Evolution & Programming of Tensor Cores

From Volta to Ampere, Hopper, and Blackwell

Yuke Wang
Rice University

February 11, 2026

Outline

The Tensor Core Concept

- **Core Function:** Specialized execution units designed to perform Matrix-Multiply-and-Accumulate (MMA) operations in a single clock cycle.
- **Operation:** $D = A \times B + C$
- **The Goal:** Massive throughput for Deep Learning training and inference, originally replacing standard FP32 FMA (Fused Multiply-Add).
- **Evolution Paradigm:** Every generation introduces support for *narrower* data types to double/quadruple throughput while minimizing memory bandwidth bottlenecks.

Generation 1 & 2: Volta and Turing

Volta (V100) - The Pioneer

- Introduced 1st Gen Tensor Cores.
- Supported **FP16** inputs with **FP32** accumulation.
- 12x peak TFLOPS over Pascal (P100).

Turing (T4) - The Inference Engine

- 2nd Gen Tensor Cores.
- Added support for **INT8** and **INT4** precisions.
- Targeted high-throughput, low-latency AI inference.

Generation 3: Ampere (A100)

Massive leap in precision support and hardware utilization:

- **New Precisions:**
  - **TF32 (TensorFloat-32):** Drop-in replacement for FP32. Uses FP32 inputs but internal math is truncated to 10-bit mantissa. Zero code change required for massive speedup.
  - **Bfloat16 (BF16):** Alternative to FP16 with the same dynamic range as FP32 (8-bit exponent).
  - **FP64 Tensor Cores:** Acceleration for HPC and scientific computing.
- **Structured Sparsity:** 2:4 sparsity in network weights doubles throughput by skipping compute on zero-values.

Generation 4: Hopper (H100)

Built for large language models (LLMs):

- **FP8 Precision:** Introduces 8-bit floating-point math, doubling throughput and halving memory usage compared to FP16/BF16.
- **Transformer Engine:** Software/hardware co-design that dynamically chooses between FP8 and FP16/BF16 layer-by-layer to maximize speed without losing accuracy.
- **TMA (Tensor Memory Accelerator):** Asynchronous hardware unit to move large blocks of data between Global Memory and Shared Memory.

Generation 5: Blackwell (B100 / B200)

Pushing the limits of quantization for Trillion-Parameter models:

- **FP4 & FP6 Precision:** Next-generation Tensor Cores operate on 4-bit and 6-bit floating point.
- **2nd Gen Transformer Engine:** Supports dynamic scaling to FP4, effectively quadrupling throughput and memory bandwidth efficiency compared to Hopper FP8.
- **Micro-Tensor Scaling:** Finer-grained scaling factors to maintain neural network accuracy at extreme low precisions.

Why the Push for Lower Precision?

Deep Learning is **Memory Bandwidth Bound**.

- Moving data (Weights/Activations) from HBM to the SM takes more time and energy than the actual math.
- **Smaller Data = Faster Operations:**
  - FP32 (32 bits) $\rightarrow$ Baseline.
  - FP16/BF16 (16 bits) $\rightarrow$ 2x throughput, 1/2 memory footprint.
  - FP8 (8 bits) $\rightarrow$ 4x throughput, 1/4 memory footprint.
  - FP4 (4 bits) $\rightarrow$ 8x throughput, 1/8 memory footprint.

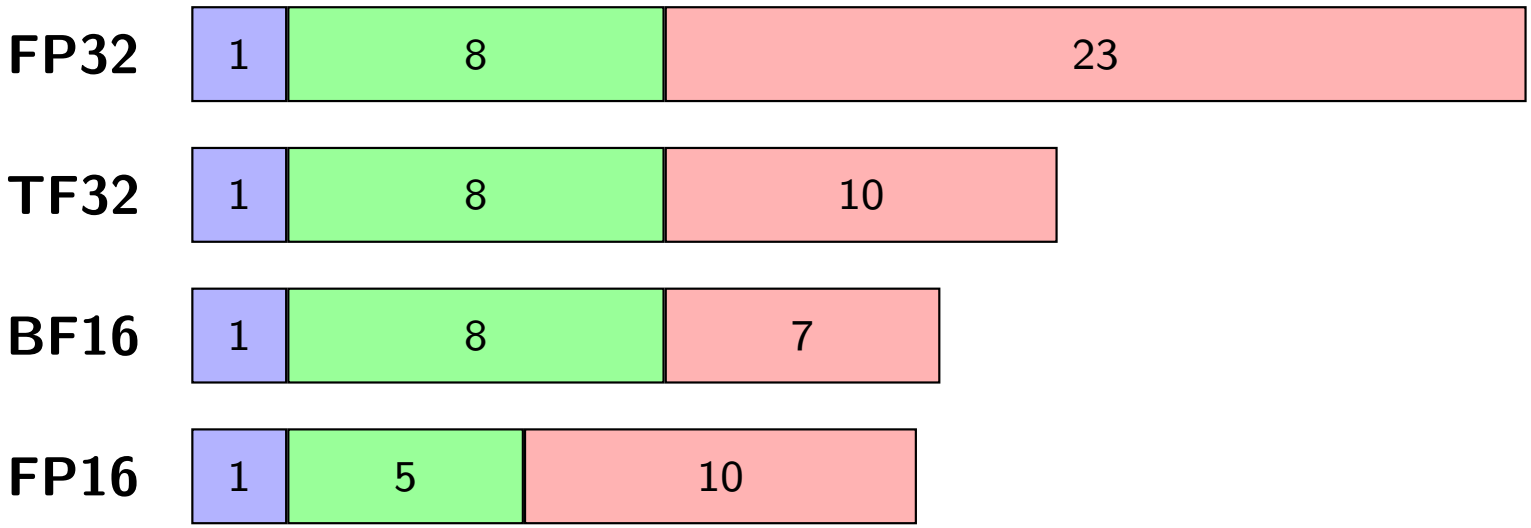
Understanding TF32 and Bfloat16

TF32 (Ampere+)

- Matches FP32 range (8-bit exponent) and FP16 precision (10-bit mantissa).
- **Benefit:** Drop-in replacement for FP32. Prevents underflow/overflow.

Bfloat16 (Ampere+)

- Matches FP32 range (8-bit exponent), but truncates to 7-bit mantissa.
- **Benefit:** Easier to train networks without complex loss scaling.



Understanding FP8 (Hopper)

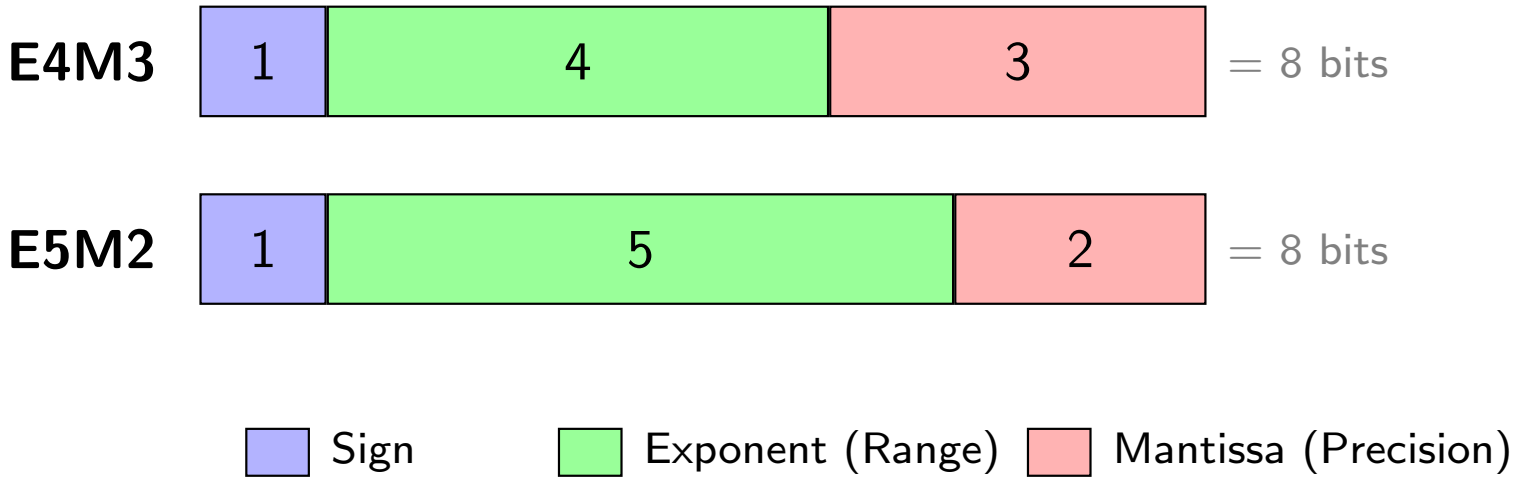
FP8 comes in two distinct formats, managed dynamically by the Transformer Engine based on the training phase.

E4M3 (Forward Pass)

- Higher precision (3 mantissa bits).
- Lower dynamic range.
- Used for Activations and Weights.

E5M2 (Backward Pass)

- Higher dynamic range (5 exponent bits).
- Lower precision.
- Used for Gradients to prevent underflow.



Libraries: The Easiest Path

Most developers will never write custom CUDA kernels for Tensor Cores.

- **cuBLASLt:** A lightweight library replacing standard cuBLAS. Essential for taking full advantage of A100/H100 features like fused epilogues (e.g., GEMM + ReLU) and exact data layout controls.
- **cuDNN:** Fully supports modern features like BF16, TF32, and Graph APIs to fuse multiple operations into single Tensor Core dispatches.
- **TensorRT / TensorRT-LLM:** Crucial for compiling and quantizing models to FP8/INT8 for Hopper/Blackwell inference.

Enabling TF32 in Code

In Ampere and newer, TF32 is often the default, but can be controlled explicitly.

cuBLAS Example:

```
// Setting Math mode for TF32 explicitly
cublasSetMathMode(handle, CUBLAS_TF32_TENSOR_OP_MATH);

// PyTorch Example:
// torch.backends.cuda.matmul.allow_tf32 = True

```

Note: You pass FP32 pointers to the API. The hardware automatically truncates the mantissa inside the SM before the matrix math!

WMMA API: Still Valid, But Evolving

The `nvcuda::wmma` API still works, but has expanded to support new types.

```
// A100/H100 supports TF32, BF16, and INT8 in WMMA
using namespace nvcuda;

// Example: Bfloat16 fragment
wmma::fragment<wmma::matrix_a, 16, 16, 16,
               __nv_bfloat16, wmma::col_major> a_frag;

// Example: TF32 fragment (Uses 'precision::tf32')
wmma::fragment<wmma::matrix_a, 16, 16, 8,
               wmma::precision::tf32, wmma::col_major> tf32_frag;

```

Note: Tile sizes change based on precision. TF32 often operates on 16x16x8 tiles.

Building a Complete WMMA GEMM Kernel (Part 1)

Let's build a functional $C = A \times B$ kernel. We assign each **warp** to compute one 16×16 tile of the output matrix C.

```cpp
#include <mma.h>
using namespace nvcuda;

__global__ void wmma_gemm_kernel(half *A, half *B, float *C, int M, int N, int K) {
    // 1. Define standard WMMA tile dimensions for FP16
    const int WMMA_M = 16;
    const int WMMA_N = 16;
    const int WMMA_K = 16;

    // 2. Calculate the global warp index (2D Grid/Block layout assumed)
    // Note: threadIdx.x must be a multiple of 32 (warpSize)
    int warpM = (blockIdx.x * blockDim.x + threadIdx.x) / warpSize;
    int warpN = (blockIdx.y * blockDim.y + threadIdx.y);

    // 3. Declare Fragments (assuming Column-Major memory layout)
    wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
    wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
    wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;

    // Initialize accumulator to 0
    wmma::fill_fragment(c_frag, 0.0f);

    // ... Continued on next slide ...

```

Building a Complete WMMA GEMM Kernel (Part 2)

Next, we loop over the K dimension, loading tiles from A and B, performing the math, and finally storing the result.

```
    // ... Continued from previous slide ...

    // 4. Loop over the K dimension
    for (int i = 0; i < K; i += WMMA_K) {
        // Calculate the memory offsets for this specific warp's tile
        int aRow = warpM * WMMA_M;
        int bCol = warpN * WMMA_N;

        // Bounds checking omitted for slide brevity

        // Load the tiles into fragments
        // Stride is M for matrix A, and K for matrix B (Col-Major)
        wmma::load_matrix_sync(a_frag, A + aRow + i * M, M);
        wmma::load_matrix_sync(b_frag, B + i + bCol * K, K);

        // Perform the Warp Matrix Multiply Accumulate
        wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
    }

    // 5. Store the final accumulated result back to Global Memory
    int cRow = warpM * WMMA_M;
    int cCol = warpN * WMMA_N;

    // Stride is M for matrix C (Col-Major)
    wmma::store_matrix_sync(C + cRow + cCol * M, c_frag, M, wmma::mem_col_major);
}
```

The Modern Standard: CUTLASS & CuTe

WMMA is often too rigid for peak performance on H100/B100.

- **CUTLASS:** NVIDIA's C++ template library for high-performance matrix multiplication.
- **CuTe (Included in CUTLASS 3.0):** A layout and tensor abstraction library used specifically to manage Hopper's complex TMA (Tensor Memory Accelerator) and asynchronous SM instructions.
- **Why use it?** It abstracts the incredibly complex PTX `mma.sync` and `wgmma` (Warp Group MMA in Hopper) instructions while achieving cuBLAS-level performance.

Rule #1: Alignment requirements grow

The Multiple of 8 rule from Volta/FP16 is no longer enough.

Alignment by Precision

- **FP16 / BF16:** Multiples of 8 (or 16 for best memory coalescing).
- **TF32:** Multiples of 4.
- **INT8 / FP8:** Multiples of 16 (often multiples of 32 for best cache line utilization).
- **FP4 (Blackwell):** Expect multiples of 32 or 64 to pack data tightly into registers.

Padding your matrices (Batch size, Feature dim) to 32 or 64 is the safest bet on modern architectures.

Optimizing for Ampere's 2:4 Sparsity

Concept: In every block of 4 elements, 2 must be zero.

- **Benefit:** 2x math throughput, 2x memory bandwidth savings.
- **Workflow:**
  1. Train dense model.
  2. Prune model using 2:4 pattern (ASP - APEX Sparse Pretraining).
  3. Fine-tune pruned model.
  4. Inference using `cuSPARSELt` or TensorRT to utilize sparse Tensor Cores.

Memory Layouts (NHWC)

NHWC is Mandatory for Peak Perf

On Ampere and Hopper, utilizing **NHWC** (Channels Last) layout is critical for Convolutions using INT8/FP8.

- **Why?** It allows the Tensor Cores to read contiguous blocks of channels (e.g., 8 channels of FP16, or 16 channels of FP8) in a single memory transaction.
- Frameworks like PyTorch support `memory_format=torch.channels_last` natively.

Hopper TMA (Tensor Memory Accelerator)

The Bottleneck: Threads spending cycles calculating memory addresses and checking bounds.

The H100 Solution:

- The TMA is an asynchronous hardware block.
- You define a multi-dimensional tensor descriptor in global memory.
- You issue **one** instruction: "TMA, fetch this 128x128 tile."
- TMA handles address generation, out-of-bounds checking, and writes directly to Shared Memory asynchronously.
- **Optimization:** Use CUTLASS 3.0, which wraps TMA instructions natively.

Utilizing the Transformer Engine

For Hopper (H100) and Blackwell (B200):

- Don't write manual FP8 quant/dequant kernels.
- Use the **NVIDIA Transformer Engine (TE)** library (integrates with PyTorch/JAX).
- TE tracks the statistics (min/max) of activations and weights during training.
- It dynamically scales values into the FP8 (or FP4) range, calls the Tensor Core matrix math, and casts back to higher precision for accumulation automatically.

Generational Scaling (Theoretical Peak)

Approximate Matrix Math Scaling (relative to Volta V100 FP16):

Architecture	**Target Precision**	**Relative Perf**
Volta (V100)	FP16	1x
Ampere (A100)	FP16/BF16	$\sim$2.5x
Ampere (A100)	Sparse FP16	$\sim$5x
Hopper (H100)	FP8	$\sim$15x
Hopper (H100)	Sparse FP8	$\sim$30x
Blackwell (B200)*	FP4	$\sim$70x+

**Note: B200 values are based on architectural announcements for dense/sparse FP4.*

Modern Optimization Checklist

1. **Pick the Right Precision:** Use TF32 for zero-code-change speedups. Use BF16/FP8 for modern LLM training.
2. **Align Dimensions:** Pad matrix M, N, K dimensions and batch sizes to multiples of 16, 32, or 64.
3. **Use Modern Abstractions:**
   - High-level: PyTorch + Transformer Engine.
   - Low-level: CUTLASS/CuTe (avoid raw WMMA for H100+ unless for simple logic).
4. **Data Layout:** Switch to NHWC for computer vision tasks.
5. **Profile:** Use `Nsight Compute` to ensure kernels with "mma", "wgmma", or "cga" in their names are running.

Questions?