From Zero to 800 TFLOPS: A Complete Guide to High-Performance CUDA GEMM on Hopper GPUs

Table of Contents

1. Introduction

This manual teaches you how to write a BF16 GEMM kernel achieving ~800 TFLOPS on NVIDIA H800 (989 TFLOPS theoretical peak). The journey from a naive implementation to peak performance involves understanding:

- **Tensor Cores** and their programming models (WMMA vs WGMMA)
- **Memory hierarchy optimization** (Global → L2 → Shared → Registers)
- **Asynchronous execution** (TMA, barriers, pipelining)
- **Hardware-specific features** (Clusters, multicast, swizzling)

Why This Matters

```
cuBLAS:     ~900 TFLOPS (highly optimized, closed source)
DeepGEMM:   ~800 TFLOPS (open source, state-of-the-art)
Our Goal:   ~800 TFLOPS (from scratch, fully understood)
Naive:      ~50 TFLOPS (where most people start)
```

The 16x performance gap between naive and optimized is NOT due to algorithmic differences—it's entirely about **understanding and utilizing hardware correctly**.

2. Prerequisites and Background

2.1 Hardware Model: NVIDIA Hopper (H100/H800)

Component	Specification
SM Count	132
Peak FP32	~989 TFLOPS
Shared Memory	228 KB / SM (164 KB usable with 64 KB L1)
L2 Cache	50 MB
HBM Bandwidth	3.35 TB/s
Cluster Size	Up to 16 SMs

2.2 Key Concepts

Tensor Cores: Specialized hardware for matrix multiply-accumulate. On Hopper: - WMMA: Warp-level (32 threads), tiles like 16×16×16 - WGMMA: Warpgroup-level (128 threads = 4 warps), tiles like 64×64×16 or larger

TMA (Tensor Memory Accelerator): Hardware unit for async memory copies with: - Automatic address calculation - Built-in swizzling - Barrier-based synchronization

Swizzling: Rearranging memory layout to avoid bank conflicts:

```
// 128B swizzle formula for BF16
// k' = k XOR ((m % 8) * 8)
// This maps consecutive threads to different banks
```

Warp Specialization: Dedicating different warps/warpgroups to different tasks: - Producer warps: Issue TMA loads - Consumer warps: Execute WGMMA compute

2.3 Memory Hierarchy Optimization

Level	Latency	Bandwidth	Strategy
Global (HBM)	~500 cycles	3.35 TB/s	Minimize, coalesce, prefetch
L2 Cache	~100 cycles	12+ TB/s	Tile scheduling, swizzle
Shared Memory	~30 cycles	128 KB/cycle/SM	TMA, swizzle, avoid conflicts
Registers	1 cycle	Unlimited	Accumulator reuse

3. Performance Roadmap

Level	Technique	TFLOPS (8K)	Key Insight
1	WMMA Baseline	~50	Tensor cores work, but memory-bound
2	WGMMA + B128 Swizzle	~42	Correct layout is non-trivial
3	TMA + WGMMA	~230	Hardware memory copy hides latency
4	Warp Specialization	~270	Producer-consumer pattern
5	Large Tile (128×128)	~360	Better arithmetic intensity
6	Cluster Multicast	~344	Share B matrix across CTAs
7	Persistent Kernel	~485	Reduce launch overhead
8	L2 Swizzle + Bidir	~513	Cache locality + dual multicast
9	TMA Store	~531	BF16 output = 50% less traffic
10	Async WG Scheduling	~583	Independent TMA/math tiles
11	DeepGEMM Style	~787	STSM + parallel TMA stores

4. Level 1: WMMA Baseline (50 TFLOPS)

4.1 The Naive Approach

```
// WMMA uses C++ API, accessible and debuggable
#include <mma.h>
using namespace nvcuda;

constexpr int WMMA_M = 16, WMMA_N = 16, WMMA_K = 16;

__global__ void wmma_gemm(
    const __nv_bfloat16* A, const __nv_bfloat16* B, float* C,
    int M, int N, int K)
{
    // Each warp computes a 16x16 output tile
    const int warpId = (blockIdx.x * blockDim.x + threadIdx.x) / 32;
    const int tileM = (warpId / (N / WMMA_N)) * WMMA_M;
    const int tileN = (warpId % (N / WMMA_N)) * WMMA_N;

    // Declare fragments
    wmma::fragment<wmma::matrix_a, 16, 16, 16, __nv_bfloat16, wmma::row_major> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, __nv_bfloat16, wmma::row_major> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

    wmma::fill_fragment(c_frag, 0.0f);

    // Main loop over K
    for (int k = 0; k < K; k += WMMA_K) {
        wmma::load_matrix_sync(a_frag, A + tileM * K + k, K);
        wmma::load_matrix_sync(b_frag, B + k * N + tileN, N);
        wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
    }

    wmma::store_matrix_sync(C + tileM * N + tileN, c_frag, N, wmma::mem_row_major);
}
```

4.2 Why It's Slow

Problem 1: Global Memory Thrashing - Each K iteration loads 16×16×2 = 512 bytes for A and B - For K=4096: 4096/16 = 256 iterations × 1KB = 256KB per warp - No data reuse between warps!

Problem 2: Low Arithmetic Intensity

```
Compute: 2 × 16 × 16 × 16 = 8192 FLOPs per iteration
Memory:  16 × 16 × 2 × 2 = 1024 bytes per iteration
Intensity: 8192 / 1024 = 8 FLOPs/byte
Required for compute-bound: ~300 FLOPs/byte on H800
```

4.3 First Improvement: Shared Memory Tiling

```
constexpr int BLOCK_M = 128, BLOCK_N = 128, BLOCK_K = 32;

__global__ void wmma_gemm_tiled(...) {
    __shared__ __nv_bfloat16 smemA[BLOCK_M][BLOCK_K + 8]; // +8 for padding
    __shared__ __nv_bfloat16 smemB[BLOCK_K][BLOCK_N + 8];

    // Cooperative load: all threads load part of tile
    for (int i = tid; i < BLOCK_M * BLOCK_K; i += blockDim.x) {
        int row = i / BLOCK_K, col = i % BLOCK_K;
        smemA[row][col] = A[(blockM + row) * K + (kOffset + col)];
    }
    __syncthreads();

    // WMMA on shared memory (much faster!)
    wmma::load_matrix_sync(a_frag, &smemA[warpM * 16][kk], BLOCK_K + 8);
    // ...
}
```

This gets us to **~80 TFLOPS**, but we're still leaving 10x performance on the table.

---

5. Level 2-3: WGMMA Fundamentals (40-230 TFLOPS)

5.1 Why WGMMA?

WGMMA (Warpgroup Matrix Multiply-Accumulate) operates on 4 warps (128 threads) and provides: - Larger tiles: 64×64×16, 64×128×16, 64×256×16 - Higher throughput: ~2x compute per instruction - Descriptor-based operands: Hardware handles address calculation

5.2 The WGMMA Programming Model

```
// WGMMA uses PTX inline assembly - no C++ API!
__device__ void wgmma_m64n64k16(float* acc, uint64_t desc_a, uint64_t desc_b) {
    asm volatile(
        "{\n"
        ".reg .pred p;\n"
        "setp.ne.b32 p, 1, 0;\n"
        "wgmma.mma_async.sync.aligned.m64n64k16.f32.bf16.bf16\n"
        "{%0, %1, %2, %3, %4, %5, %6, %7,\n"
        " %8, %9, %10, %11, %12, %13, %14, %15,\n"
        " %16, %17, %18, %19, %20, %21, %22, %23,\n"
        " %24, %25, %26, %27, %28, %29, %30, %31},\n"
        "%32, %33, p, 1, 1, 0, 0;\n"
        "}\n"
        : "+f"(acc[0]), "+f"(acc[1]), ... "+f"(acc[31])
        : "l"(desc_a), "l"(desc_b)
    );
}
```

5.3 The Descriptor: Key to WGMMA

WGMMA doesn't take pointers—it takes 64-bit **descriptors** that encode:

```
__device__ uint64_t make_desc_b128(const void* smem_ptr, int stride_bytes) {
    uint32_t addr = (uint32_t)__cvta_generic_to_shared(smem_ptr);
    uint64_t desc = 0;

    // Bits 0-13:  Base address (>> 4)
    desc |= (uint64_t)((addr >> 4) & 0x3FFF);

    // Bits 14-15: Layout type (3 = B128 swizzle)
    desc |= (uint64_t)(3) << 14;

    // Bits 32-45: Stride (>> 4) - distance between 8-row blocks
    desc |= (uint64_t)((stride_bytes >> 4) & 0x3FFF) << 32;

    // Bit 62: Swizzle enable
    desc |= (uint64_t)(1) << 62;

    return desc;
}
```

5.4 The 128B Swizzle: CRITICAL Understanding

The swizzle formula:

```
// For matrix A[m][k] stored in shared memory:
int k_swizzled = k ^ ((m % 8) * 8);
int index = m * stride + k_swizzled;
```

Why swizzle? - WGMMA reads 8 rows simultaneously - Without swizzle: threads 0-7 all hit same bank → 8-way conflict! - With swizzle: XOR spreads accesses across all 32 banks

Visual example (simplified):

```
Without swizzle:        With 128B swizzle:
Row 0: Banks 0,1,2,3    Row 0: Banks 0,1,2,3
Row 1: Banks 0,1,2,3    Row 1: Banks 8,9,10,11   (XOR 8)
Row 2: Banks 0,1,2,3    Row 2: Banks 16,17,18,19 (XOR 16)
...                     ...
Row 7: Banks 0,1,2,3    Row 7: Banks 56,57,58,59 (XOR 56)
```

5.5 WGMMA Synchronization: The Three-Step Dance

```
// 1. FENCE: Ensure memory operations complete before WGMMA reads
__device__ void wgmma_fence() {
    asm volatile("wgmma.fence.sync.aligned;\n" ::: "memory");
}

// 2. COMMIT: Group WGMMA operations for collective waiting
__device__ void wgmma_commit_group() {
    asm volatile("wgmma.commit_group.sync.aligned;\n" ::: "memory");
}

// 3. WAIT: Block until N groups remain pending
template<int N>
__device__ void wgmma_wait_group() {
    asm volatile("wgmma.wait_group.sync.aligned %0;\n" :: "n"(N) : "memory");
}
```

5.6 Common Mistake: Missing fence.proxy.async

CRITICAL BUG: When using manual shared memory loads (not TMA), you MUST add:

```
// After loading to shared memory
for (int i = tid; i < size; i += 128) {
    smem[swizzle(i)] = global[i];
}

// CRITICAL: This sequence is REQUIRED
__syncthreads();
asm volatile("fence.proxy.async;\n" ::: "memory");  // ← Often forgotten!
__syncwarp();

// Now WGMMA can safely read
wgmma_fence();
wgmma_m64n64k16(acc, desc_a, desc_b);
```

Why? The PTX memory model has two "proxy" types: - **Generic proxy**: Regular shared memory loads/stores - **Async proxy**: WGMMA reads from shared memory

Writes via generic proxy are NOT automatically visible to async proxy!

When is it NOT needed? TMA writes use async proxy, so TMA → WGMMA doesn't need it.

6. Level 4-5: Warp Specialization and Large Tiles (270-360 TFLOPS)

6.1 TMA: The Game Changer

TMA (Tensor Memory Accelerator) provides hardware-accelerated memory copies:

```
// TMA descriptor creation (host side, simplified)
CUtensorMap create_tma_descriptor(
    void* global_ptr, int inner_dim, int outer_dim, int stride)
{
    CUtensorMap tmap;
    cuTensorMapEncodeTiled(
        &tmap,
        CU_TENSOR_MAP_DATA_TYPE_BFLOAT16,
        2,  // 2D tensor
        global_ptr,
        {inner_dim, outer_dim},  // Global dimensions
        {stride * sizeof(bf16)}, // Stride in bytes
        {64, 64},                // Box size (tile loaded per TMA)
        {1, 1},                  // Element strides
        CU_TENSOR_MAP_INTERLEAVE_NONE,
        CU_TENSOR_MAP_SWIZZLE_128B,  // Matches WGMMA B128!
        CU_TENSOR_MAP_L2_PROMOTION_NONE,
        CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE
    );
    return tmap;
}
```

```
// TMA load (device side)
__device__ void tma_load_2d(
    const CUtensorMap* desc,
    ClusterBarrier* barrier,
    void* smem_addr,
    int coord_x, int coord_y)
{
    asm volatile(
        "cp.async.bulk.tensor.2d.shared::cluster.global.mbarrier::complete_tx::bytes"
        " [%0], [%1, {%3, %4}], [%2];"
        :: "r"(__cvta_generic_to_shared(smem_addr)),
           "l"((uint64_t)desc),
           "r"(__cvta_generic_to_shared(&barrier->barrier)),
           "r"(coord_x), "r"(coord_y)
        : "memory");
}
```

6.2 Warp Specialization Pattern

```
__global__ void gemm_warp_specialized(...) {
    int warpgroup = threadIdx.x / 128;

    if (warpgroup == 0) {
        // TMA Producer: Issue memory loads
        warpgroup_reg_dealloc<40>();  // Release registers for math WG

        for (int k = 0; k < num_k_tiles; k++) {
            // Wait for consumers to finish with previous buffer
            empty_barrier[stage]->wait(phase);

            // Issue TMA loads
            tma_load_2d(&desc_A, full_barrier[stage], sA[stage], k*BK, bm);
            tma_load_2d(&desc_B, full_barrier[stage], sB[stage], k*BK, bn);
            full_barrier[stage]->arrive_tx(TILE_BYTES);

            // Advance to next stage
            stage = (stage + 1) % NUM_STAGES;
        }
    }
    else {
        // WGMMA Consumer: Compute on loaded data
        warpgroup_reg_alloc<232>();  // Claim more registers

        for (int k = 0; k < num_k_tiles; k++) {
            // Wait for producer to fill buffer
            full_barrier[stage]->wait(phase);

            // Compute WGMMA
            wgmma_fence();
            for (int ki = 0; ki < BK; ki += 16) {
                wgmma_m64n64k16(acc, desc_a, desc_b);
            }
            wgmma_commit();
            wgmma_wait();

            // Signal buffer is free
            empty_barrier[stage]->arrive();
            stage = (stage + 1) % NUM_STAGES;
        }
    }
}
```

6.3 Multi-Stage Pipelining

The key insight: **Overlap TMA loads with WGMMA compute**

```
Time →
Stage 0: [TMA Load] [Compute] [        ] [        ]
Stage 1: [        ] [TMA Load] [Compute] [        ]
Stage 2: [        ] [        ] [TMA Load] [Compute]
Stage 3: [        ] [        ] [        ] [TMA Load]
                    ↑
         Compute and Load happen simultaneously!
```

```
// 4-stage pipeline: when consuming stage N, producer loads stage N+3
constexpr int NUM_STAGES = 4;

// Barrier initialization
if (tid == 0) {
    for (int s = 0; s < NUM_STAGES; s++) {
        full_barrier[s]->init(1);              // Producer arrives once
        empty_barrier[s]->init(num_consumers); // All consumers must arrive
    }
}
```

6.4 Large Tiles for Better Arithmetic Intensity

Tile Size	A Load	B Load	Compute	Intensity
64×64×64	8 KB	8 KB	524K FLOPs	32 FLOPs/byte
128×128×64	16 KB	16 KB	2.1M FLOPs	65 FLOPs/byte
128×256×64	16 KB	32 KB	4.2M FLOPs	87 FLOPs/byte

Larger tiles = more compute per byte loaded = closer to compute-bound

7. Level 6-7: Cluster Multicast and Persistent Kernels (350-485 TFLOPS)

7.1 Clusters: Multi-SM Cooperation

Hopper introduces **thread block clusters**—groups of CTAs that can share data.

```
// Launch configuration for 2×1×1 cluster
cudaLaunchConfig_t cfg = {};
cudaLaunchAttribute attrs[1];
attrs[0].id = cudaLaunchAttributeClusterDimension;
attrs[0].val.clusterDim.x = 2;  // 2 CTAs in cluster
attrs[0].val.clusterDim.y = 1;
attrs[0].val.clusterDim.z = 1;
cfg.attrs = attrs;
cfg.numAttrs = 1;
```

7.2 TMA Multicast: Share B Across CTAs

In GEMM, matrix B is reused across M dimension. With clusters:

```
// CTA 0 loads B and multicasts to CTA 1
if (cluster_rank() == 0) {
    tma_multicast_2d(
        &desc_B, barrier, sB,
        k_offset, n_offset,
        0x3  // Multicast mask: bits 0,1 = send to CTAs 0 and 1
    );
}
```

Benefit: Each B tile loaded once, used by 2 CTAs → 50% less B memory traffic!

7.3 Cluster Synchronization

```
// Get rank within cluster
__device__ uint32_t cluster_rank() {
    uint32_t r;
    asm volatile("mov.u32 %0, %cluster_ctarank;" : "=r"(r));
    return r;
}

// Synchronize entire cluster
__device__ void cluster_sync() {
    asm volatile(
        "barrier.cluster.arrive;\n"
        "barrier.cluster.wait;\n"
        ::: "memory");
}

// Arrive at remote CTA's barrier
__device__ void arrive_remote(ClusterBarrier* bar, uint32_t target_cta) {
    uint32_t smem_addr = __cvta_generic_to_shared(&bar->barrier);
    uint32_t remote_addr;
    asm volatile(
        "mapa.shared::cluster.u32 %0, %1, %2;"
        : "=r"(remote_addr)
        : "r"(smem_addr), "r"(target_cta));
    asm volatile(
        "mbarrier.arrive.shared::cluster.b64 _, [%0];"
        :: "r"(remote_addr) : "memory");
}
```

7.4 Persistent Kernels

Instead of launching one block per output tile, launch once and loop:

```
__global__ void gemm_persistent(...) {
    int cluster_id = blockIdx.x / CLUSTER_SIZE;
    int num_clusters = gridDim.x / CLUSTER_SIZE;

    // Each cluster loops over its assigned tiles
    for (int tile = cluster_id; tile < num_tiles; tile += num_clusters) {
        // Initialize barriers for this tile
        if (tid == 0) {
            for (int s = 0; s < NUM_STAGES; s++) {
                full[s]->init(1);
                empty[s]->init(expected_arrivals);
            }
        }
        cluster_sync();

        // Process tile (TMA loads + WGMMA compute)
        process_tile(tile);

        cluster_sync();
    }
}

// Launch with fixed grid size = SM count
kernel<<<num_SMs, threads_per_block>>>(...);
```

Benefits: - Reduced kernel launch overhead - Better load balancing (work stealing) - Barrier state preserved across tiles

7.5 The m64n256k16 WGMMA

For BN=256, use larger WGMMA instruction:

```
// m64n256k16: 128 float accumulators per thread
__device__ void wgmma_m64n256k16(float* c, uint64_t desc_a, uint64_t desc_b) {
    asm volatile(
        "wgmma.mma_async.sync.aligned.m64n256k16.f32.bf16.bf16\n"
        "{%0, %1, ..., %127}, %128, %129, p, 1, 1, 0, 0;\n"
        : "+f"(c[0]), ..., "+f"(c[127])
        : "l"(desc_a), "l"(desc_b)
    );
}
```

Register Pressure Warning: - m64n256k16 needs 128 registers per accumulator - For BM=128, you need 2× m64n256k16 = 256 registers - This CAN cause register spill if not managed carefully!

Solution: Process tiles sequentially, not in parallel:

```
// BAD: 256 registers live simultaneously → spill!
float acc0[128], acc1[128];
wgmma_m64n256k16(acc0, ...);  // rows 0-63
wgmma_m64n256k16(acc1, ...);  // rows 64-127

// GOOD: Reuse accumulator → only 128 registers
float acc[128];
// Process rows 0-63
for (k...) wgmma_m64n256k16(acc, desc_a_low, ...);
store(acc, output_low);

// Process rows 64-127
for (k...) wgmma_m64n256k16(acc, desc_a_high, ...);
store(acc, output_high);
```

---

8. Level 8-9: L2 Swizzle and TMA Store (513-531 TFLOPS)

8.1 L2 Cache Swizzle (Triton-Style)

Tile scheduling affects L2 cache hit rate:

```
// Naive linear scheduling
tile_m = tile_idx / num_n_tiles;
tile_n = tile_idx % num_n_tiles;

// Problem: Adjacent tiles in M don't share L2 for B!

// GROUP_M swizzle (Triton-style)
constexpr int GROUP_M = 8;
int num_pid_in_group = GROUP_M * num_n_tiles;
int group_id = tile_idx / num_pid_in_group;
int first_pid_m = group_id * GROUP_M;
int group_size_m = min(GROUP_M, num_m_tiles - first_pid_m);
tile_m = first_pid_m + (tile_idx % group_size_m);
tile_n = (tile_idx % num_pid_in_group) / group_size_m;
```

Visual:

```
Naive:                    GROUP_M=8 Swizzle:
(0,0)(0,1)(0,2)...        (0,0)(1,0)(2,0)...(7,0)
(1,0)(1,1)(1,2)...  →     (0,1)(1,1)(2,1)...(7,1)
(2,0)(2,1)(2,2)...        ...

Consecutive tiles share B → L2 hits!
```

8.2 Bidirectional Multicast

With Cluster 2×1, each CTA can load different halves of B:

```
if (cta == 0) {
    // CTA 0 loads B columns 0-127, multicast to CTA 1
    tma_multicast_2d(&desc_B, bar, sB, k, bn, 0x3);
} else {
    // CTA 1 loads B columns 128-255, multicast to CTA 0
    tma_multicast_2d(&desc_B, bar, sB + 128*BK, k, bn+128, 0x3);
}
```

Result: Both CTAs share full B tile, each loads half → 2x bandwidth efficiency!

8.3 TMA Store: The Final Frontier

Instead of scalar stores, use TMA for output:

```
// Create TMA descriptor for output C
CUtensorMap desc_C;
cuTensorMapEncodeTiled(
    &desc_C, CU_TENSOR_MAP_DATA_TYPE_BFLOAT16, 2,
    C_ptr, {N, M}, {N * sizeof(bf16)},
    {64, 64}, {1, 1}, ..., CU_TENSOR_MAP_SWIZZLE_128B, ...
);

// In kernel: Store via TMA
__device__ void store_with_tma(float* acc, __nv_bfloat16* sC, CUtensorMap* desc_C) {
    // 1. Convert FP32 acc to BF16 in shared memory
    store_acc_to_smem(sC, acc);

    // 2. Fence for TMA
    asm volatile("fence.proxy.async.shared::cta;\n" ::: "memory");

    // 3. Issue TMA store
    asm volatile(
        "cp.async.bulk.tensor.2d.global.shared::cta.bulk_group"
        " [%0, {%2, %3}], [%1];"
        :: "l"(desc_C), "r"(__cvta_generic_to_shared(sC)),
           "r"(n_coord), "r"(m_coord)
        : "memory");

    // 4. Wait for completion
    asm volatile("cp.async.bulk.commit_group;\n" ::: "memory");
    asm volatile("cp.async.bulk.wait_group.read 0;\n" ::: "memory");
}
```

Key Insight: BF16 output = 2 bytes vs FP32 = 4 bytes → 50% less memory traffic!

9. Level 10-11: Async Warpgroup and DeepGEMM Style (583-787 TFLOPS)

9.1 Independent TMA/Math Tile Scheduling

Previous levels: TMA and Math WGs process same tile index. Level 10+: They process **different** tile indices!

```
// TMA warpgroup: runs ahead, filling pipeline
if (wg == 0 && tid == 0) {
    int stage = 0, phase = 0;

    for (int tile = cluster_id; tile < num_tiles; tile += num_clusters) {
        for (int k = 0; k < num_k; k++) {
            // Wait for empty (math consumed previous data)
            if (stage == 0 && phase > 0) {
                empty[0]->wait(phase ^ 1);
            } else if (tile > cluster_id || k >= NUM_STAGES) {
                empty[stage]->wait(phase ^ 1);
            }

            // Issue TMA
            tma_load_2d(&dA, full[stage], sA[stage], k*BK, bm);
            tma_multicast_2d(&dB, full[stage], sB[stage], k*BK, bn, 0x3);
            full[stage]->arrive_tx(TX);

            // Advance
            stage++;
            if (stage == NUM_STAGES) { stage = 0; phase ^= 1; }
        }
    }
}

// Math warpgroup: independent tile loop
else if (wg >= 1) {
    int stage = 0, phase = 0;

    for (int tile = cluster_id; tile < num_tiles; tile += num_clusters) {
        // Initialize accumulator
        for (int i = 0; i < 128; i++) acc[i] = 0;

        for (int k = 0; k < num_k; k++) {
            full[stage]->wait(phase);

            // WGMMA compute
            wgmma_fence();
            for (int ki = 0; ki < BK; ki += 16) {
                wgmma_m64n256k16(acc, desc_a, desc_b);
            }
            wgmma_commit();
            wgmma_wait();

            // Signal empty to remote CTAs
            if (lane < CLUSTER_SIZE) empty[stage]->arrive_remote(lane);

            stage++;
            if (stage == NUM_STAGES) { stage = 0; phase ^= 1; }
        }

        // Store result
        store_tile(acc);
    }
}
```

9.2 STSM: Matrix Store Instruction

For efficient shared memory stores, use `stmatrix`:

```
__device__ void stsm_x2(__nv_bfloat162 v0, __nv_bfloat162 v1, void* smem_ptr) {
    uint32_t src0 = *reinterpret_cast<uint32_t*>(&v0);
    uint32_t src1 = *reinterpret_cast<uint32_t*>(&v1);
    asm volatile(
        "stmatrix.sync.aligned.x2.m8n8.shared.b16 [%0], {%1, %2};\n"
        :: "r"(__cvta_generic_to_shared(smem_ptr)),
           "r"(src0), "r"(src1)
    );
}
```

9.3 Output Swizzle for TMA Store

TMA store requires swizzled layout in shared memory:

```
__device__ void store_acc_to_smem_swizzle(
    __nv_bfloat16* sD, float* acc,
    int warp_idx, int lane_idx, int m_offset)
{
    constexpr int SWIZZLE_BYTES = 128;

    for (int i = 0; i < 32; i++) {
        int atom = i / 8;  // Which 64-column block
        int in_atom = i % 8;

        int row = lane_idx;
        int col = in_atom;
        col ^= row % 8;  // XOR swizzle!

        // Calculate swizzled address
        uint8_t* ptr = (uint8_t*)sD +
            warp_idx * 16 * SWIZZLE_BYTES +
            m_offset * SWIZZLE_BYTES +
            atom * BM * SWIZZLE_BYTES +
            row * 128 + col * 16;

        __nv_bfloat162 v0 = __floats2bfloat162_rn(acc[i*4], acc[i*4+1]);
        __nv_bfloat162 v1 = __floats2bfloat162_rn(acc[i*4+2], acc[i*4+3]);
        stsm_x2(v0, v1, ptr);
    }
}
```

9.4 Parallel TMA Stores

For BN=256, issue 4 parallel TMA stores (64 columns each):

```
if (math_wg == 0 && ltid < 4) {
    int col_offset = ltid * 64;
    __nv_bfloat16* atom_ptr = sD + ltid * BM * 64;

    tma_store_2d(&desc_C, atom_ptr, bn + col_offset, bm);
    tma_store_arrive();
}

// Wait after all tiles
if (ltid < 4) tma_store_wait();
```

9.5 Optimized Descriptor Computation

Pre-compute base descriptor, use offsets:

```
// Pre-compute (outside loop)
uint32_t base_desc_a = ((uint32_t)__cvta_generic_to_shared(sA[0])) >> 4;
uint32_t base_desc_b = ((uint32_t)__cvta_generic_to_shared(sB[0])) >> 4;
constexpr uint64_t desc_hi = ((uint64_t)3 << 14) |
                             ((uint64_t)(STRIDE >> 4) << 32) |
                             ((uint64_t)1 << 62);

// In loop (fast!)
uint32_t s_off = stage * (SMEM_AB / 16);
uint32_t k_off = ki * (16 / 16);  // ki/WGMMA_K
uint64_t da = ((base_desc_a + s_off + k_off) & 0x3FFF) | desc_hi;
uint64_t db = ((base_desc_b + s_off + k_off) & 0x3FFF) | desc_hi;
wgmma_m64n256k16(acc, da, db);
```

10. Critical Pitfalls and Debugging

10.1 Register Spill

Symptom: Performance drops dramatically (e.g., 50 TFLOPS instead of 500).

Check: Compile with `--ptxas-options=-v`:

```
ptxas info: Used 255 registers, 7788 bytes spill stores, 7788 bytes spill loads
```

Causes: 1. Too many accumulators live simultaneously 2. Loop unrolling creates too many variables 3. Missing `#pragma unroll` causing inefficient code

Solutions: 1. Process tiles sequentially with accumulator reuse 2. Use `#pragma nounroll` where appropriate 3. Reduce `NUM_MATH_REGS` request

10.2 Incorrect Results

Common causes:

1. **Missing fence.proxy.async** (manual loads before WGMMA)

```
// WRONG
smem[i] = global[i];
__syncthreads();
wgmma_m64n64k16(acc, desc, desc);  // WGMMA may see stale data!

// CORRECT
smem[i] = global[i];
__syncthreads();
asm volatile("fence.proxy.async;\n" ::: "memory");
__syncwarp();
wgmma_fence();
wgmma_m64n64k16(acc, desc, desc);
```

2. **Wrong descriptor stride**

```
// For BK=64, each 8-row block spans 8*64*2 = 1024 bytes
constexpr int WGMMA_STRIDE = 8 * BK * sizeof(__nv_bfloat16);  // 1024, NOT 64!
```

3. **Swizzle mismatch** between TMA and descriptor

```
// TMA: CU_TENSOR_MAP_SWIZZLE_128B
// Descriptor: layout_type = 1 (B128) → bits 62-63 = 0b01, bit 14-15 = 0b11
```

4. **Barrier race conditions**

```
// WRONG: empty arrives before math finishes
empty[s]->arrive();
wgmma_wait();  // Race!

// CORRECT
wgmma_wait();
empty[s]->arrive();
```

10.3 Hangs and Deadlocks

Cause: Barrier expected arrivals don't match actual arrivals.

Debug:

```
// Add debug prints (disable in production!)
if (tid == 0) {
    printf("Block %d: init barrier %d with %d arrivals\n",
           blockIdx.x, s, expected_arrivals);
}
```

Common mistakes: - `empty->init(4)` but only 3 consumer warps call `arrive()` - Cluster barriers initialized with wrong count - Not all threads reach `arrive()` due to early exit

10.4 Performance Debugging

Use NCU profiler:

```
ncu --set full -o report ./gemm_kernel
```

Key metrics: - **SM Throughput**: Should be >80% for compute-bound - **Memory Throughput**: Should be >80% for memory-bound - **Achieved Occupancy**: Low = register pressure - **Stall reasons**: Shows bottleneck (memory, sync, etc.)

---

11. Reference Implementation Patterns

11.1 Complete Level 7 Kernel Structure

```
__global__ void gemm_level7(
    const __grid_constant__ TmaDescriptor dA,
    const __grid_constant__ TmaDescriptor dB,
    float* C, int M, int N, int K, int num_tiles, int num_clusters)
{
    extern __shared__ char smem[];

    // Shared memory layout
    __nv_bfloat16* sA[NUM_STAGES], *sB[NUM_STAGES];
    ClusterBarrier *full[NUM_STAGES], *empty[NUM_STAGES];
    // ... initialize pointers ...

    int tid = threadIdx.x;
    int wg = tid / 128;
    int cta = cluster_rank();

    // Barrier init
    if (tid == 0) {
        for (int s = 0; s < NUM_STAGES; s++) {
            full[s]->init(1);
            empty[s]->init(CLUSTER_SIZE * num_consumer_warps);
        }
        fence_barrier_init();
    }
    cluster_sync();

    // Persistent loop
    for (int tile = cluster_id; tile < num_tiles; tile += num_clusters) {
        // Get tile coordinates (with L2 swizzle)
        int tile_m, tile_n;
        get_tile_coords(tile, num_n_tiles, num_m_clusters, tile_m, tile_n);
        int bm = tile_m * BM * CLUSTER_SIZE + cta * BM;
        int bn = tile_n * BN;

        // TMA warpgroup
        if (wg == 0) {
            warpgroup_reg_dealloc<40>();
            if (tid == 0) {
                for (int k = 0; k < num_k; k++) {
                    // Wait, load, signal
                    empty[stage]->wait(phase);
                    tma_load_2d(&dA, full[stage], sA[stage], k*BK, bm);
                    tma_multicast_2d(&dB, full[stage], sB[stage], k*BK, bn, 0x3);
                    full[stage]->arrive_tx(TX);
                    advance_stage();
                }
            }
        }
        // Math warpgroups
        else {
            warpgroup_reg_alloc<224>();
            float acc[128] = {0};

            for (int k = 0; k < num_k; k++) {
                full[stage]->wait(phase);
                wgmma_fence();
                for (int ki = 0; ki < BK; ki += 16) {
                    wgmma_m64n256k16(acc, desc_a, desc_b);
                }
                wgmma_commit();
                wgmma_wait();
                empty[stage]->arrive_remote(cta ^ 1);
                advance_stage();
            }

            store_results(acc, C, bm, bn);
        }

        cluster_sync();
    }
}
```

11.2 Output Coordinate Mapping

For m64n64k16, each of 128 threads owns 32 elements of the 64×64 output:

```
__device__ void get_coord_m64n64(int tid, int reg, int& row, int& col) {
    // tid: thread index within warpgroup (0-127)
    // reg: register index (0-31)
    int t0 = tid % 4;       // Column group (0-3)
    int t1 = (tid / 4) % 8; // Row within 8-row block
    int t2 = tid / 32;      // Which warp (0-3)

    int r0 = reg % 2;       // Even/odd row offset
    int r1 = (reg / 2) % 2; // +8 row offset
    int r2 = reg / 4;       // Which 8-column group

    int linear = t0 * 128 + t1 + t2 * 16 + r0 * 64 + r1 * 8 + r2 * 512;
    row = linear % 64;
    col = linear / 64;
}
```

For m64n256k16, adjust for 4× wider output:

```
__device__ void get_coord_m64n256(int tid, int reg, int& row, int& col) {
    int chunk = reg / 32;      // Which 64-column segment (0-3)
    int local_reg = reg % 32;

    int lm, ln;
    get_coord_m64n64(tid, local_reg, lm, ln);

    row = lm;
    col = chunk * 64 + ln;
}
```

---

12. Appendix: PTX Instruction Reference

12.1 WGMMA Instructions

```
wgmma.mma_async.sync.aligned.m64n64k16.f32.bf16.bf16
    {dst0, ..., dst31}, desc_a, desc_b, p, scale_d, imm_scale_a, imm_scale_b, imm_trans_b;

Parameters:
- m64n64k16: Tile dimensions (also m64n128k16, m64n256k16)
- f32: Accumulator type
- bf16.bf16: A and B element types
- p: Predicate (1 = accumulate, 0 = overwrite)
- scale_d: Scale accumulator (usually 1)
- imm_scale_a/b: Scale inputs (usually 1, 1)
- imm_trans_b: Transpose B (0 = no, 1 = yes)
```

12.2 TMA Instructions

```
cp.async.bulk.tensor.2d.shared::cluster.global.mbarrier::complete_tx::bytes
    [smem], [desc, {x, y}], [barrier];

cp.async.bulk.tensor.2d.shared::cluster.global.mbarrier::complete_tx::bytes.multicast::cluster
    [smem], [desc, {x, y}], [barrier], multicast_mask;

cp.async.bulk.tensor.2d.global.shared::cta.bulk_group
    [desc, {x, y}], [smem];
```

12.3 Barrier Instructions

```
mbarrier.init.shared.b64 [addr], count;
mbarrier.arrive.shared.b64 _, [addr];
mbarrier.arrive.expect_tx.shared.b64 _, [addr], tx_count;
mbarrier.try_wait.parity.shared.b64 pred, [addr], phase;
mbarrier.arrive.shared::cluster.b64 _, [remote_addr];
```

12.4 Synchronization

```
fence.proxy.async;                    // Generic → Async visibility
fence.proxy.async.shared::cta;        // For TMA store
fence.mbarrier_init.release.cluster;  // Barrier visibility
wgmma.fence.sync.aligned;             // Before WGMMA
wgmma.commit_group.sync.aligned;      // Group WGMMA ops
wgmma.wait_group.sync.aligned N;      // Wait for N groups
barrier.cluster.arrive;               // Cluster sync
barrier.cluster.wait;
```

12.5 Register Management

```
setmaxnreg.inc.sync.aligned.u32 N;    // Request N more registers
setmaxnreg.dec.sync.aligned.u32 N;    // Release N registers
```

Conclusion

The journey from 50 to 800 TFLOPS involves:

1. **Understanding hardware**: Tensor Cores, TMA, clusters
2. **Memory optimization**: Swizzling, tiling, pipelining
3. **Parallelism**: Warp specialization, async execution
4. **Attention to detail**: Barriers, descriptors, register pressure

Each optimization level builds on the previous. Start with Level 1, verify correctness, then progressively add optimizations. Use NCU profiler to identify bottlenecks.

The code in `knowledge/golden_samples/` provides working examples for each level. Study them, modify them, and measure the impact of each change.

Happy optimizing!

References

- NVIDIA Hopper Architecture Whitepaper
- CUTLASS 3.x Documentation
- DeepGEMM
- PTX ISA Documentation