Blackwell GEMM Optimization Manual

From Zero to 1500 TFLOPS: A Complete Guide to High-Performance CUDA GEMM on Blackwell GPUs

Table of Contents

1. Introduction

This manual teaches you how to write a BF16 GEMM kernel achieving ~1500 TFLOPS on NVIDIA GB200 (Blackwell architecture, SM100). The journey from a naive correct implementation to near-peak performance spans 10 optimization levels, each introducing new hardware concepts and coding patterns.

Why This Matters

```
cuBLAS:        ~1765 TFLOPS  (highly optimized, closed source)
DeepGEMM:      ~1500 TFLOPS  (open source, state-of-the-art)
Our Goal:      ~1500 TFLOPS  (from scratch, fully understood)
Naive Correct: ~91 TFLOPS    (where we start)
```

The **17x performance gap** between naive and optimized is NOT due to algorithmic differences — it's entirely about **understanding and utilizing Blackwell hardware correctly**.

What's New on Blackwell (SM100) vs Hopper (SM90)

Feature	Hopper (SM90)	Blackwell (SM100)
Matrix Multiply	WGMMA (Warpgroup MMA)	**UMMA** (tcgen05.mma)
Accumulator Storage	Registers	**TMEM** (Tensor Memory)
Instruction	`wgmma.mma_async`	**`tcgen05.mma.cta_group::2`**
Dual-SM Cooperative	N/A	**2SM via `cta_group::2`**
SM Count	132 (H100)	**192** (GB200)
BF16 Peak	~989 TFLOPS	**~2250 TFLOPS**
Shared Memory	228 KB/SM	**232 KB/SM**
HBM Bandwidth	3.35 TB/s	**~8 TB/s**
TMEM Columns	N/A	Up to **512**

2. Prerequisites and Background

2.1 Hardware Model: NVIDIA GB200 (Blackwell)

Component	Specification
GPU Architecture	SM100 (Blackwell)
SM Count	192
BF16 Tensor Core Peak	~2250 TFLOPS
Shared Memory	232 KB / SM
TMEM (Tensor Memory)	512 columns × 128 rows / SM
L2 Cache	~100 MB

HBM Bandwidth	~8 TB/s
Max Cluster Size	16 CTAs

2.2 SM100 New Hardware Features

UMMA (Unified Matrix Multiply-Accumulate): The new Tensor Core instruction on SM100, replacing WGMMA. - Issued via PTX instruction `tcgen05.mma.cta_group::{1|2}.kind::f16` - Supports `cta_group::2` — two SMs cooperate on one UMMA instruction - Accumulates results in **TMEM** instead of registers

TMEM (Tensor Memory): A new on-chip memory dedicated to Tensor Core accumulators. - Each SM has 128 rows × up to 512 columns - Accessed via `tcgen05.ld.sync.aligned.32x32b.x{4|8}` instructions - Allocated/deallocated per CTA group via `tcgen05.alloc/dealloc`

TMA (Tensor Memory Accelerator): Hardware DMA unit (same as Hopper). - Async copy Global → Shared Memory with automatic swizzling - Async copy Shared → Global (TMA Store) for epilogue - Configured via `CUtensorMap` descriptors on host side

Cluster and 2SM Mode: Two CTAs form a cluster, enabling: - Shared barrier addressing via `mapa.shared::cluster` - `cta_group::2` TMA loads that signal a shared barrier - `cta_group::2` UMMA that utilizes both SMs' Tensor Cores

2.3 The GEMM Problem

```
D(M, N) = A(M, K) × B(N, K)^T
```

- Input A: M×K, row-major, BF16
- Input B: N×K, row-major, BF16
- Output D: M×N, row-major, BF16
- Accumulation: FP32 (in TMEM)

2.4 Memory Hierarchy on Blackwell

Level	Latency	Bandwidth	Strategy
Global (HBM)	~400 cycles	8 TB/s	Minimize access, maximize reuse
L2 Cache	~100 cycles	~16 TB/s	2D swizzle tile scheduling
Shared Memory (SMEM)	~20 cycles	~20 TB/s	TMA loads with 128B swizzle
TMEM	~10 cycles	Tensor Core internal	Accumulator double buffering
			UMMA descriptor

Registers	~1 cycle	Unlimited	caching

2.5 Build Environment

All code is compiled with:

```
nvcc -gencode arch=compute_100a,code=sm_100a -O2 --shared \
     -Xcompiler -fPIC -o matmul.so matmul.cu -lcuda
```

The `compute_100a` / `sm_100a` (with “a” suffix) is required to enable architecture-specific instructions like `tcgen05.*`.

---

3. Performance Roadmap

```
Level 0:  91 TFLOPS  ────
Level 1:  97 TFLOPS  ─────
Level 2:  147 TFLOPS ────────
Level 3:  480 TFLOPS ────────────────────────
Level 4:  529 TFLOPS ───────────────────────────
Level 5:  582 TFLOPS ──────────────────────────────
Level 6:  704 TFLOPS ───────────────────────────────────
Level 7-8: 1100 TFLOPS ───────────────────────────────────────────────────────
Level 9: 1500 TFLOPS ──────────────────────────────────────────────────────────────────────────
cuBLAS:  1765 TFLOPS ─────────────────────────────────────────────────────────────────────────────────
```

Level	Key Optimization	TFLOPS	vs cuBLAS
0	Correct 2SM UMMA baseline	91	5%
1	BF16 format fix, single-stage clean	97	5.5%
2	2-stage pipeline, only-leader write	147	8%
3	True warp specialization + 8-stage deep pipeline	480	27%
4	cta_group::2 TMA + SMEM-staged epilogue	529	30%
5	Persistent kernel + 2D swizzle scheduler	582	33%
6	BLOCK_M=256, TMA Store epilogue, 3-role warps	704	40%
7-8	TMEM double buffer, barrier fine-tuning, descriptor caching	1100	62%
	DeepGEMM architecture match		

9	(256×256 tiles, separated SMEM)	1500	85%

---

4. Level 0: Baseline — Correct 2SM UMMA (91 TFLOPS)

Goal

Write the first correct BF16 GEMM kernel using SM100's 2SM UMMA instruction. No performance optimization — just get it right.

Architecture

```
┌──────────────── Cluster (2 CTAs) ─────────────────┐
│  CTA 0 (SM 0)              CTA 1 (SM 1)           │
│  SMEM:                     SMEM:                   │
│    A[128×64] (same)         A[128×64] (same)       │
│    B[64×64]  (cols 0-63)    B[64×64]  (cols 64-127)│
│       └─────────┬─────────────────┘                  │
│     tcgen05.mma.cta_group::2                       │
│     (UMMA_M=256, UMMA_N=128, UMMA_K=16)           │
│       ┌─────────┴─────────┐                          │
│   TMEM CTA 0 (128 rows) TMEM CTA 1 (128 rows)    │
│   → Store to D            → Store to D (redundant)  │
└──────────────────────────────────────────────────────┘
```

Key Components

1. TMA Descriptor Creation (Host Side):

```
CUtensorMap tma_a;
cuTensorMapEncodeTiled(&tma_a,
    CU_TENSOR_MAP_DATA_TYPE_BFLOAT16, 2,   // rank=2
    (void*)A_ptr,
    dims, strides, box, estrides,
    CU_TENSOR_MAP_INTERLEAVE_NONE,
    CU_TENSOR_MAP_SWIZZLE_128B,            // critical for UMMA
    CU_TENSOR_MAP_L2_PROMOTION_NONE,
    CU_TENSOR_MAP_FLOAT_OOB_FILL_NAN_REQUEST_ZERO_FMA);
```

2. SMEM Descriptor for UMMA:

```
__device__ uint64_t make_smem_desc(void* smem_ptr, uint32_t sbo) {
    uint64_t d = 0;
    uint32_t addr = __cvta_generic_to_shared(smem_ptr) >> 4;
    d |= (uint64_t)(addr & 0x3FFF);              // [0:14)  start_address
    d |= (uint64_t)((sbo >> 4) & 0x3FFF) << 32;  // [32:46) stride_byte_offset
    d |= (uint64_t)1 << 46;                       // [46:48) version = 1 (SM100)
    d |= (uint64_t)2 << 61;                       // [61:64) layout_type = SWIZZLE_128B
    return d;
}
```

3. Instruction Descriptor:

```
__device__ uint32_t make_instr_desc(uint32_t M, uint32_t N) {
    uint32_t d = 0;
    d |= (1u << 4);           // c_format = FP32 (accumulator)
    d |= (1u << 7);           // a_format = BF16  ← NOT 3!
    d |= (1u << 10);          // b_format = BF16
    d |= ((N / 8) << 17);     // n_dim
    d |= ((M / 16) << 24);    // m_dim
    return d;
}
```

4. Issuing UMMA:

```
// tcgen05.mma.cta_group::2.kind::f16  [tmem], desc_a, desc_b, idesc, scaleD
asm volatile(
    "{\n\t.reg .pred p;\n\t"
    "setp.ne.b32 p, %4, 0;\n\t"
    "tcgen05.mma.cta_group::2.kind::f16 [%0], %1, %2, %3, p;\n\t}\n"
    :: "r"(tmem_addr), "l"(desc_a), "l"(desc_b), "r"(idesc), "r"(accum));
```

5. UMMA Commit (signals barrier when SMEM is consumed):

```
asm volatile(
    "tcgen05.commit.cta_group::2"
    ".mbarrier::arrive::one.shared::cluster.multicast::cluster.b64"
    " [%0], %1;"
    :: "r"(bar_addr), "h"((uint16_t)0x3));
```

Critical Bug: BF16 Format Code

The instruction descriptor's `a_format` / `b_format` for BF16 is **1**, not 3:

```
F16F32Format: F16=0, BF16=1, TF32=2
```

If you use 3 (the common guess), you get `illegal instruction` at runtime.

Execution Flow

```
for each K block:
    1. barrier_wait(empty_bar)  → wait SMEM free (parity trick on first iter)
    2. TMA load A, B           → Global → Shared Memory
    3. barrier_wait(full_bar)   → wait TMA completion
    4. cluster_sync()           → ensure both CTAs have loaded
    5. UMMA compute             → 4 tcgen05.mma calls per K block
    6. umma_commit(empty_bar)   → signal SMEM can be reused

Epilogue:
    7. barrier_wait(tmem_bar)   → TMEM accumulator ready
    8. tcgen05.ld               → read FP32 from TMEM
    9. Convert to BF16          → write to Global Memory D
```

Result

- **91 TFLOPS** at 8192^3 (5% of cuBLAS)
- Correctness: PASS (allclose atol=1e-2)

---

5. Level 1-2: Multi-Stage Pipeline and Barrier Optimization (100-150 TFLOPS)

Level 1: Clean Baseline (97 TFLOPS)

Fixed the instruction descriptor bug and cleaned up barrier flow. Single-stage, simple but correct.

Level 2: 2-Stage Pipeline + Only-Leader Write (147 TFLOPS)

Key changes: 1. **2-stage SMEM double buffering** with per-stage full/empty barriers 2. **Only leader CTA writes output** — eliminates redundant global writes 3. **Hoisted `elect_one()` outside K loop** — reduces per-iteration overhead

```
// Pre-compute descriptors for 2 stages
uint32_t alo[2], blo[2], ahi, bhi;
for (int s = 0; s < 2; ++s) {
    uint64_t ad = make_smem_desc(smem_a(s), UMMA_SBO);
    alo[s] = (uint32_t)ad;
    if (s == 0) ahi = (uint32_t)(ad >> 32);
    // same for B...
}
```

Phase/parity management:

```
// Phase flips every time stage_idx wraps to 0
uint32_t stage = 0, phase = 0;
for (int kb = 0; kb < num_k_blocks; ++kb) {
    barrier_wait(empty_bar[stage], phase ^ 1);  // parity trick
    // ... TMA load, UMMA, commit ...
    stage = (stage + 1) % NUM_STAGES;
    phase ^= (stage == 0);
}
```

The **parity trick**: On the first iteration, `barrier_wait(empty_bar[0], 1)` succeeds immediately because the barrier starts at parity 0, which differs from the requested parity 1.

Why Performance Is Still Low

At this point the bottleneck is `cluster_sync()` — a full cluster barrier per K iteration: - 128 K blocks × ~1μs/sync = ~128 μs overhead per tile - This dominates the ~12 ms kernel runtime for 8192^3

---

6. Level 3-4: True Warp Specialization and Deep Pipeline

(400-650 TFLOPS)

Level 3: True Warp Specialization + 8-Stage Pipeline (480 TFLOPS)

The breakthrough: Replace the single unified loop with **independent loops** for TMA and MMA warps.

```
Warp 0 (TMA):  for kb in 0..num_k_blocks:
                   wait(empty_bar[stage])
                   issue TMA load for stage
                   arrive_expect_tx(full_bar[stage])

Warp 1 (MMA):  for kb in 0..num_k_blocks:
                   wait(full_bar[stage])
                   issue 4 UMMA instructions
                   commit(empty_bar[stage])
                   if last: commit(tmem_bar)

Other warps:    Go directly to epilogue
```

Why 8 stages: TMA can prefetch up to **8 K blocks** ahead of MMA: - SMEM per stage = 16K (A) + 8K (B) = 24K - 8 stages = 192K < 232K (SM100 capacity) - Deep pipeline hides memory latency entirely

cta_group::2 TMA with cluster barrier signaling:

```
// Leader CTA: arrive + set expected TX bytes from BOTH CTAs
if (is_leader)
    barrier_arrive_expect_tx(full_bar[stage], TMA_BYTES * CLUSTER_SIZE);
else
    barrier_arrive_expect_tx_cluster(full_bar[stage], TMA_BYTES, 0);
```

The non-leader uses `mapa.shared::cluster` to redirect its arrive to the leader CTA's barrier:

```
asm volatile("mapa.shared::cluster.u32 %0, %1, %2;"
    : "=r"(remote_addr) : "r"(local_addr), "r"(target_cta));
asm volatile("mbarrier.arrive.expect_tx.shared::cluster.b64 _, [%0], %1;"
    :: "r"(remote_addr), "r"(tx_bytes));
```

Level 4: SMEM-Staged Epilogue (529 TFLOPS)

Replaced the scalar epilogue (per-element TMEM → register → global) with:

```
Phase 1: TMEM → SMEM  (all 128 threads write coalesced)
Phase 2: SMEM → Global (128-bit vectorized stores)
```

This dramatically improves epilogue throughput through coalesced memory access.

---

7. Level 5: Persistent Kernel and 2D Swizzle Tile Scheduler (550-580 TFLOPS)

Persistent Kernel

Instead of one CTA per tile, launch **one CTA per SM** and loop over tiles:

```
// Launch config: gridDim = num_sms (not num_tiles)
TileScheduler scheduler(M, N, num_clusters);
uint32_t m_block, n_block;
while (scheduler.get_next_block(m_block, n_block)) {
    // K-loop (TMA + MMA via warp specialization)
    // Epilogue
    // Barrier re-initialization
}
```

Benefits: - **TMEM persists** across tiles (allocated once) - **No kernel launch overhead** between tiles - Enables tile scheduling for L2 cache reuse

2D Swizzle Tile Scheduler

Tiles are assigned in a pattern that maximizes L2 cache reuse:

```
Groups of 8 consecutive M blocks share the same N block:
  Group 0: M=[0..7], iterate N=0, N=1, N=2, ...
  Group 1: M=[8..15], iterate N=0, N=1, N=2, ...
```

This keeps **B-tile data hot in L2 cache** — 8 consecutive tiles share the same B columns, giving 8× B-data reuse.

---

8. Level 6-8: TMA Store Epilogue, TMEM Double Buffer, and Fine-Tuning (650-1100 TFLOPS)

Level 6: Three-Role Warp Specialization + TMA Store (704 TFLOPS)

BLOCK_M increased to 256 — eliminates ~50% tensor core waste:

With `cta_group::2` UMMA_M=256, the hardware distributes 128 rows to each SM. With BLOCK_M=128, SM1 reads out-of-bounds A data (garbage). With BLOCK_M=256, both SMs read valid rows.

Three persistent warp roles:

```
Warps 0-3: Non-epilogue
  Warp 0: TMA load (1 elected thread)
  Warp 1: MMA compute (leader CTA only)
  Warp 2: TMEM alloc/dealloc
  Warp 3: idle

Warps 4-7: Epilogue (128 threads)
  TMEM → SMEM (swizzled 128B layout)
  SMEM → Global (TMA Store)
```

TMA Store for output:

```
// TMA Store: SMEM → Global with 128B swizzle
asm volatile(
    "cp.async.bulk.tensor.2d.global.shared::cta.tile.bulk_group"
    " [%0, {%1, %2}], [%3];"
    :: "l"(desc), "r"(n_idx), "r"(m_idx), "r"(smem_addr));
```

Level 7-8: TMEM Double Buffering and Descriptor Caching (1100 TFLOPS)

TMEM double buffering: Use 2 epilogue stages so MMA can write tile N+1’s results while epilogue reads tile N.

`__shfl_sync` descriptor caching (DeepGEMM technique):

```
// Lane i stores stage i's descriptor lo in a register
uint32_t my_a_lo = a_lo_base + lane_idx * (SMEM_A_SIZE / 16);

// Inside K loop, broadcast current stage's descriptor via shuffle
uint32_t cur_a_lo = __shfl_sync(0xFFFFFFFF, my_a_lo, stage);
```

This saves ~16 registers vs. storing `uint32_t alo[8]` in an array.

---

9. Level 9: Matching DeepGEMM (1400-1600 TFLOPS)

Key Changes

#	What	Before	Level 9 (DeepGEMM)
1	Tile size	128×128	**256×256**
2	Pipeline stages	8	**4** (larger tiles fill SMEM)
3	M-waves	1	**2** (BLOCK_M / WAVE_BLOCK_M = 256/128)
4	SMEM layout	Interleaved (A+B per stage)	**Separated** (all A, then all B)
5	MMA warp	1 elected thread	**All 32 threads** for barrier_wait + shfl
6	`tmem_empty` signaling	1 thread/CTA	**All 128 threads** via cluster addressing
7	L2 promotion	NONE	**L2_256B**
8	Swizzle group	8	**16**

Results

Size	Ours (TFLOPS)	DeepGEMM	cuBLAS	Ours/DG
4096^3	1559	1660	1589	0.94x
6144^3	1590	1512	1510	1.05x
8192^3	1480	1499	1471	0.99x
10240^3	1425	1500	1427	0.95x
12288^3	1463	1466	1504	1.00x

Average: ~98% of DeepGEMM, ~100% of cuBLAS.

Critical Insight: Accumulation Flag for M-Waves

Each M-wave writes to **separate TMEM columns**, so the accumulation flag `scale_c = (kb > 0 || k > 0)` does NOT depend on the wave index. All waves clear on the first k-step:

```
// WRONG: uint32_t accum = (kb > 0 || k > 0 || w > 0) ? 1u : 0u;
// RIGHT: uint32_t accum = (kb > 0 || k > 0) ? 1u : 0u;
```

---

10. Critical Pitfalls and Debugging

Bug 1: BF16 Format Code in InstrDescriptor

The `a_format` / `b_format` for BF16 is **1**, not 3.

```
// CUTLASS F16F32Format enum:
// F16 = 0, BF16 = 1, TF32 = 2
d |= (1u << 7);   // a_format = BF16 = 1  ← NOT 3!
```

Using the wrong value causes `illegal instruction` at runtime — no compile error.

Bug 2: InstrDescriptor Must Be 32-bit

The `tcgen05.mma` instruction’s `idesc` operand is a 32-bit register (`"r"` constraint), not 64-bit (`"l"`). CUTLASS stores it as `uint64_t` in the upper 32 bits, extracting with `static_cast<uint32_t>(desc >> 32)` .

Bug 3: scaleD Predicate Required

The UMMA instruction MUST include the `scaleD` predicate argument:

```
// WRONG: tcgen05.mma.cta_group::2.kind::f16 [%0], %1, %2, %3;
// RIGHT: tcgen05.mma.cta_group::2.kind::f16 [%0], %1, %2, %3, p;
```

Bug 4: `compute_100a` Required

Standard `sm_100` doesn't support `tcgen05.*` instructions. You must use:

```
nvcc -gencode arch=compute_100a,code=sm_100a ...
```

Debugging Techniques

Illegal Instruction at Runtime: - Check `__CUDA_ARCH__` guards - Verify format codes in instruction descriptor - Use `CUDA_LAUNCH_BLOCKING=1` to get synchronous errors - Create minimal test kernels to bisect (TMA only, TMEM only, UMMA only)

Kernel Hangs (Deadlocks): - Trace barrier init count vs. expected arrivals - Verify cluster_sync participation — ALL threads in ALL CTAs must reach it - Check parity/phase management in multi-stage pipelines

Incorrect Results: - Verify accumulation flag logic for M-waves - Check SMEM descriptor SBO (stride byte offset) calculation - Ensure TMA descriptor box dimensions match BLOCK_K × BLOCK_M

---

11. Appendix: PTX Instruction Reference

UMMA (Matrix Multiply-Accumulate)

```
// Issue UMMA: cta_group::2, BF16 inputs, FP32 accumulator
tcgen05.mma.cta_group::2.kind::f16 [tmem_addr], desc_a, desc_b, idesc, scaleD;

// Commit: signal barrier when UMMA finishes reading SMEM
tcgen05.commit.cta_group::2
    .mbarrier::arrive::one.shared::cluster.multicast::cluster.b64
    [bar_addr], cta_mask;
```

TMEM Operations

```
// Allocate TMEM (full warp must execute)
tcgen05.alloc.cta_group::2.sync.aligned.shared::cta.b32 [smem_addr], ncols;

// Deallocate TMEM
tcgen05.dealloc.cta_group::2.sync.aligned.b32 tmem_addr, ncols;

// Load from TMEM (32 rows × 4 or 8 columns)
tcgen05.ld.sync.aligned.32x32b.x4.b32 {r0,r1,r2,r3}, [col_addr];
tcgen05.ld.sync.aligned.32x32b.x8.b32 {r0,...,r7}, [col_addr];

// Wait for TMEM loads to complete
tcgen05.wait::ld.sync.aligned;

// Fences
tcgen05.fence::after_thread_sync;    // after barrier_wait, before UMMA
tcgen05.fence::before_thread_sync;   // before barrier_arrive, after tmem_load
```

TMA Operations

```
// TMA Load (standard)
cp.async.bulk.tensor.2d.shared::cta.global.tile
    .mbarrier::complete_tx::bytes
    [smem_addr], [desc, {coord0, coord1}], [bar_addr];

// TMA Load (cta_group::2, routes to leader's barrier)
cp.async.bulk.tensor.2d.cta_group::2.shared::cluster.global.tile
    .mbarrier::complete_tx::bytes
    [smem_addr], [desc, {coord0, coord1}], [bar_addr];

// TMA Store
cp.async.bulk.tensor.2d.global.shared::cta.tile.bulk_group
    [desc, {coord0, coord1}], [smem_addr];

// TMA Store synchronization
fence.proxy.async.shared::cta;       // make SMEM writes visible
cp.async.bulk.commit_group;           // commit outstanding stores
cp.async.bulk.wait_group N;           // wait until ≤ N groups pending
```

Barrier Operations

```
// Initialize barrier
mbarrier.init.shared::cta.b64 [addr], count;

// Arrive with expected transaction bytes
mbarrier.arrive.expect_tx.shared::cta.b64 _, [addr], tx_bytes;

// Arrive at remote CTA's barrier (cluster addressing)
mapa.shared::cluster.u32  remote_addr, local_addr, target_cta;
mbarrier.arrive.shared::cluster.b64 _, [remote_addr];

// Wait for barrier parity
mbarrier.try_wait.parity.shared::cta.b64 pred, [addr], phase;

// Fence after barrier init
fence.mbarrier_init.release.cluster;

// Cluster synchronization
barrier.cluster.arrive.aligned;
barrier.cluster.wait.aligned;
```

Descriptor Bit Layouts

SmemDescriptor (64-bit):

```
Bits [0:14)   start_address    = smem_ptr >> 4
Bits [16:30)  leading_byte_offset >> 4  (0 for K-major)
Bits [32:46)  stride_byte_offset >> 4   (SBO)
Bits [46:48)  version          = 1 (SM100)
Bits [49:52)  base_offset      = 0
Bits [52]     lbo_mode         = 0
Bits [61:64)  layout_type      = 2 (SWIZZLE_128B)
```

InstrDescriptor (32-bit):

```
Bits [4:6)    c_format   = 1 (FP32)
Bits [7:10)   a_format   = 1 (BF16)   ← NOT 3!
Bits [10:13)  b_format   = 1 (BF16)
Bits [15]     a_major    = 0 (K-major)
Bits [16]     b_major    = 0 (K-major)
Bits [17:23)  n_dim      = N / 8
Bits [24:29)  m_dim      = M / 16
```

Summary

Building a high-performance GEMM on Blackwell from scratch teaches you every layer of GPU programming:

1. **Hardware understanding** — UMMA, TMEM, TMA are the building blocks
2. **Pipeline design** — Deep multi-stage pipelines hide memory latency
3. **Warp specialization** — Independent TMA/MMA/Epilogue loops maximize utilization
4. **Tile scheduling** — 2D swizzle for L2 cache reuse
5. **Barrier choreography** — The most subtle and error-prone aspect

The journey from 91 TFLOPS to 1500 TFLOPS is a **17× improvement** — all achieved through better utilization of the same hardware, with zero algorithmic changes to the matrix multiplication itself.

This manual is based on a hands-on development session optimizing BF16 GEMM on GB200, using DeepGEMM as the reference implementation. All code is written from scratch using inline PTX, with no CUTLASS/CuTe/DeepGEMM headers.