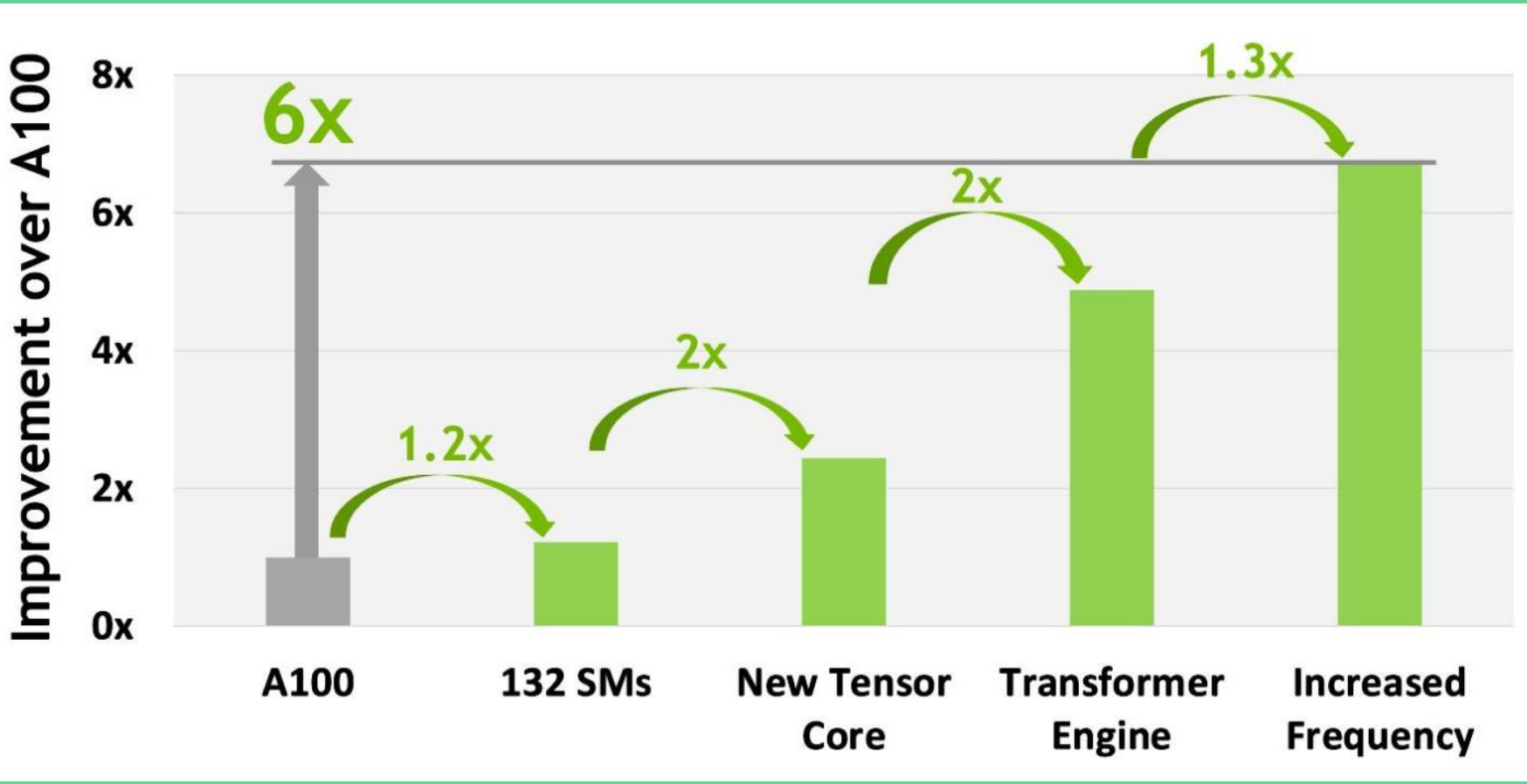
Benchmarking and Dissecting the Nvidia Hopper GPU Architecture

Amaan Mohammed

What questions should we ask when a vendor claims performance numbers from a new feature?


Improvement over A100
8x
6x
4x
2x
0x
6x
1.2x
2x
2x
1.3x
A100
132 SMs
New Tensor Core
Transformer Engine
Increased Frequency

Motivation and Context

- NVIDIA has a new architecture with new features and claimed performance
- Unexplored
  - How they operate
  - How they can perform while considering varying workload details
  - Comparing to previous architectures using same workload not explored
  - Comparing to previous architectures using same workload but new features
- “Try to examine new ISAs?”
- Explore new CUDA APIs

What They Did

1. Analysis of instructions across 3 generations Ampere, Ada, Hopper
    a. Latency, throughput, behavior of memory hierarchy, old and new instructions for all of the above (find out hidden bottlenecks that NVIDIA didn't propagandize)
    b. May not predict real application behavior
2. Workload investigation across layers
    a. Measuring peak performance of FP units for GEMM at low level - instructions
    b. New Transformer Engine - libraries
        i. Library for accelerating transformers, especially with new FP8 hardware
            1. Most ML programmers don't use raw instructions - PyTorch, CUDA, etc.
    c. Application level - LLM generation
        i. If LLM decoding is largely memory-bound, synchronization bound, SW/compiler bound, peak throughput and compute won't matter for application.

Goal: Unlock true performance of this architecture's features for AI, scientific programming, etc.

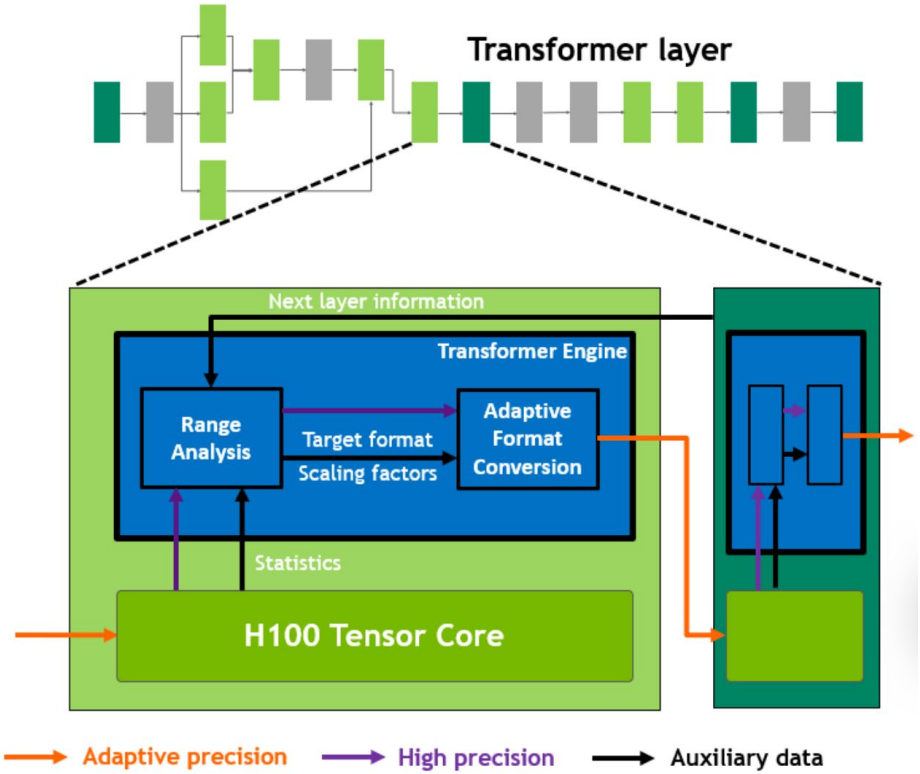
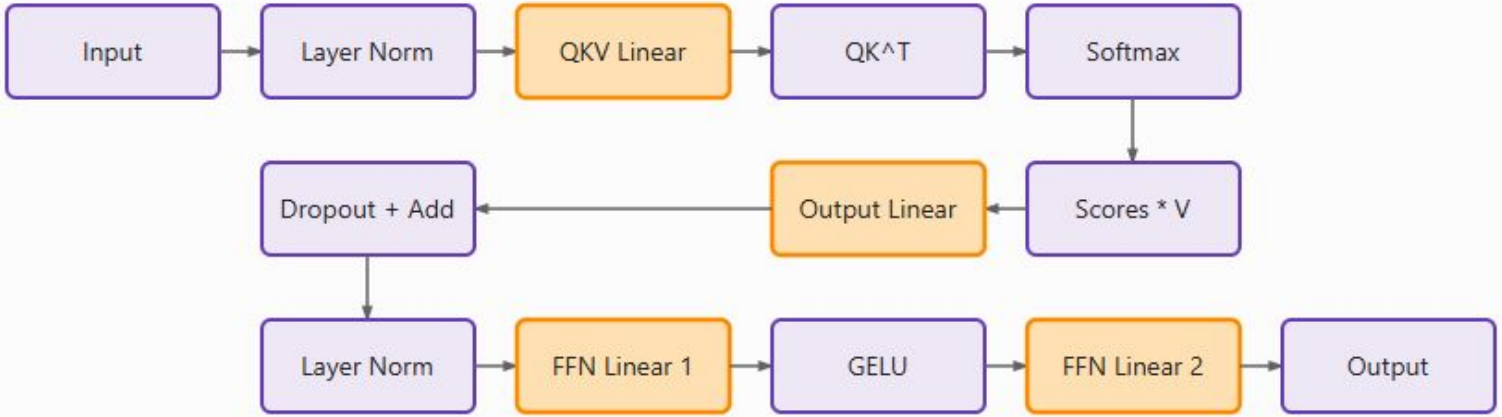


Figure 25. Transformer Engine Conceptual Operation.

Library

- Depends on FP8 HW
- At each Transformer layer, the Transformer Engine analyzes stats of TC output
- Uses information about next layer to decide what precision format is needed.
- Before storing tensor to memory, TE converts tensor format

Transformer Layer – default precision of operation in low precision recipe



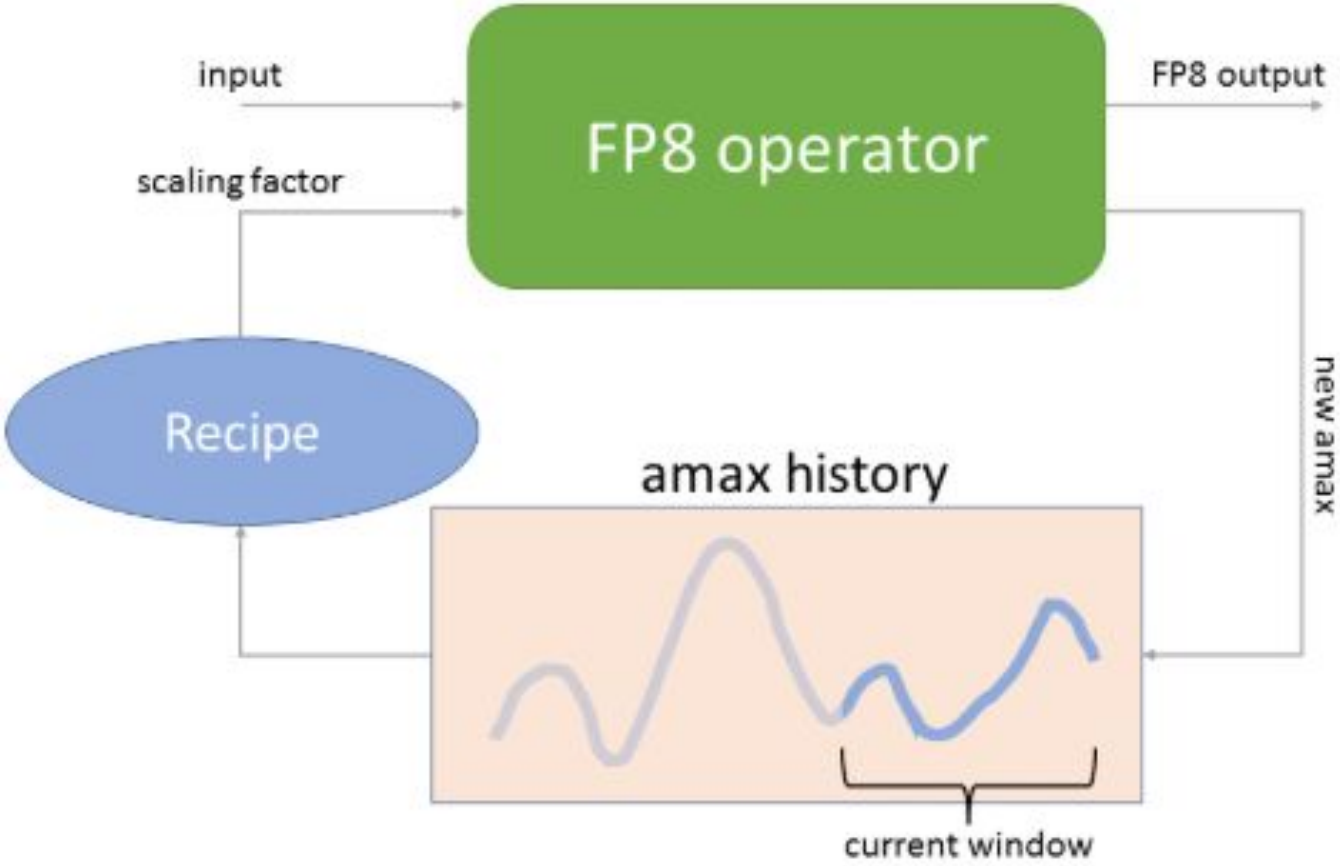
Memory State:

Parameters

Gradients

Higher Precision (FP32/BF16/FP16)

Lower Precision (FP8, MXFP8 etc.)


input
scaling factor
FP8 operator
FP8 output
Recipe
amax history
new amax
current window

Tensor Cores on Volta

Each Tensor Core operates on a 4x4 matrix and performs the following operation:

$D = A \times B + C$

where **A**, **B**, **C**, and **D** are 4x4 matrices (Figure 8). The matrix multiply inputs **A** and **B** are FP16 matrices, while the accumulation matrices **C** and **D** may be FP16 or FP32 matrices (see Figure 8).

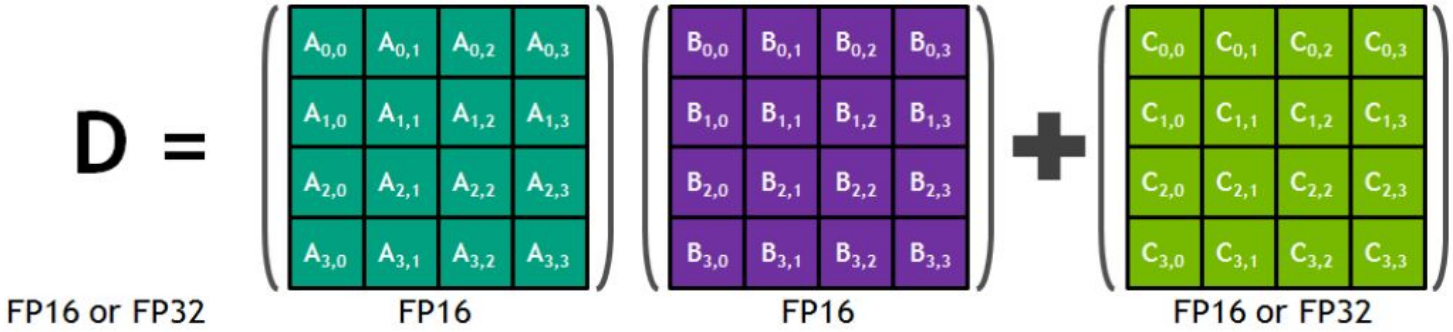


Figure 8. Tensor Core 4x4 Matrix Multiply and Accumulate

Build up to larger GEMM using this basic instruction, can do 12x of these with 1x Pascal

```
// First, create a cuBLAS handle:
cublasStatus_t cublasStat = cublasCreate(&handle);

// Set the math mode to allow cuBLAS to use Tensor Cores:
cublasStat = cublasSetMathMode(handle, CUBLAS_TENSOR_OP_MATH);

// Allocate and initialize your matrices (only the A matrix is shown):
size_t matrixSizeA = (size_t)rowsA * colsA;
T_ELEM_IN **devPtrA = 0;

cudaMalloc((void**)&devPtrA[0], matrixSizeA * sizeof(devPtrA[0][0]));
T_ELEM_IN A  = (T_ELEM_IN *)malloc(matrixSizeA * sizeof(A[0]));

memset( A, 0xFF, matrixSizeA* sizeof(A[0]));
status1 = cublasSetMatrix(rowsA, colsA, sizeof(A[0]), A, rowsA, devPtrA[i], rowsA);

// ... allocate and initialize B and C matrices (not shown) ...

// Invoke the GEMM, ensuring k, lda, ldb, and ldc are all multiples of 8,
// and m is a multiple of 4:
cublasStat = cublasGemmEx(handle, transa, transb, m, n, k, alpha,
                          A, CUDA_R_16F, lda,
                          B, CUDA_R_16F, ldb,
                          beta, C, CUDA_R_16F, ldc, CUDA_R_32F, algo);
```

Strides of rows in memory

$$C = \alpha AB + \beta C$$

Later Architectures:

- INT8, INT4, FP64, BF16, TF32,
- Support for 2:4 sparse matrices
  - HW can skip
- Hopper added FP8
  - Modernize for transformers and LLMs

New Tensor Core w/FP8 Functionality

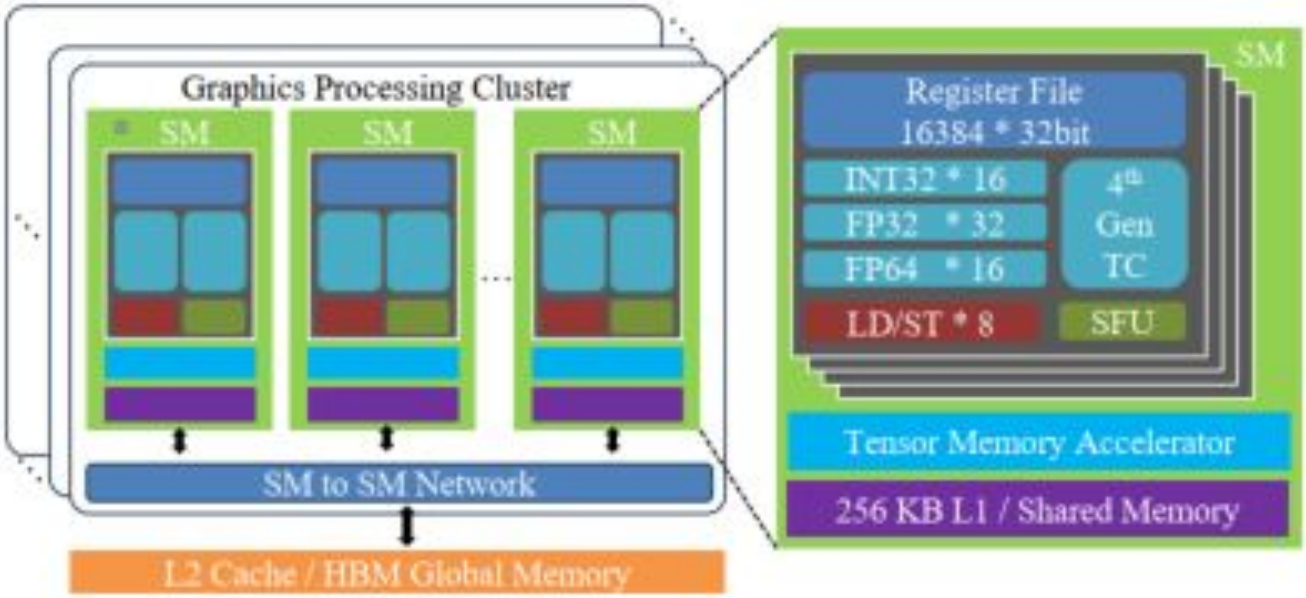
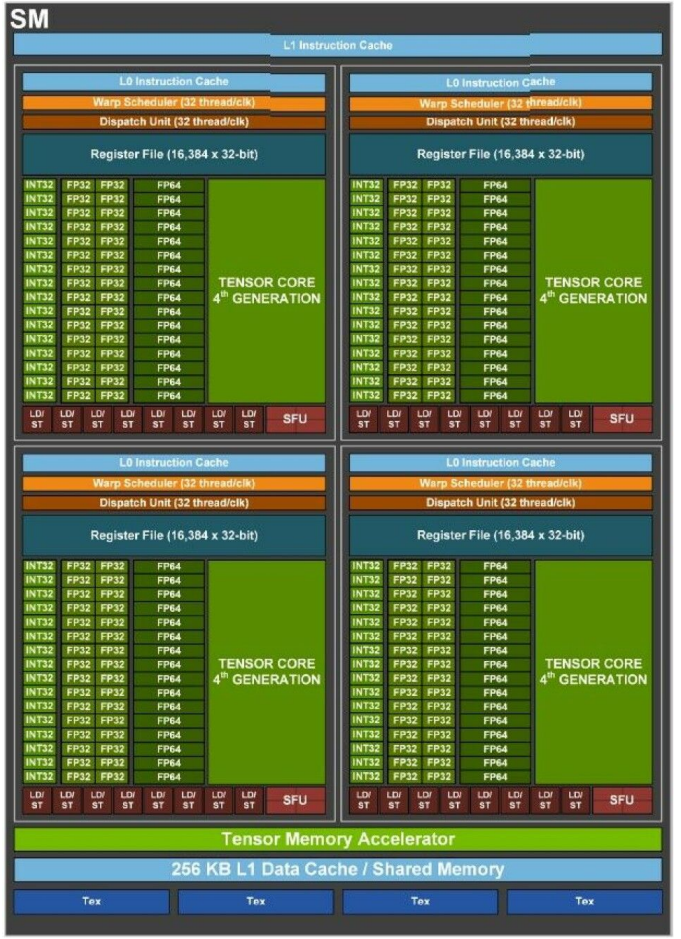


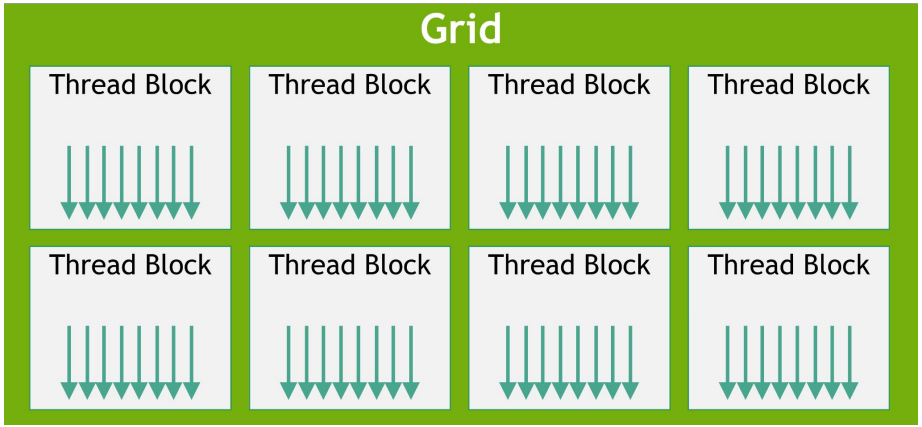
Fig. 1: Hopper architecture

Multiple SMs

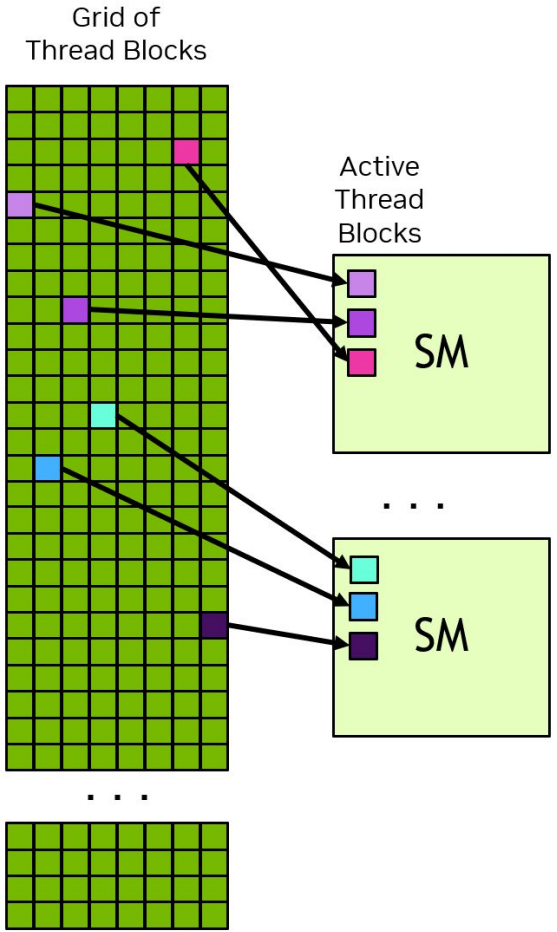
- Each SM has:
  - CUDA cores (e.g. FP32, FP64, Tensor cores)
    - More general purpose cores unlike Tensor which only does GEMM
  - Shared Memory & L1 Cache
  - Register File Load(LD)/Store(DT) Units
  - Units like SFU & Warp Scheduler
- Older generations considered increased memory bandwidth and computer from simply adding more units (extra Si)
  - New execution paths for specific workloads
    - Older studies not as sufficient



CUDA Kernel



- Each Thread block assigned to SM
  - Can be multiple
  - w/in thread block are groups of 32 threads called warps
  - Blocks can be assigned a different number of threads (not physical)
- Scheduled by warp scheduler
- SM takes care of threadblock work
  - Ex. assign a matrix multiply tile to a threadblock



Problems with Old Studies

- Legacy APIs
  - Used vendor CUBLAS and CUTLASS libraries incorporating legacy wmma APIs
  - No exposure of low-level instruction behavior
- Old wmma APIs can’t leverage new features like sparse matrix multiplication
  - Also limit size types
- wmma easier due to existing libraries but limited control
- New mma/wgmma gives more control and flexibility but architecture specific

- New libraries ignore use of sparse features, low-precision floating point and available matrix sizes
- No study of new dynamic programming instructions and new shared memory features

What They Did - Testing & Features

Instructions and Memory Analysis

- L1 Cache (hardware managed)
  - Load data from global to L1. Use thread to access it for latency measurement
  - Throughput: Issue a block with 1024 threads to access L1 cache (L1 limited to SM
- Shared memory (programmer managed and limited to SM)
  - Similar L1
  - Warmup and declare it within block
- L2 Cache
  - Use cg (cache global stores in L2)
  - Use thread to access for latency measurement
  - Since L2 cache shared by all SMs, throughput tests use large num of blocks to access.
- Global memory
  - Allocate global memory exceeding L2 size to avoid prefetching
  - Initialization to reduce cold misses and use a fixed stride for access
  - Launch 4 consecutive threads to measure latency
  - Use vectorized memory access to measure memory bandwidth

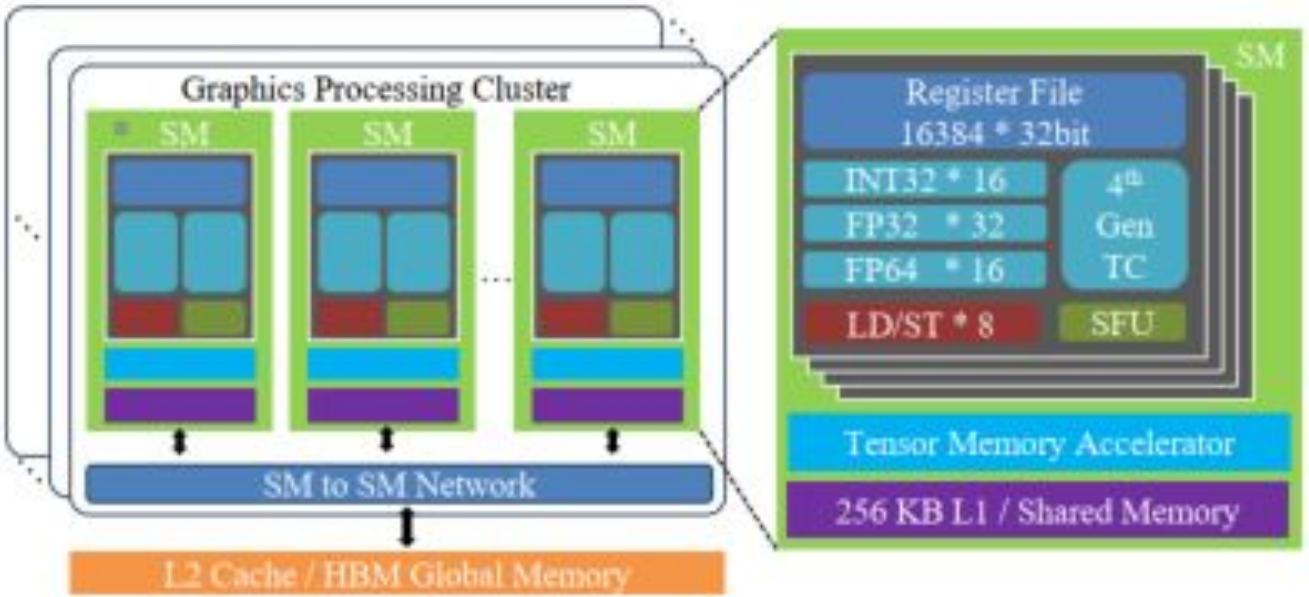


Fig. 1: Hopper architecture

Why does it matter to analyze what library/code is used to carry out GEMM op?

Tensor Cores - mma instructions

- With Ampere and Ada can use legacy C-level wmma or PTX-level mma.
  - wmma had limitation, whereas PTX mma doesn't
    - Limited tile sizes, no use of sparse features, less control of mapping
    - Need to use low level instruction instead of APIs for full perforomance

$$D(m \times n) = A(m \times k) \times B(k \times n) + C(m \times n)$$

- Hopper -> wgmma (mma still supported) for full potential
  - Mma: warp-level (32 thread), synchronous), requires matrices in register file
    - Instruction issued and then warp scheduler waits
  - Wgmma: warp-group level, asynchronous, works on larger tiles and can read operands from shared memory directly
    - Use 4 warp groups to work on a larger tile together
    - Do other work while we wait. Ex. overlap next tile data movement and computation on curr tile
      - More programmer complexity

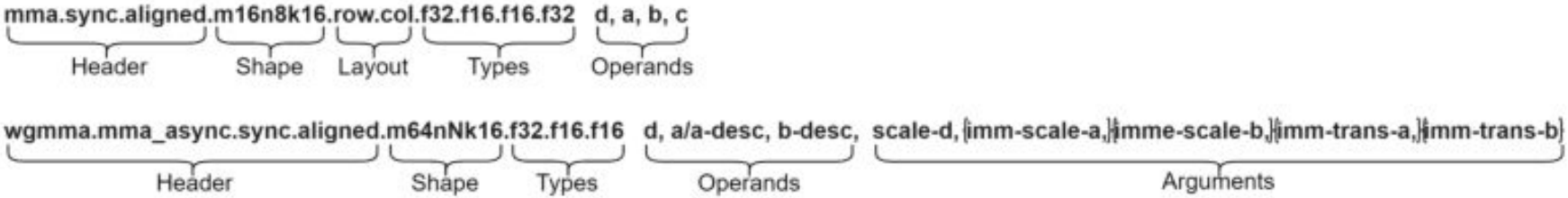


Fig. 2: The *mma* and *wgmma* instructions that perform $D = A \times B + C$ and $D = A \times B\{+D\}$, respectively.

Transformer Engine

- Variety of library modules for Transformer layers to be used in PyTorch
  - Linear Layer use FP8 conversions. Operation introduces some overhead amortized over larger matrix sizes.
  - Transformer Layer: Some modifications used for compatibility with TE FP8 usage
    - Switch activation functions and normalization functions. Changes in hidden state sizes
  - Same type of modifications with overall tests on generating text for LLAMA using ShareGPT synthesized client requests
    - Standardized inputs and text generation lengths across tests

Dynamic Programming X Instructions

- Usually used to do min/max operations for comparing previously computed solutions
- Now hardware accelerated on Hopper
- Latency - thread iteratively issues DPX functions
- Throughput - Have thread block repeatedly issue DPX functions
- Location of DPX Hardware - “we vary the number of launched blocks and observed the relationship between DPX throughput and the launched block count.”

1.4.1.5. Dynamic Programming Instructions

The NVIDIA Hopper architecture adds support for new instructions to accelerate dynamic programming algorithms, such as the Smith-Waterman algorithm for sequence alignment in bioinformatics, and algorithms in graph theory, game theory, ML, and finance problems. The new instructions permit computation of max and min values among three operands, max and min operations yielding predicates, combined add operation with max or min, operating on signed and unsigned 32-bit int and 16-bit short2 types, and half2. All DPX instructions with 16-bit short types DPX instructions enable 128 operations per cycle per SM.

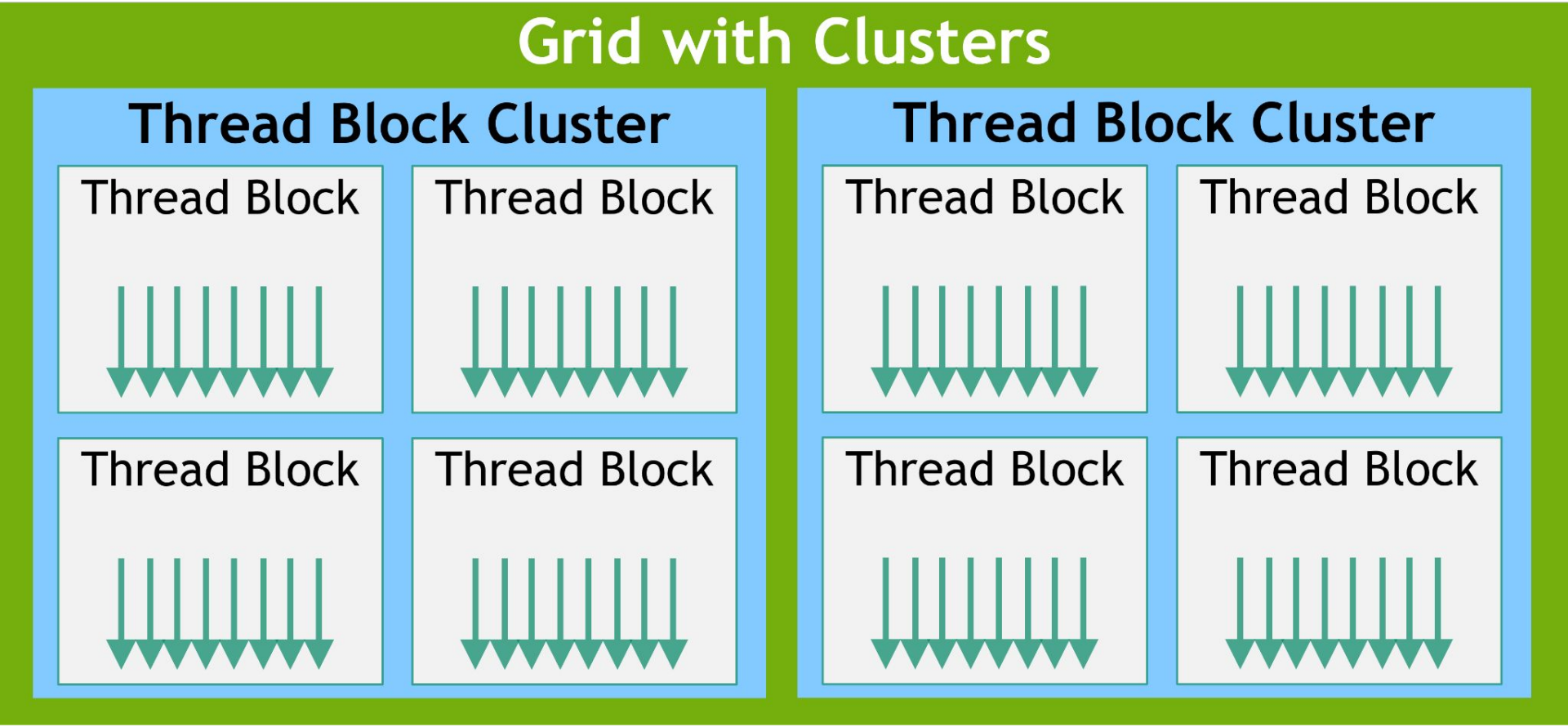
Asynchronous Data Movement (Tensor Memory Accelerator)

- Asynchronous copies between thread blocks within cluster
- Using cuda::memcpy_async
  - Avoids thread occupation during data movement
  - Computations overlap with data movement
  - Non-blocking data transfers between global memory and shared memory

- Tested features with Matrix Multiply Application across different block sizes
  - Version 1 - Conventional tiled matrix multiplication; synchronous copies from global memory
  - Version 2 - “AsyncPipe” -> double shared memory buffer size, two stage pipeline, data copy overlap over execution

Distributed Shared Memory

- Usually used for direct SM-SM communication -> done through mapa instruc
  - Loads, stores, atomics across share memory blocks owned by other SMs
  - Would otherwise have to use global memory to share


Grid with Clusters
Thread Block Cluster
Thread Block
Thread Block
Thread Block
Thread Block
Thread Block Cluster
Thread Block
Thread Block
Thread Block
Thread Block

- Latency - launch two blocks each with one thread
  - Ensure on different SMs
  - Time how long it takes to add values of one block to another
- Throughput - Ring-Based Copy
  - One block per SM, blocks into clusters, arrange clusters (size N) into rings of blocks
    - Cluster: Block 0, Block 1, ..., Block N
      - Block k communicates with Block (k + 1) % N
  - Measure how much data can be moved/updated in traffic
  - Tune cluster size and block size (number of threads) in later tests
- Histogram Application with DSM
  - Distribute bins across blocks in same cluster
  - Counting values into bins using shared memory
    - Each thread loads an element, determines associated bin in DSM address, then increments the bin

What They Did - Results

Results and Conclusion 1 - Instruction Type and GEMM

- For Hopper Tensor Cores using mma instructions can only attain an average of 62.9% of theoretical performance
  - Performance locked up behind old APIs and old code using wmma (libraries) or mma (PTX)
  - Sparse mma instructions (not wgmma) can only achieve 1.42x speed over dense mma <- don't use mma
- INT4 mma instruction are compiled into IMAD instruction running on CUDA cores not Tensor Cores
  - Programmer-visible instruction name is not enough have to see what HW it compiles to
- WGMMA -> specialized mma for Hopper
  - NVIDIA says Ada AND Hopper have 4th generation Tensor Cores
  - Can achieve 95%+ of peak with tiles initialized as zeroes using wgmma + async
    - Use large block sizes to let computational density amortize shared memory accesses
- FP8 speed-ups only available to wgmma not mma
  - To even get close to peak performance that NVIDIA says have to closely match hardware selection, use right code and ensure compilation to right instructions.

Results and Conclusion 2 - Transformer Engine and Layers

- Inspection of instruction level, library level, and application level
- When workload is large and compute-dense FP8 Tensor core speed (from wgmma and all other instructions) can provide better performance but not near peak for te.Linear modules
  - However LLM decoding is memory bound
  - FP8 conversion costs need to be amortized over large GEMM to have large throughput
  - FP8 peak throughput does not automatically speedup LLM generation

Results and Conclusion 3 - New Features and Data Movement

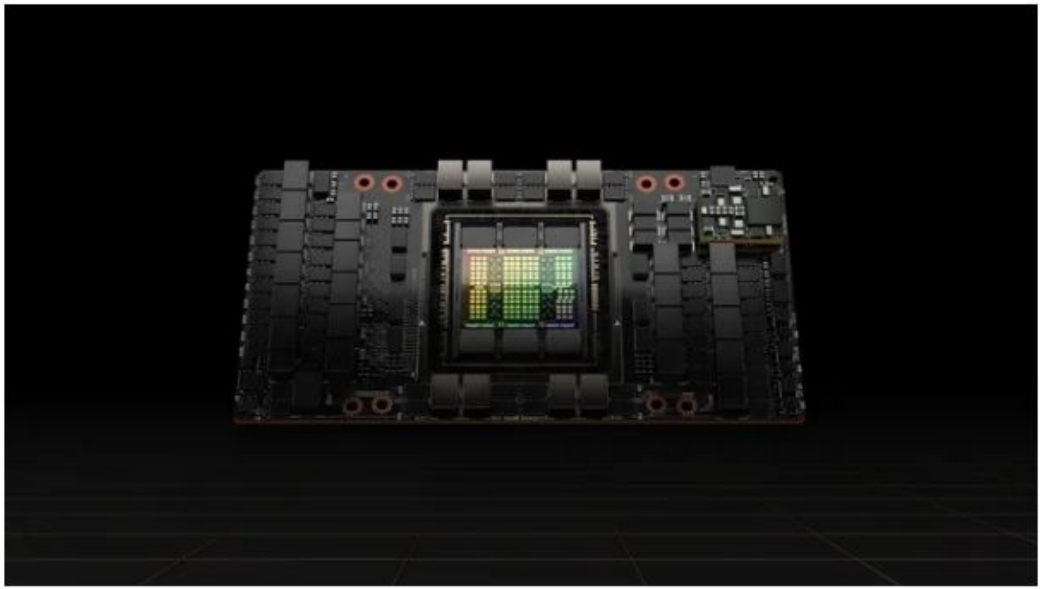
- DPX
  - For certain more complex instructions hardware acceleration of Hopper helps
    - Ex. relu (not simple ones like __viaddmax_s32 which returns max of 3 vals)
    - Max throughput with DPX when number of blocks is multiple of SMs
      - Paper concludes DPX unit is at SM level
- Asynchronous Data Movement
  - AsyncPipe shows better performance at smaller blocks sizes as Synchronous method can't make up for cost of moving memory synchronously
  - Results diminish with large block sizes
  - No direct explanation from paper as to why this is
    - There are enough other warps to schedule because the matrix is so large while waiting for next tile? Doubled shared-memory buffer space costs?
- Distributed Shared Memory
  - "SM-to-SM network latency is 180 cycles, a 32% reduction compared to L2 cache. This validates the advantages of the network, facilitating efficient data exchange from producers to consumers"
  - Need to balance number of blocks vs number of clusters allowed to share their "shared memory"
    - More blocks in cluster that can share memory? Lower Throughput
    - But larger clusters can reduce individual transfer costs for pieces of data (don't have to use L2 as often)
    - Future research

Interesting Note

Nvidia Gimps H100 Hopper GPU to Sell as H800 to China

News By Zhiye Liu published March 21, 2023

Nvidia slashes the chip-to-chip data transfer rate on the H100



“Nvidia reduced the chip-to-chip data transfer rate on the H800 to approximately half of the H100. That would leave the H800 with an interconnect restricted to 300 GBps”

- How fast GPUs communicatwqqqqqwqe each other
  - NVLink bandwidth restrictions
  - multi-GPU scaling
  - Large model training for distributed AI

Communication is bottleneck as a macro problem rather than efficient utilization as a micro problem

References

1. Benchmarking and Dissecting the Nvidia Hopper GPU Architecture
2. NVIDIA Hopper Blog Post
3. Applying Hopper to LLMs
4. Hopper Export Restrictions and Ban