores, lower precision

They stack: a fast GEMM coalesces loads, double-buffers tiles and uses Tensor Cores.

Optimize memory traffic: move fewer bytes, reuse each one more

Technique	Core idea	Example
Memory coalescing	Neighboring threads read contiguous addresses	Thread i reads x[i], not x[32·i]
Shared-memory tiling	Load a block once; cooperating threads reuse it	Matrix tiles reused across many outputs
Register tiling	Each thread computes several outputs from registers	Several accumulators per thread
Kernel fusion	Consume intermediates at once, skipping the HBM round trip	Matmul + bias + activation in one kernel
Padding / swizzling	Change shared-memory layout to avoid bank conflicts	Transpose tile 32×32 → 32×33
Cache-aware ordering	Schedule nearby blocks so they reuse cached data	Group output tiles that read the same inputs

Hide waiting and expose parallelism: keep every SM issuing work

Technique	Core idea	Example
Async copies + double buffering	Load the next tile while computing on the current one	Alternate two shared-memory buffers in a GEMM
Instruction-level parallelism	Independent instructions issue while earlier ones wait	Several accumulators, not one dependency chain
Warp specialization	Different warps take different pipeline roles	Producer warps load; consumer warps run MMAs
Split-K / split reductions	Spread a long reduction over more workers, then merge	Split a long KV sequence across blocks in decode
Warp-level communication	Exchange values register to register in a warp	Shuffle-based warp sum or max

> **The cost:** split-K adds a merge step; pipelining adds buffers and syncs.

Cheaper computation: run the same math on faster instructions

Technique	Core idea	Example
Tensor Core mapping	Shape matrix ops into tiles and layouts the hardware supports	MMA instructions instead of scalar FMAs
Lower precision	Fewer bytes stored and moved; faster math where supported	FP16/BF16 inputs, FP32 accumulation
Unrolling + specialization	Fix structure at compile time; drop runtime work	Kernel built for one head dim or mask type
Less branch divergence	Keep threads in a warp on the same path	Group tasks with similar control flow
Autotuning	Measure variants for each workload shape	Search tile size, warp count, pipeline depth

Two separate choices: Tensor Cores need fitting shapes; precision must stay accurate.