PART III · CHAPTER 7

GPU Kernels: CUDA, HIP & Triton

From Thread Hierarchy to Tensor Cores — Writing and Optimizing the Compute Primitives That Power Every AI Model

Book: AI/ML Infrastructure from Silicon to Scale | **Author:** Venkat Vinjam | **Cross-ref:** *Roofline (Ch 1) · HBM Hierarchy (Ch 4) · FlashAttention concept (Ch 4) · Inference kernels (Ch 6) · Quantization (Ch 8) · Networking/NCCL (Ch 14)*

> *"A GPU kernel is a contract between the programmer and the hardware. Honor the hardware's preferences — aligned memory, full warps, tiled access — and the hardware rewards you with near-peak throughput. Violate them, and you pay in idle cycles."*
>
> — GPU Performance Engineering Maxim

Every TFLOP of AI computation ultimately executes as a GPU kernel — a function launched across thousands of parallel threads that reads from HBM, computes in registers and SMEM, and writes results back. The difference between a 15% MFU training run and a 65% MFU run almost always traces to *kernel quality*: how well the execution maps to the hardware's parallel execution model.

This chapter builds the complete mental model from first principles: why does naive GEMM achieve only 0.6% compute utilization despite correct arithmetic? Why does tiling recover 100x? How do Tensor Cores deliver 30x more throughput per cycle? And how does a Triton kernel written in 40 lines of Python approach cuBLAS efficiency through automatic compilation? Every answer connects directly to measurable infrastructure cost. A kernel running at 40% MFU instead of 70% costs 75% more per token at identical hardware spend.

FOUNDATIONAL

- Explain the grid/block/warp/thread hierarchy and hardware mapping
- Understand why memory coalescing is the #1 performance factor
- Describe what a Tensor Core does and why it requires aligned data
- Know the occupancy-vs-performance tradeoff intuitively

INTERMEDIATE

- Calculate arithmetic intensity and locate any kernel on the roofline
- Write and analyze a tiled GEMM in CUDA and equivalent Triton
- Use Nsight Compute to identify the binding bottleneck
- Quantify CUDA Graph speedup for decode-phase inference

PRINCIPAL LEVEL

- Design the full kernel optimization strategy for a new LLM architecture
- Choose among cuBLAS/CUTLASS/Triton/raw CUDA for any custom operator
- Explain FlashAttention v3's WGMMA+TMA pipeline with H100 specifics
- Quantify cost savings from kernel optimization at fleet scale

7.1 The GPU Execution Model — From Grid to Instruction

7.1.1 The Three-Level Thread Hierarchy

CUDA programs express parallelism through a three-level hierarchy that maps directly to hardware. Understanding this mapping is the prerequisite for all kernel optimization — every performance technique is ultimately about aligning computation to this hierarchy efficiently.

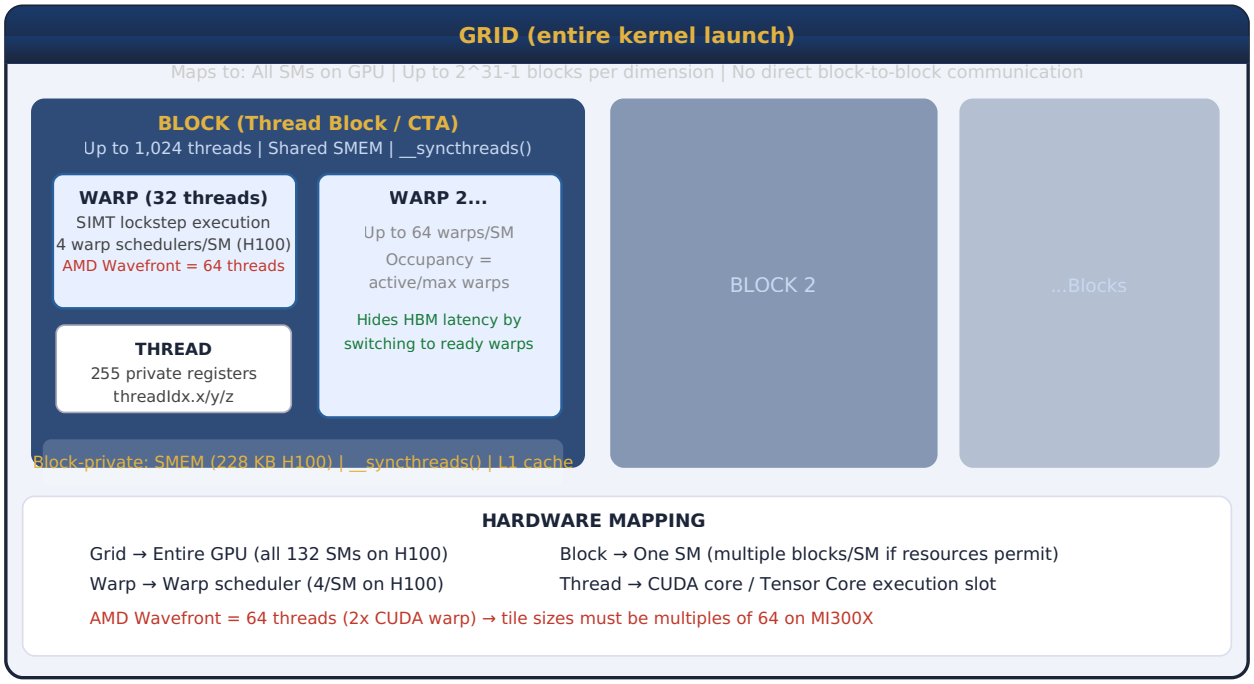


Figure 7.1 *— CUDA thread hierarchy and hardware mapping. Grid maps to all SMs; block maps to one SM; warp (32 threads) is the actual scheduling unit. AMD's Wavefront of 64 threads requires adjusted tile sizing in HIP kernels.*

7.1.2 The SM Execution Engine — H100 Architecture

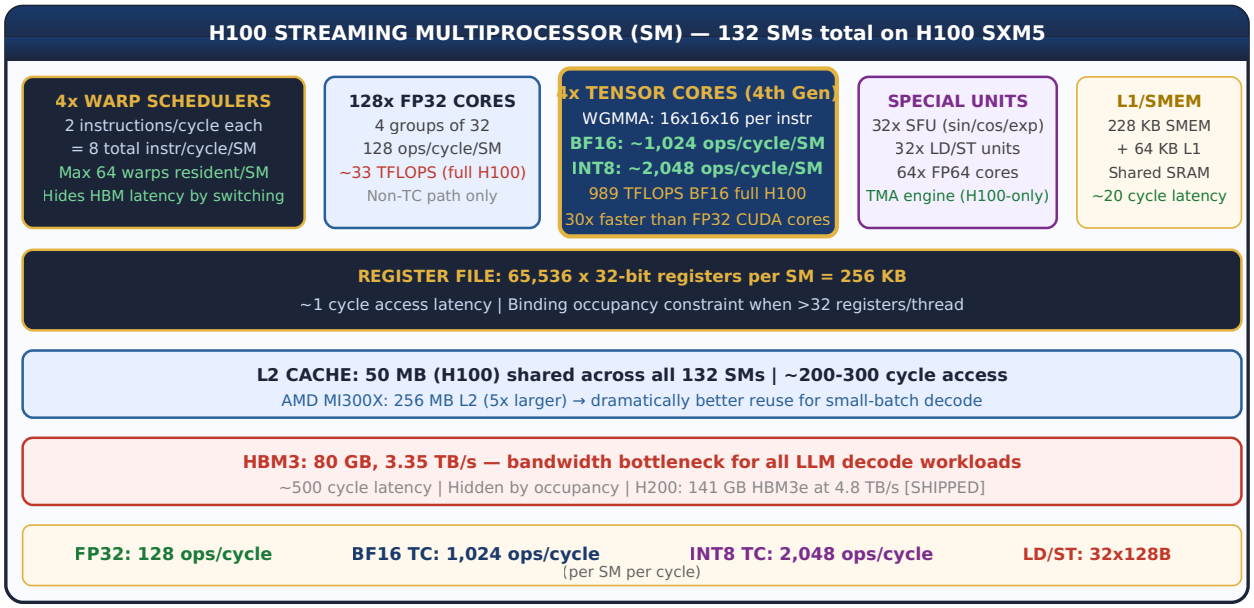


Figure 7.2 *— H100 SM internal architecture. Tensor Cores deliver 30x more throughput than FP32 CUDA cores for matrix multiply — the economic foundation of LLM training. SMEM is software-managed (programmer-controlled); L1 is hardware-managed. HBM at ~500 cycle latency is hidden by switching between resident warps.*

> **Why Tensor Cores Are the Economic Foundation of LLM Training**
>
> Without Tensor Cores, H100 delivers ~33 TFLOPS FP32. With Tensor Cores in BF16: 989 TFLOPS — a 30x increase for matrix multiply. A 70B training run requires ~2.1×10^{24} FLOPs. At 33 TFLOPS: 6,400 GPU-days. At 989 TFLOPS with 60% MFU: ~230 GPU-days. That 28x difference is $15M vs $530,000 at $3/GPU-hour on 256 GPUs. Tensor Cores are not a feature; they are why modern LLM training is economically feasible.

7.1.3 SIMT Execution and Warp Divergence

All 32 threads in a warp execute the same instruction simultaneously (SIMT — Single Instruction, Multiple Thread). When threads take different code paths, the warp serializes both branches with masked inactive threads. This is warp divergence, and it is the mechanism that kills performance in branchy kernels.

```
WARP DIVERGENCE — cost model:

// BAD: 50/50 split (both branches execute sequentially — 2x slower)
if (threadIdx.x < 16) {
    result = heavy_compute_A();  // Threads 0-15 active, 16-31 masked (wasted)
} else {
    result = heavy_compute_B();  // Threads 16-31 active, 0-15 masked (wasted)
}
// Cost: latency(A) + latency(B) — no overlap

// GOOD: Warp-uniform condition (all 32 threads take same path)
if (blockIdx.x < N/2) { result = heavy_compute_A(); }

LLM DIVERGENCE SCENARIOS:
  Causal attention mask: upper triangle is zero — per-element branching if naive
  Flash Attention FIX:   process in TILE*TILE blocks, apply mask to entire tile
                         — warp-uniform operation, zero divergence
  Variable seq lengths:  pad to power-of-2 bucket → uniform per batch
```

7.1.4 Occupancy — The Hardware Utilization Multiplier

Occupancy is the ratio of active warps to the maximum number of warps an SM can support. High occupancy hides HBM latency: while one warp waits 500 cycles for data, the scheduler switches to a ready warp. But more occupancy is not always better — large tiles with lower occupancy often outperform small tiles with maximal occupancy.

```
OCCUPANCY CALCULATION (H100, typical GEMM kernel):

H100 SM limits:
  Max warps/SM: 64  |  Max threads/SM: 2,048  |  Registers/SM: 65,536  |  SMEM: 228 KB

Kernel: blockSize=256 threads (8 warps), 32 regs/thread, 32 KB SMEM/block

  Register check: 65,536 / (32 regs * 32 threads) = 64 warps → 100% occupancy
  SMEM check:     228 KB / 32 KB = 7 blocks * 8 warps = 56 warps → 87.5%  ← BINDING
  Block limit:    min(32, 7) = 7 blocks * 256 threads = 1,792 / 2,048 = 87.5%

Final occupancy = 87.5%  (SMEM is the binding constraint)

TO IMPROVE: reduce SMEM to 28 KB → 8 blocks → 100% occupancy
RISK:       smaller tile = more HBM round-trips = often SLOWER overall
RULE:       always measure end-to-end throughput, not just occupancy
```

7.2 Writing and Understanding GEMM Kernels

GEMM (General Matrix-Matrix Multiply) constitutes 85–95% of compute time during LLM training prefill and 70–90% during decode for large models. Understanding how it maps to GPU hardware from first principles is the single most valuable kernel knowledge a Principal Performance Engineer can possess.

7.2.1 Why Naive GEMM Fails — The Arithmetic Intensity Argument

```
NAIVE GEMM ANALYSIS: C[M*N] = A[M*K] * B[K*N], M=N=K=4096

Each thread computes ONE output: loops K, reads A[row][k] and B[k][col] from HBM every iteration.

FLOPs      = 2 * 4096^3            = 137 GFLOP
HBM reads  = 2 * 4096^3 elements * 4 bytes (FP32) = 549 GB

Arithmetic Intensity = 137 GFLOP / 549 GB = 0.25 FLOP/byte
H100 ridge point     = 989 TFLOPS / 3.35 TB/s = 295 FLOP/byte

0.25 << 295 → DEEPLY MEMORY BOUND (1,180x below compute regime)

Time (HBM bound): 549 GB / 3,350 GB/s = 0.164 s
Time (compute):   137 GFLOP / 100,000 GFLOPS = 0.001 s
Compute utilization = 0.001 / 0.164 = 0.6%

99.4% of time: stalled waiting for HBM data.
This is why naive GEMM is a teaching artifact — never used in production.
```

7.2.2 Tiled GEMM — The Fix

Load a tile of A and B into SMEM, compute partial results entirely in registers, then advance to the next tile. This is tiling (blocking) — the algorithm behind every production GEMM implementation.

```
TILED GEMM ANALYSIS (TILE_SIZE=32, M=N=K=4096, BF16):

Tile loads 32x32 elements of A and B into SMEM (shared across 32 threads each).
Accumulator stays in registers across all K/32=128 tile iterations.

HBM traffic per output element:
  Naive: K reads of A-row + K reads of B-col = 4096+4096 = 8,192 reads
  Tiled: K/TILE_SIZE tile iterations, each element shared across TILE_SIZE threads
         = 128 tiles * 2 elements / 32 threads = 8 reads → 1,024x less traffic

Arithmetic Intensity (BF16 tiled):
  FLOPs = 137 GFLOP (unchanged)
  HBM   = 2 * 4096^2 * 2 bytes (BF16, each weight loaded once) = 67 GB
  AI    = 137 GFLOP / 67 GB = 2,045 FLOP/byte

  2,045 >> 295 FLOP/byte ridge point → COMPUTE BOUND ✓

SMEM check: 2 tiles * 32 * 32 * 2 bytes = 4 KB per block << 228 KB → 57 blocks/SM fit
```

7.2.3 Tensor Cores — The Real GEMM Engine SHIPPED

```
TENSOR CORE INSTRUCTION (H100 WGMMA, BF16):

Operation:  D[16x16] += A[16x16] * B[16x16]  (BF16 input, FP32 accumulation)
Hardware:   4,096 multiply-adds per instruction (dedicated circuitry)
Throughput: 4,096 MACs * 132 SMs / ~6 cycles / 1.98 GHz = 989 TFLOPS

vs FP32 CUDA cores:  128 ops/SM/cycle * 132 * 1.98 GHz = 33 TFLOPS
Tensor Core speedup: 989 / 33 = 30x for matrix multiply

ALIGNMENT REQUIREMENTS:
  Matrix dimensions:    multiples of 16 (BF16), multiples of 128 (FP8)
  Leading dimension:    multiples of 8 (cache-line aligned for BF16)
  Safe LLM convention:  d_model multiples of 64

FAILURE MODE (silent 30x regression):
  Non-aligned dim → Tensor Core falls back to FP32 CUDA core path
  Kernel is "correct" but runs 30x slower — undetectable without profiling!

VERIFY WITH NCU: sm__pipe_tensor_cycles_active.avg.pct_of_peak
  >60% → Good TC usage   |   <30% → Alignment problem or wrong dtype
```

The Hidden Alignment Tax — FFN Intermediate Dimensions

SwiGLU FFN uses intermediate dim = 8/3 * d_model. For d_model=8192: 8/3*8192 ≈ 21,845 — NOT divisible by 16. This silently disables Tensor Cores on all FFN kernels, regressing ~36% of model compute by 30x. Fix: round to nearest multiple of 64 (21,888 works). Verify every custom dimension before deployment.

7.2.4 CUTLASS — NVIDIA's Templated GEMM Library SHIPPED

```
CUTLASS 3.x HIERARCHY:
  ThreadblockTile (128x128x64) → one CUDA block, one SM
    WarpTile (64x64x64)        → one warp's Tensor Core work
      InstructionTile (16x16x16) → one WGMMA hardware instruction

H100 3-STAGE SOFTWARE PIPELINE:
  Stage N-1: TMA loads A+B tiles from HBM → SMEM  (async, zero SM cycles)
  Stage N:   WGMMA executes on tile N             (Tensor Cores full speed)
  Stage N+1: TMA pre-fetches tile N+1             (overlaps Stage N compute)
  Result: Memory latency completely hidden → 75-85% MFU for aligned GEMM

USE CUTLASS WHEN:
  Custom epilogue: fuse quantization/scale/bias-inject after GEMM output
  Unusual dtypes:  FP8 mixed precision with custom per-tensor descaling
  Non-standard shapes: batch=1 decode GEMM (M=1) needing custom tile choices
```

7.3 Memory Access Patterns and CUDA Graphs

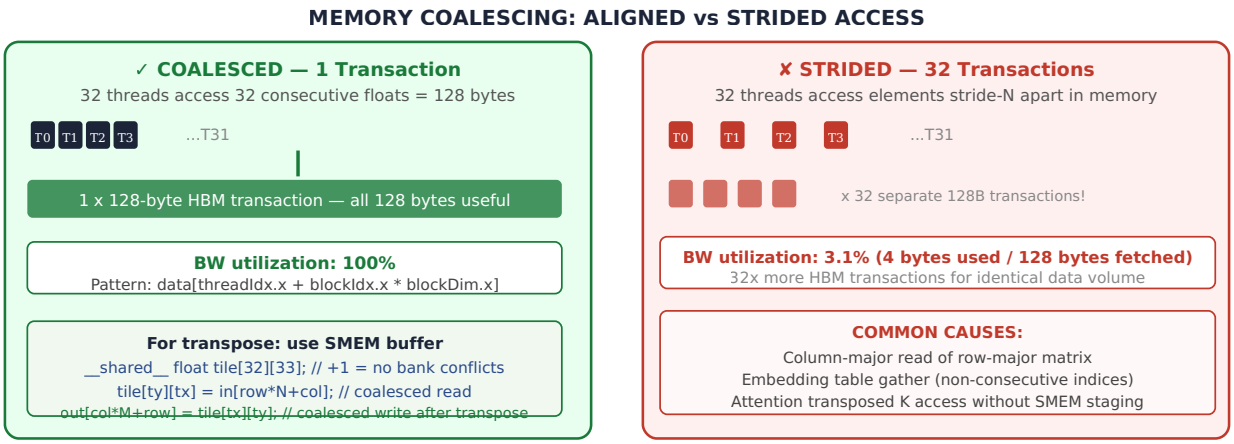


Figure 7.4 *— Memory coalescing: aligned vs. strided access. Coalesced: 32 threads, 32 consecutive floats, 1 HBM transaction, 100% bandwidth utilization. Strided: 32 separate 128-byte transactions each carrying 4 useful bytes — 3.1% efficiency. The transpose fix uses SMEM with +1 column padding to eliminate bank conflicts while keeping both reads and writes coalesced.*

```
CUDA GRAPHS — Eliminating Kernel Launch Overhead

PROBLEM: Each standard CUDA kernel launch costs ~5 microseconds of CPU overhead.
  LLaMA-3 70B decode: 6 kernels/layer * 80 layers = 480 launches per step
  Total overhead: 480 * 5us = 2.4ms

  At TPOT=25ms: 9.6% overhead   |   At TPOT=10ms: 24% — launch overhead DOMINATES!

SOLUTION:
// CAPTURE (once at startup, ~100-200ms cost):
cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
  launch_qkv_gemm(stream, ...);         // Recorded, NOT executed
  launch_flash_attention(stream, ...);  // Recorded
  // ... all 480 kernels
cudaStreamEndCapture(stream, &graph);
cudaGraphInstantiate(&graphExec, graph, nullptr, nullptr, 0);

// REPLAY every decode step (~0.024ms overhead):
cudaGraphLaunch(graphExec, stream);

Before: 2.4ms launch overhead   After: 0.024ms (100x reduction)
TPOT improvement: 10-30% on decode workloads

LIMITATION: Graph is STATIC — batch size and tensor shapes cannot change
SOLUTION:   Capture one graph per (batch_size, seq_len) configuration (vLLM approach)
```

7.4 HIP — CUDA on AMD Hardware

HIP (Heterogeneous Interface for Portability) is AMD's CUDA-equivalent programming model for ROCm. The API is ~95% identical to CUDA; the performance expertise lies in understanding the hardware differences.

```
CUDA → HIP KEY TRANSLATIONS:
  cudaMalloc()           → hipMalloc()           (identical)
  __syncthreads()        → __syncthreads()       (identical)
  __shfl_xor_sync(mask)  → __shfl_xor()          (HIP has no mask arg)
  cudaGraph*             → hipGraph*              (identical)
  #include <cuda_runtime.h>  →  #include <hip/hip_runtime.h>
  nvcc kernel.cu         → hipcc kernel.hip
  AUTO-MIGRATION: hipify-clang source.cu --output-dir hip_src/ (~95% coverage)

CRITICAL HARDWARE DIFFERENCES (H100 vs MI300X):
  Feature              H100 SXM5           MI300X             Impact
  ──────────────────────────────────────
  Warp/Wavefront       32 threads           64 threads          TILE multiples of 64!
  SMEM per unit        228 KB/SM            64 KB/CU            Smaller tiles on AMD
  L2 Cache             50 MB                256 MB              5x better reuse
  HBM                  80 GB, 3.35 TB/s     192 GB, 5.3 TB/s    MI300X wins both
  Tensor Op            WGMMA (16x16x16)     MFMA (16x16x4)      Different intrinsics
  Reduction loop steps 5 (log2 32)          6 (log2 64)         Add one HIP iteration
```

7.5 Triton — Python-Level GPU Kernel Programming

Triton (OpenAI, 2019) allows writing high-performance GPU kernels in Python-like syntax. Unlike CUDA — which programs at the thread level — Triton programs at the *tile level*: you describe operations on tiles, and the compiler manages thread assignment, SMEM staging, and memory coalescing automatically.

> **Why Triton Is the Primary Tool for Custom AI Kernels**
>
> Meta's KernelEvolve, OpenAI's custom attention kernels, and HuggingFace's kernel library all use Triton. It achieves 85–95% of cuBLAS throughput with 10–20x less code. `tl.dot()` automatically lowers to WGMMA Tensor Core instructions — no manual TC intrinsics required.

```
// TRITON PROGRAMMING MODEL — vector addition:
@triton.jit
def add_kernel(x_ptr, y_ptr, out_ptr, n_elements, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(axis=0)                 # Which tile? (like blockIdx.x)
    offsets = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask)      # Auto-coalesced HBM load
    y = tl.load(y_ptr + offsets, mask=mask)      # Stays in registers
    tl.store(out_ptr + offsets, x + y, mask=mask)

# No threadIdx, no __syncthreads, no warp management needed.
# Triton compiles tile operations to optimal CUDA/HIP with auto SMEM staging.
```

7.5.2 Triton GEMM with Auto-Tuning

```
@triton.autotune(
    configs=[
        triton.Config({'BLOCK_M': 128, 'BLOCK_N': 256, 'BLOCK_K': 64,  'num_stages': 3}),
        triton.Config({'BLOCK_M': 128, 'BLOCK_N': 128, 'BLOCK_K': 32,  'num_stages': 4}),
        triton.Config({'BLOCK_M': 64,  'BLOCK_N': 256, 'BLOCK_K': 32,  'num_stages': 4}),
    ],
    key=['M', 'N', 'K'],
)
@triton.jit
def matmul_kernel(A, B, C, M, N, K, sa_m, sa_k, sb_k, sb_n, sc_m, sc_n,
                  BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr):
    pid_m, pid_n = tl.program_id(0), tl.program_id(1)
    offs_m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
    offs_n = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
    offs_k = tl.arange(0, BLOCK_K)
    acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)  # Accumulator in registers!
    a_ptrs = A + offs_m[:,None]*sa_m + offs_k[None,:]*sa_k
    b_ptrs = B + offs_k[:,None]*sb_k + offs_n[None,:]*sb_n
    for k in range(0, K, BLOCK_K):
        a = tl.load(a_ptrs, mask=offs_k[None,:] < K-k, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:,None] < K-k, other=0.0)
        acc += tl.dot(a, b)   # ← compiles to WGMMA Tensor Core instructions!
        a_ptrs += BLOCK_K*sa_k;  b_ptrs += BLOCK_K*sb_k
    c_ptrs = C + offs_m[:,None]*sc_m + offs_n[None,:]*sc_n
    tl.store(c_ptrs, acc.to(tl.bfloat16), mask=(offs_m[:,None]<M)&(offs_n[None,:]<N))

# SMEM budget: num_stages*(BLOCK_M+BLOCK_N)*BLOCK_K*2 bytes <= 228 KB (H100)
# Performance: 85-95% of cuBLAS for aligned shapes with BLOCK_M=128, BLOCK_N=128
```

7.5.4 Fused Kernels — The Primary Triton Use Case

```
// FUSED RMSNorm + Linear — eliminates one HBM round-trip:
// Unfused: Read X → Write X_norm → Read X_norm → Write Y  (4 HBM ops)
// Fused:   Read X → Write Y                                (2 HBM ops)

@triton.jit
def fused_rmsnorm_linear(X_ptr, W_norm_ptr, W_lin_ptr, Y_ptr, B, T, D, outD, eps,
                         BLOCK_D: tl.constexpr, BLOCK_OUT: tl.constexpr):
    b, t = tl.program_id(0), tl.program_id(1)
    offs_d = tl.arange(0, BLOCK_D)
    x = tl.load(X_ptr + (b*T+t)*D + offs_d, mask=offs_d < D)  # ONE HBM read
    rms = tl.sqrt(tl.sum(x*x, 0) / D + eps)                  # In registers
    x_normed = x / rms * tl.load(W_norm_ptr + offs_d, mask=offs_d < D)
    offs_out = tl.arange(0, BLOCK_OUT)
    W = tl.load(W_lin_ptr + offs_d[:,None]*outD + offs_out[None,:], ...)
    y = tl.sum(x_normed[:,None] * W, axis=0)
    tl.store(Y_ptr + (b*T+t)*outD + offs_out, y, ...)          # ONE HBM write

# Eliminates 2 HBM round-trips (write X_norm + read X_norm) → 1.5-2x speedup
```

7.6 Key LLM Kernels in Production

Kernel	Type	Bottleneck (Prefill)	Bottleneck (Decode)	~% Time	Primary Optimization
QKV Projection GEMM	GEMM	Compute	Memory BW	25%	Tensor Core efficiency, FP8
Flash Attention	Custom	Compute	Memory BW	20%	FAv3, GQA, FP8 mode
Output Projection	GEMM	Compute	Memory BW	8%	Standard optimized GEMM
FFN Gate + Up GEMMs	GEMM	Compute	Memory BW	24%	Fuse with SiLU activation
FFN Down GEMM	GEMM	Compute	Memory BW	12%	Standard optimized GEMM
RMSNorm x2, SiLU, Residual	Elementwise	Memory BW	Memory BW	<1%	Fuse with adjacent GEMM
AllReduce (TP only)	Collective	Network	Network	Variable	Overlap with compute (Ch 14)

The 85% Rule: Optimize GEMM, Not Elementwise Ops

~85% of LLM inference compute time is GEMM kernels. Perfectly optimizing standalone RMSNorm delivers under 1% improvement. One 10% MFU gain in GEMM (70% → 77%) delivers 8.5x more value than the same relative gain in RMSNorm. Always compute expected gain = (fraction_of_runtime) × (speedup − 1) before allocating engineering effort.

7.6.2 Flash Attention v3 — The Most Important Custom Kernel SHIPPED

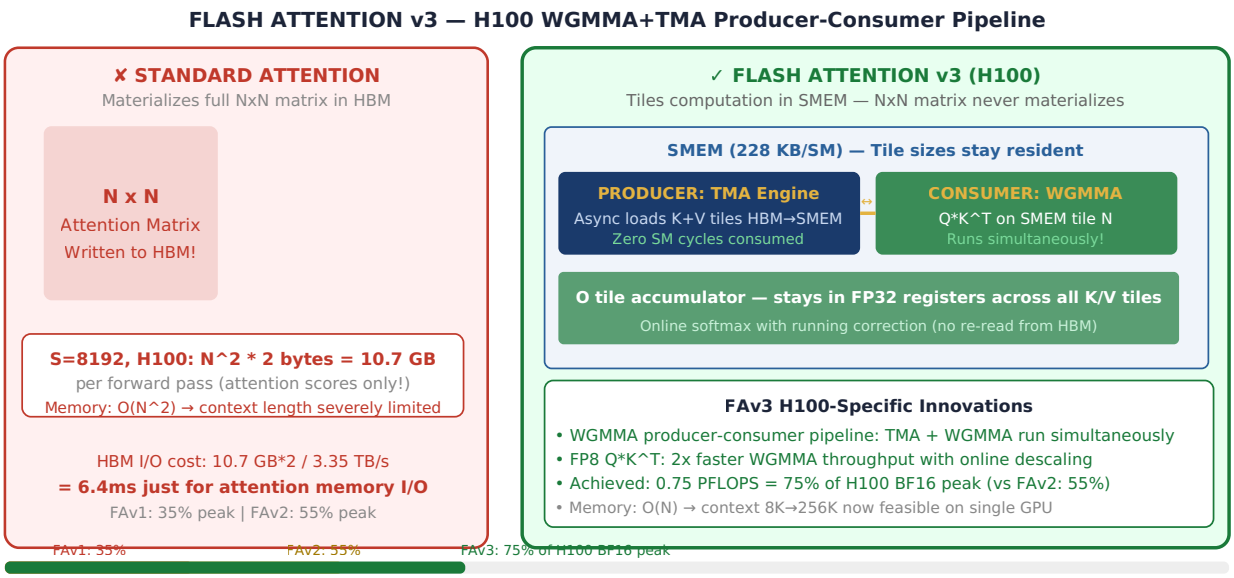


Figure 7.5 *— Flash Attention v3 vs. standard attention. Standard attention materializes an N×N matrix in HBM: at S=8,192 tokens, that is 10.7 GB and 6.4ms of HBM I/O per forward pass. FAv3 tiles Q, K, V computation into SMEM — the result accumulator stays in FP32 registers across all tile iterations. The H100-specific WGMMA+TMA producer-consumer pipeline runs memory loading and compute simultaneously, reaching 75% of BF16 peak.*

7.7 Profiling GPU Kernels with Nsight Compute

7.7.1 The Five-Step NCU Analysis Protocol

```
NCU 5-STEP KERNEL OPTIMIZATION PROTOCOL:

Step 1: BASELINE — establish ground truth
  ncu --set basic --kernel-name "my_kernel" -o baseline python run.py
  Record: TFLOPS achieved, HBM BW%, L2 hit rate, occupancy, wall time

Step 2: ROOFLINE CLASSIFICATION
  ncu --section SpeedOfLight_RooflineChart --kernel-name "my_kernel" python run.py
  Decision tree:
    Point below MEMORY roof   → memory-bandwidth-bound  → fix data movement
    Point below COMPUTE roof  → compute-bound           → fix arithmetic efficiency
    Point near BOTH roofs     → well-optimized; look for the lower ceiling

Step 3: DEEP-DIVE THE BOTTLENECK
  Memory-bound:   ncu --section MemoryWorkloadAnalysis ...
    Check: L1/L2 hit rates, uncoalesced access %, SMEM bank conflicts
  Compute-bound:  ncu --section ComputeWorkloadAnalysis ...
    Check: TC utilization %, warp stall reasons, instruction mix

Step 4: APPLY IN PRIORITY ORDER
  1. Fix uncoalesced memory access          (5-20x speedup)  ← highest ROI
  2. Align shapes / switch to BF16/FP8     (2-30x speedup if hitting CUDA cores)
  3. Increase SMEM tile size               (2-5x speedup)
  4. Tune occupancy / reduce register use  (1.2-2x speedup)
  5. Software pipelining / async copies    (1.3-1.8x speedup)

Step 5: VALIDATE AND ITERATE
  Re-run ncu. Did bottleneck improve? Did resolving it expose the next one?
  Correctness check: compare output tensors numerically against reference.
```

7.7.2 Critical NCU Metrics Reference

```
KEY NSIGHT COMPUTE METRICS FOR AI KERNEL OPTIMIZATION:

TENSOR CORE UTILIZATION:
  sm__pipe_tensor_cycles_active.avg.pct_of_peak
  Target: >60% for GEMM kernels
  Red flag: <30% → wrong dtype, misaligned shape, or not using tl.dot/WMMA API

MEMORY EFFICIENCY:
  dram__bytes_read.avg                        — actual HBM traffic
  Theoretical minimum (decode):              weight_size * 1 (each weight loaded once)
  Red flag: actual > 3x theoretical          ← uncoalesced access or excessive recompute

WARP STALL REASONS (what warps wait for):
  smsp__warp_issue_stall_long_scoreboard     — waiting for HBM data → increase tile
  smsp__warp_issue_stall_barrier            — __syncthreads() overhead → pipeline better
  smsp__warp_issue_stall_mio_throttle       — memory instruction queue full

  long_scoreboard >20 cycles/warp → more arithmetic between memory ops
  barrier         >10 cycles/warp → overlap loads with computation

OCCUPANCY:
  sm__warps_active.avg.pct_of_peak_sustained_active — achieved occupancy %
  launch__occupancy_limit_registers                  — registers are binding
  launch__occupancy_limit_shared_memory             — SMEM is binding
```

7.8 Advanced Kernel Techniques

7.8.1 Persistent Kernels — For Ultra-Low Latency Decode

```
PERSISTENT KERNEL PATTERN (TPOT targets < 5ms):

Standard: Launch new kernel every decode step
  480 launches * 5us = 2.4ms overhead (at TPOT=5ms that is 48% of total time!)

Persistent: ONE kernel launch, loops internally
  Launch once → loop forever → 0 launch overhead on subsequent steps

__global__ void persistent_decode(volatile int* step_counter, KVCache* kv, ...) {
    while (true) {
        while (atomicRead(step_counter) == my_last_step) { /* spin wait */ }
        int step = atomicRead(step_counter);
        execute_decode_step(kv, step, ...);
        atomicStore(completion_counter, step);
    }
}

TRADEOFF: Holds SM slots permanently → reduces scheduler flexibility
BEST FOR: TPOT targets <5ms where launch overhead >20% of total time
ALTERNATIVE: CUDA Graphs (80-90% of persistent kernel benefit, more flexible)
```

7.8.2 Tensor Memory Accelerator (TMA) — H100-Specific SHIPPED

```
TMA: DEDICATED HBM → SMEM DMA ENGINE (H100 ONLY)

WITHOUT TMA (A100 and earlier):
  Compute warps issue LD/ST instructions to copy HBM → SMEM
  During loading: Tensor Cores sit idle → TC utilization = 0% while copying!

WITH TMA (H100):
  TMA hardware unit copies HBM → SMEM asynchronously, consuming ZERO SM cycles
  Compute warps run WGMMA on tile N while TMA loads tile N+1 in background
  Result: near-100% TC utilization — loading is "free" compute time

// Setup: encode tensor layout once
CUtensorMap tensorMap;
cuTensorMapEncodeTiled(&tensorMap, CU_TENSOR_MAP_DATA_TYPE_FLOAT16,
                       2, addr, {M, N}, {TILE_M, TILE_N}, ...);

// In kernel: async tile copy (TMA engine does the work)
__pipeline_memcpy_async(&smem_tile[0][0], &tensorMap, tile_coords);
__pipeline_commit();
wgmma_execute(&output, &smem_prev_A, &smem_prev_B);  // Compute previous tile
__pipeline_wait_prior(0);                              // Wait for new tile

This is the exact hardware mechanism enabling FAv3's 75% peak efficiency.
FAv2 on A100: 55% peak — the 20% gap is entirely explained by missing TMA.
```

7.9 The Kernel Ecosystem — Library Selection Guide

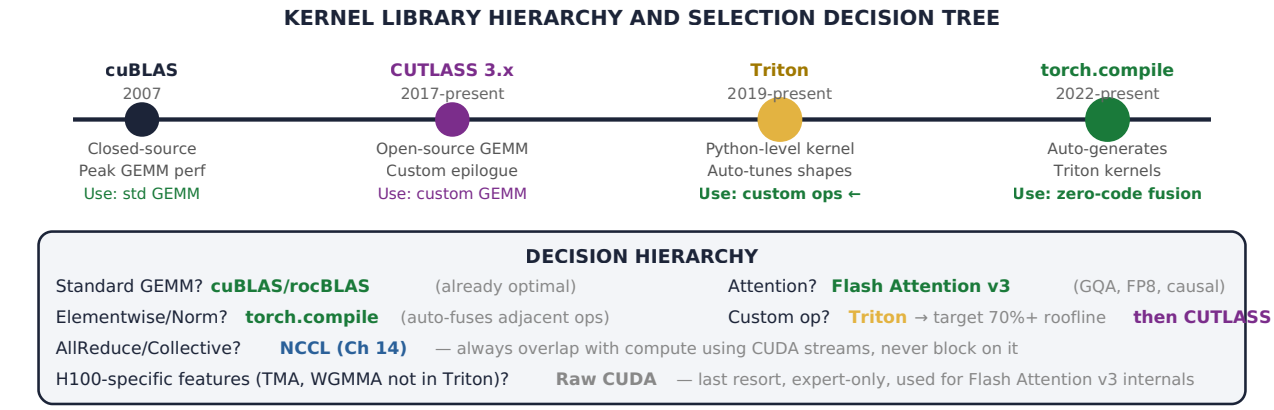


Figure 7.6 *— GPU kernel library hierarchy and selection decision tree. The correct tool depends on operation type. cuBLAS handles standard GEMM. Flash Attention v3 handles attention. torch.compile provides zero-code operator fusion for elementwise ops. Triton is the workhorse for custom operators. CUTLASS when Triton cannot achieve roofline ceiling. Raw CUDA only for H100-specific hardware features not yet in Triton's API.*

7.10 Profiling Cookbook

🔬 **PROFILING COOKBOOK — GPU KERNEL DIAGNOSIS**

```
# ──────────────────────────────────────────────
# SYMPTOM: Training MFU is 40%; target is 65%
# GOAL: Identify the binding kernel-level bottleneck
# ──────────────────────────────────────────────

# STEP 1: Timeline profiling — find the slow kernel
nsys profile \
  --trace=cuda,nvtx \
  --output=training_profile \
  python train.py --max-steps=50
# Open in Nsight Systems GUI. Look for:
# - CPU-GPU gaps (CPU-bound issue or launch overhead)
# - Specific kernels that are 10x longer than neighbors

# STEP 2: Roofline-classify the slow kernel
ncu \
  --section SpeedOfLight_RooflineChart \
  --kernel-name "flash_attention_fwd_kernel" \
  --launch-skip 20 --launch-count 3 \
  -o roofline_analysis python train.py
# Both SOL <60%: occupancy or launch overhead issue
# Compute SOL > Memory SOL: compute-bound → check TC utilization
# Memory SOL > Compute SOL: memory-bound → check coalescing, tile size

# STEP 3: Full deep-dive analysis
ncu \
  --set full \
  --section SpeedOfLight_RooflineChart \
  --section MemoryWorkloadAnalysis \
  --section ComputeWorkloadAnalysis \
  --section WarpStateStatistics \
  --kernel-name "flash_attention_fwd_kernel" \
  --launch-skip 10 --launch-count 1 \
  -o deep_analysis python inference_bench.py

# STEP 4: Query Tensor Core utilization
ncu --import deep_analysis.ncu-rep \
    --query-metrics "sm__pipe_tensor_cycles_active.avg.pct_of_peak"
# <30%: Not using TCs → check dtype (must be BF16/FP16/FP8), alignment (mult of 16)
# 30-60%: TC used but memory-feeding is bottleneck → increase tile size
# >60%: Good TC usage → check if SMEM tiling and pipelining are optimal

# STEP 5: AMD equivalent (omniperf + rocprof)
omniperf profile --name my_kernel -- python model.py
```

```
omniperf analyze -p workload_dir/ --filter-dispatch-id "0:20" --roof-only
# AMD key metrics:
# SQ_INSTS_VALU    → vector ALU ops (SM throughput equivalent)
# TCC_HIT/MISS     → L2 cache hit rate (MI300X target: >80% for small models)
# SQ_WAIT_INST_ANY → stall cycles (equivalent to NCU long_scoreboard)
# MFMA utilization → TC equivalent: omniperf → Compute → MFMA Utilization %

# DECISION TREE SUMMARY:
# TC <30%          → Fix dtype / alignment FIRST     (2-30x speedup)
# Uncoalesced >10% → Fix memory access pattern      (5-20x speedup)
# Occupancy <30%   → Reduce SMEM or registers       (1.5-3x speedup)
# Barrier stalls   → Pipeline loads with compute    (1.3-1.8x speedup)
# HBM >3x minimum  → Increase tile / add SMEM reuse (2-5x speedup)
```

Worked Example: Quantifying the Cost of a Slow Kernel

WORKED EXAMPLE 7.1 — FROM 40% MFU TO 65% MFU: THE FULL COST MATH

Setup

A 512-GPU training cluster running LLaMA-3 70B achieves 40% MFU due to suboptimal kernel configuration: no FP8, misaligned FFN dimensions, Tensor Core utilization at 45%. Target: 65% MFU through three specific kernel interventions. Hardware: 512× H100 SXM5 at $3/GPU-hour (cloud reserved). Training run: 1,000,000 optimizer steps.

Step 1: Baseline Throughput

```
H100 BF16 peak: 989 TFLOPS per GPU
Current achieved: 40% * 989 = 395.6 TFLOPS/GPU
512-GPU aggregate: 512 * 395.6 = 202,547 TFLOPS

Model FLOPs per step (LLaMA-3 70B, seq=4096, batch=2048):
  FLOPs ≈ 6 * P * S * B = 6 * 70e9 * 4096 * 2048 = 3.53 * 10^18 FLOPs

Step time (current):
  t = 3.53e18 / (202,547 * 10^12) = 17.4 seconds/step
```

Step 2: Target Throughput After Optimization

```
Target MFU: 65%
Target throughput: 65% * 989 = 642.9 TFLOPS/GPU
512-GPU aggregate: 512 * 642.9 = 329,165 TFLOPS

Step time (optimized):
  t = 3.53e18 / (329,165 * 10^12) = 10.7 seconds/step

Speedup: 17.4 / 10.7 = 1.63x faster training
```

Step 3: Cost Savings

```
TRAINING RUN COST (1,000,000 steps):

Before optimization:
  Total time = 1e6 * 17.4s = 17,400,000 s = 4,833 hours
  Cost = 4,833 hours * 512 GPUs * $3/GPU-hr = $7,426,560

After optimization:
  Total time = 1e6 * 10.7s = 10,700,000 s = 2,972 hours
  Cost = 2,972 * 512 * $3 = $4,565,760

SAVINGS: $7,426,560 - $4,565,760 = $2,860,800 (~$2.9M saved per run)

THREE INTERVENTIONS DELIVERING THE 40% → 65% MFU JUMP:
  1. Enable FP8 + Transformer Engine:      +8-12% MFU
     (2x GEMM throughput; auto per-layer scaling, zero code changes)
  2. Fix aligned dimensions (mult of 64):  +5-10% MFU
     (TC utilization 45% → 75%+; eliminates silent CUDA core fallback)
  3. torch.compile with max-autotune:      +5-8% MFU
     (fusion of elementwise ops; CUDA Graph capture for fixed shapes)
  Combined expected gain: +18-30% MFU → 58-70% MFU from 40% baseline
```

Step 4: Engineering ROI

```
Engineering investment:
  2 engineers * 6 weeks * $250/hr fully loaded = $240,000

First-run savings: $2,860,800

ROI = ($2,860,800 - $240,000) / $240,000 = 10.9x return in first run alone

At 2 major training runs per quarter: $11.4M/year savings
Kernel optimization at cluster scale is the highest-ROI systems
engineering investment available to a Principal Performance Architect.
```

CHAPTER 7 QUICK REFERENCE — NUMBERS TO KNOW COLD

Parameter	H100 SXM5	MI300X	Performance Impact
BF16 Tensor Core peak	989 TFLOPS	~1,307 TFLOPS	Training throughput ceiling
FP32 CUDA core peak	33 TFLOPS	~41 TFLOPS	Cost of misaligned GEMM
TC vs CUDA core ratio	30x	~32x	Silent 30x regression if not aligned
SMEM per SM/CU	228 KB	64 KB	AMD requires smaller tiles
Register file per SM	65,536 x 32b	~65,536 x 32b	Similar occupancy arithmetic
L2 Cache	50 MB	256 MB	MI300X: 5x reuse advantage
HBM bandwidth	3.35 TB/s	5.3 TB/s	Decode throughput ceiling
Warp / Wavefront	32 threads	64 threads	Tile sizing multiples
Kernel launch overhead	~5 µs	~5 µs	480 launches = 2.4ms/step
CUDA Graph overhead	~0.024ms	~0.024ms	100x vs standard launches
Alignment requirement	Mult of 16 (64 safe)	Mult of 16	Required for TC enablement
FAv3 achieved efficiency	75% of BF16 peak	FAv2: ~60%	Attention throughput ceiling
Triton vs cuBLAS	85-95% parity	85-95% parity	10-20x less code
HBM latency	~500 cycles	~500 cycles	Hidden by occupancy (>50%)

Key Takeaways

CORE CONCEPTS

- The CUDA thread hierarchy (grid → block → warp → thread) maps directly to hardware. The warp (32 threads) is the actual scheduling unit; occupancy is how many warps stay resident to hide ~500-cycle HBM latency.
- Naive GEMM achieves 0.6% compute utilization: arithmetic intensity of 0.25 FLOP/byte is 1,180x below the H100 ridge point of 295 FLOP/byte. Tiling raises AI to ~2,045 FLOP/byte, converting the kernel from deeply memory-bound to compute-bound.

- Tensor Cores deliver 30x more throughput than FP32 CUDA cores for matrix multiply — but only when operands are BF16/FP16/FP8 AND matrix dimensions are multiples of 16. Non-aligned shapes silently fall back to the CUDA core path at 30x regression.
- Triton programs at the tile level, auto-targets Tensor Cores via `tl.dot()`, and achieves 85–95% of cuBLAS throughput. It is the primary tool for custom AI operators including fused elementwise+GEMM sequences.

COST & PRODUCTIVITY IMPACT

- Improving MFU from 40% to 65% on a 512-GPU training run saves ~$2.9M per 1M-step run at $3/GPU-hr. Engineering investment: ~$240K. First-run ROI: 10.9x. Three interventions deliver this: FP8, dimension alignment, and torch.compile.
- CUDA Graphs eliminate 2.4ms of kernel launch overhead per decode step (480 launches × 5µs). At TPOT=10ms that overhead is 24% of total decode time — a 10–30% TPOT improvement with zero changes to the model or kernel implementation.
- Fusing RMSNorm+Linear in Triton eliminates 2 HBM round-trips (write and re-read intermediate X_norm), delivering 1.5–2x speedup on that sequence. Operator fusion is the highest-leverage elementwise optimization pattern.

PRINCIPAL-LEVEL INSIGHT

- 100% occupancy is NOT the optimization goal. A kernel with 50% occupancy and large tiles (high arithmetic intensity) consistently outperforms one at 100% occupancy with small tiles. Always measure end-to-end throughput; occupancy is a diagnostic tool, not the objective.
- Flash Attention v3's 75% peak efficiency (vs FAv2's 55%) comes entirely from H100-specific hardware: the WGMMA+TMA producer-consumer pipeline runs memory loading and compute simultaneously. This is not available on A100. Kernel performance is hardware-generation-specific — every new GPU generation requires re-profiling and re-optimization of critical kernels.
- The kernel selection hierarchy is: cuBLAS → FlashAttention → torch.compile → Triton → CUTLASS → raw CUDA. Each step down adds engineering cost. Most custom operators stop at Triton. Raw CUDA is reserved for hardware-specific features (TMA, WGMMA) not yet exposed in Triton's API — a shrinking set with each Triton release.

Review Questions

FOUNDATIONAL Build the mental model — no math required

1. Explain why warp divergence degrades performance. A causal attention mask sets the upper triangle of the attention matrix to negative infinity. Why does naive attention suffer from warp divergence on this mask, and how does Flash Attention's tiling approach eliminate it?

2. You observe a GPU kernel with 30% occupancy and 25% compute utilization. A colleague says "just increase occupancy to 100% by reducing SMEM usage per block." Why might this make performance *worse* rather than better? What should you measure instead to validate the change?

3. A model architect sets the hidden dimension of a new transformer to d_model=6,000. What kernel performance problem does this create, and how would you detect it using Nsight Compute? What simple change fixes it?

INTERMEDIATE Apply and calculate — production context

1. A CUDA kernel on H100 uses 48 registers/thread and 48 KB SMEM/block, with blockSize=256. H100 limits: 65,536 regs/SM, 228 KB SMEM/SM, max 2,048 threads/SM. (a) Calculate achieved occupancy. (b) Which resource is the binding constraint? (c) What single change yields the largest occupancy improvement, and what is the tradeoff risk?

2. An LLM inference system before CUDA Graphs has TPOT=22ms with 480 kernel launches per decode step. After CUDA Graphs, TPOT drops to 19.5ms. (a) What was the kernel launch overhead before? (b) Is this consistent with the ~5μs/launch number? (c) If TPOT target is 8ms, what fraction would launch overhead represent without CUDA Graphs?

3. You profile a Triton attention kernel and find: Tensor Core utilization=12%, Memory throughput=78% of HBM peak. (a) Classify the bottleneck. (b) What three specific changes to the kernel would you attempt first, and in what order? (c) After fixing TC utilization to 68%, memory throughput drops to 45%. What happened, and is this a regression?

PRINCIPAL LEVEL Design, decide, and defend — architecture reviews

1. **Custom operator strategy for a new architecture:** Your team is deploying Mamba-2 (a state-space model) alongside transformer decoders. Mamba-2's core is a selective scan — not a standard GEMM or attention. (a) Walk through the kernel selection hierarchy for selective scan. (b) Estimate the arithmetic intensity of a selective scan and determine which side of the H100 roofline it likely falls on. (c) Design a Triton kernel strategy: specify tile dimensions, memory access patterns, and expected efficiency vs. roofline. Justify every number with reference to H100 hardware specs.

2. **Multi-GPU kernel overlap design:** A training system on 8×H100 with TP=8 has this per-layer sequence: QKV GEMM → AllReduce → Attention → FFN GEMMs → AllReduce. Current MFU is 52% due to AllReduce blocking compute. Design a kernel overlap strategy using CUDA streams and async collectives to reach 70%+ MFU. What synchronization constraints limit perfect overlap? Quantify expected improvement with numbers, including AllReduce latency for a 70B model parameter shard size.

3. **TCO-driven kernel investment ranking:** Three kernel projects: (A) rewrite attention in raw CUDA/TMA — +15% MFU, 3 engineer-months; (B) fix GEMM alignment across all layers — +8% MFU, 2 weeks; (C) persistent decode kernel — −18% TPOT, inference-only, 6 weeks. Cluster: 60% training, 40% inference, $4M/month hardware. Rank by ROI with full dollar-savings calculation for each project. Which do you fund first, which do you defer, and under what conditions would your ranking change?