Tensor Cores

1. Tensor Cores
   - introduction
   - Volta, Ampere, Hopper architectures
   - warp matrix functions
2. Simple Matrix Multiplication
   - from an NVIDIA technical blog
   - demonstration code
   - runs on volta and ampere

MCS 572 Lecture 32
Introduction to Supercomputing
Jan Verschelde, 8 November 2024

Tensor Cores

1. Tensor Cores
   - introduction
   - Volta, Ampere, Hopper architectures
   - warp matrix functions
2. Simple Matrix Multiplication
   - from an NVIDIA technical blog
   - demonstration code
   - runs on volta and ampere

introduction

Training deep neural networks is computationally expensive.

While targeted to General Matrix Multiply (GEMM),
convolution operations can be reduced to GEMM.

The tensor core peak performance in double precision increased from 19.5 on Ampere A100 to 134 TFLOPS on Hopper H100.

high throughput computing

The Volta V100 gives a 12-fold increase in *throughput*, compared to the Pascal P100.

Definition (throughput)

Throughput measures how much information a system can process in a given amount of time.

High Performance Computing (HPC) measures FLOPS.

High Throughput Computing (HTC) measures the number of jobs that can be completed over a long period.

While related, HPC is concerned with speed,
HTC is also concerned with robustness and reliability.

Tensor Cores

1. Tensor Cores
   - introduction
   - Volta, Ampere, Hopper architectures
   - warp matrix functions
2. Simple Matrix Multiplication
   - from an NVIDIA technical blog
   - demonstration code
   - runs on volta and ampere

the Volta Streaming Multiprocessor

Each Streaming Multiprocessor (SM) has 8 Tensor Cores.
The V100 has 80 SMs

$\Rightarrow$ 640 tensor cores in total.

The V100 offers 125 tensor TFLOPS of mixed precision.

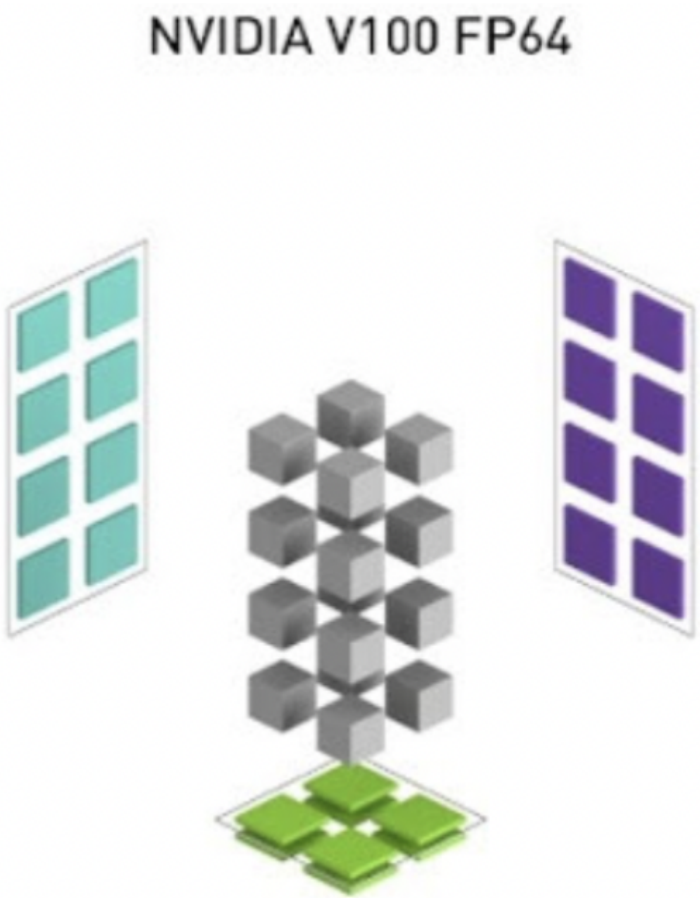
Tensor cores accelerate convolutions and matrix operations.

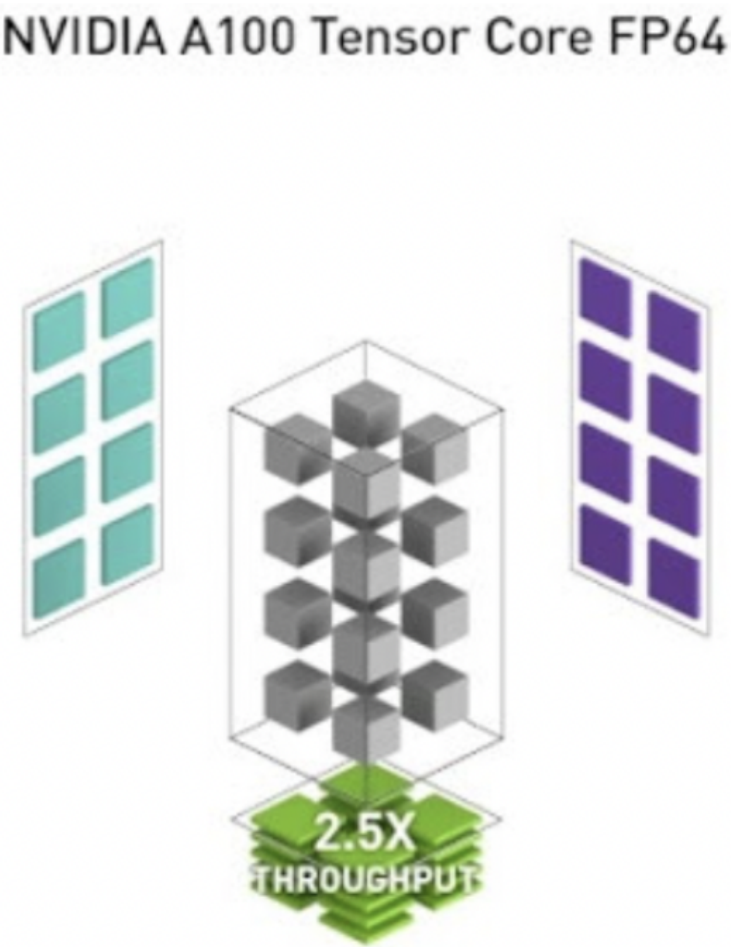
Used to accelerate HPC, data center, and machine learning applications.

From the NVIDIA Volta Architecture white paper.

V100 versus A100

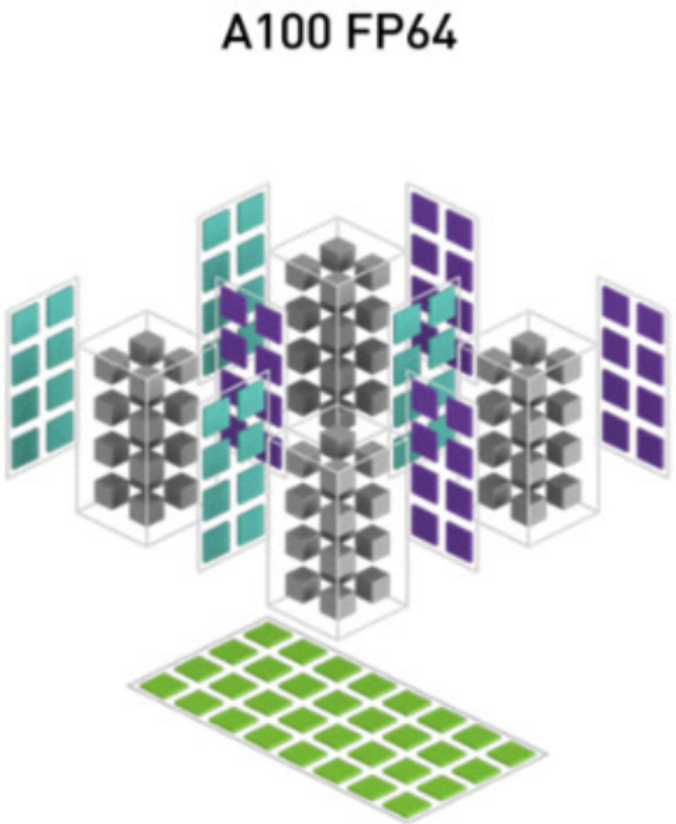
from the NVIDIA white paper on Ampere architecture

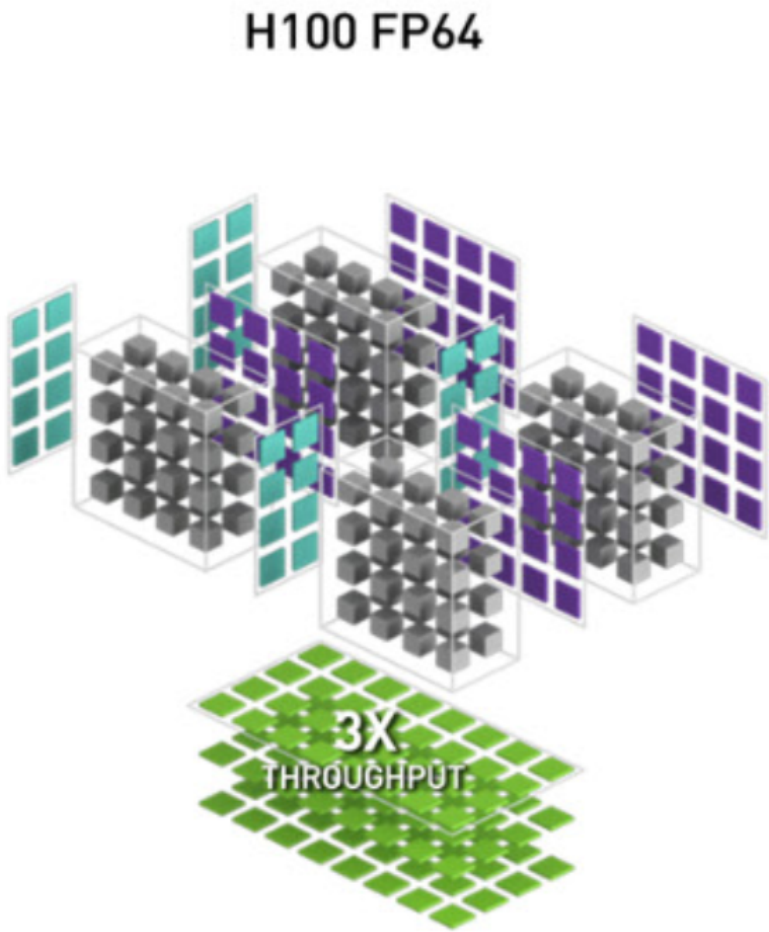




A100 versus H100

from the NVIDIA white paper on Hopper architecture





Tensor Cores

1. Tensor Cores
   - introduction
   - Volta, Ampere, Hopper architectures
   - warp matrix functions
2. Simple Matrix Multiplication
   - from an NVIDIA technical blog
   - demonstration code
   - runs on volta and ampere

mixed precision matrix operations

Vishal Mehta, SC 2019 getting started with tensor cores in HPC

TENSOR CORE

Mixed Precision Matrix Math
4x4 matrices

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 FP16 or FP32

$$D = AB + C$$

from the programming guide

About Warp Matrix Functions:

- For compute capability 7.0 or higher.
- Double precision is supported for compute capability $\geq$ 8.0.
- All threads in a warp must execute the same code.
  Code execution is likely to hang otherwise.
- A *fragment* is a templated type with template parameters describing which matrix the fragment holds (*A*, *B* or accumulator), the shape of the overall WMMA operation, the data type and, for *A* and *B* matrices, whether the data is row or column major.

Source: Release 12.1 of the CUDA C++ Programming Guide, section 10.2.4, 28 February 2023.

running an example

On a Volta computer, in the folder

```
/usr/local/cuda/samples/0_Simple/cudaTensorCoreGemm

$ ./cudaTensorCoreGemm
Initializing...
GPU Device 0: "Quadro GV100" with compute capability 7.0

M: 4096 (16 x 256)
N: 4096 (16 x 256)
K: 4096 (16 x 256)
Preparing data for GPU...
Required shared memory size: 64 Kb
Computing... using high performance kernel compute_gemm
Time: 2.768896 ms
TFLOPS: 49.64
$
```

running the same example on the A100

On `ampere`, in the folder `cudaTensorCoreGemm` of

```
/usr/local/cuda/samples/Samples/3_CUDA_Features

$ ./cudaTensorCoreGemm
Initializing...
GPU Device 0: "Ampere" with compute capability 8.0

M: 4096 (16 x 256)
N: 4096 (16 x 256)
K: 4096 (16 x 256)
Preparing data for GPU...
Required shared memory size: 64 Kb
Computing... using high performance kernel compute_gemm
Time: 1.756160 ms
TFLOPS: 78.26
$
```

a short explanation

The Warp Matrix Multipy and Accumulate (WMMA) computes

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

where

- matrix A is `M`-by-`K` row major,
- matrix B is `K`-by-`N` column major, and
- matrices C and D are `M`-by-`N`.

Each Cooperative Thread Array (CTA)

- consists of 8 warps and
- computes one 128-by-128 tile,
- using shared memory for the matrix C.

Tensor Cores

1. Tensor Cores
   - introduction
   - Volta, Ampere, Hopper architectures
   - warp matrix functions

2. Simple Matrix Multiplication
   - from an NVIDIA technical blog
   - demonstration code
   - runs on volta and ampere

demonstration code

Based on a 2017 NVIDIA Technical Blog,
by Jeremy Appleyard and Scott Yokim,
available from `https://developer.nvidia.com/blog`
on Programming Tensor Cores in CUDA 9.

The demonstration code is available on github via
`https://github.com/NVIDIA-developer-blog/code-samples`
posted with a `Makefile`.

Show the use of the WMMA (Warp Matrix Multiply Accumulate) API
to perform a matrix multiplication.

- For performance, use the `cudaTensorCoreGemm`
  in the CUDA Toolkit.
- For highest performance, use cuBLAS.

steps in the code

There are four steps in the demonstration code:

1. Use headers and namespaces.
2. Declarations and initialization:
   A simple warp is responsible for a single 16-by-16 section of the output matrix. Tiling happens with a 2D grid:

```
int warpM = (blockIdx.x*blockDim.x+threadIdx.x)/ warpSize;
int warpN = (blockIdx.y*blockDim.y+threadIdx.y);
```

3. The inner loop performs the matrix multiplication.
4. Finishing up: store the accumulated data to memory.

Tensor Cores

1. Tensor Cores
   - introduction
   - Volta, Ampere, Hopper architectures
   - warp matrix functions

2. Simple Matrix Multiplication
   - from an NVIDIA technical blog
   - demonstration code
   - runs on volta and ampere

fragments

A fragment is a templated type with parameters as follows:

1. which matrix the fragment holds, A, B, or accumulator;
2. the shape of the overall WMMA operation;
3. the data type;
4. for A and B matrices, whether the data is row or column major.

The parameters are specified at the declaration of the fragment.

Accumulator fragments are filled with zeros at initialization.

declaration and initialization

Declaration of the fragments:

```
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> acc_frag;
wmma::fragment<wmma::accumulator,
               WMMA_M, WMMA_N, WMMA_K, float> c_frag;
```

Initialization of the accumulator fragment:

```
   wmma::fill_fragment(acc_frag, 0.0f);
```

the inner loop

One tile of the output matrix is computed by one warp.

- The loop runs over the rows of A and columns of B, to produce an m-by-n output tile.
- Data is loaded from global memory into a fragment.
- If a tile is discontinous in memory, the stride must be provided to the load function.
- The Matrix Multiply Accumulate (MMA) accumulates in place, so both first and last arguments are the accumulator fragment previously initialized to zero.

headers and declarations

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

// Must be multiples of 16 for wmma code to work
#define MATRIX_M 16384
#define MATRIX_N 16384
#define MATRIX_K 16384

// The only dimensions currently supported by WMMA
const int WMMA_M = 16;
const int WMMA_N = 16;
const int WMMA_K = 16;
```

start of the kernel

```
// Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
// assuming:
//  1) Matrices are packed in memory.
//  2) M, N and K are multiples of 16.
//  3) Neither A nor B are transposed.

__global__ void wmma_example
  ( half *a, half *b, float *c,
    int M, int N, int K, float alpha, float beta)
{
    // Leading dimensions.
    int lda = M;
    int ldb = K;
    int ldc = M;
```

loop over k

```
for (int i = 0; i < K; i += WMMA_K)
{
   int aRow = warpM * WMMA_M;
   int aCol = i;

   int bRow = i;
   int bCol = warpN * WMMA_N;

   // Bounds checking
   if (aRow < M && aCol < K && bRow < K && bCol < N)
   {
      // Load the inputs
      wmma::load_matrix_sync(a_frag, a+aRow+aCol*lda, lda);
      wmma::load_matrix_sync(b_frag, b+bRow+bCol*ldb, ldb);

      // Perform the matrix multiplication
      wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
   }
}
```

scale by beta and alpha

The code below loads the current value of `c`,
scales it by `beta` and adds this to our result scaled by `alpha`.

```
int cRow = warpM * WMMA_M;
int cCol = warpN * WMMA_N;

if (cRow < M && cCol < N)
{
   wmma::load_matrix_sync(c_frag, c + cRow + cCol * ldc,
                          ldc, wmma::mem_col_major);

#pragma unroll
   for(int i=0; i < c_frag.num_elements; i++)
   {
      c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];
   }
   // Store the output
   wmma::store_matrix_sync(c + cRow + cCol * ldc, c_frag,
                           ldc, wmma::mem_col_major);
}
```

Tensor Cores

1. Tensor Cores
   - introduction
   - Volta, Ampere, Hopper architectures
   - warp matrix functions

2. Simple Matrix Multiplication
   - from an NVIDIA technical blog
   - demonstration code
   - runs on volta and ampere

runs on volta and ampere

```
$ ./TCGemm

M = 16384, N = 16384, K = 16384.
alpha = 2.000000, beta = 2.000000

Running with wmma...
Running with cuBLAS...

Checking results...
Results verified: cublas and WMMA agree.
```

On the Volta V100:

```
wmma took 631.051270ms
cublas took 99.577888ms
```

On the Ampere A100:

```
wmma took 501.762054ms
cublas took 38.711296ms
```

some suggested reading

- NVIDIA. CUDA C++ Programming Guide.
- Da Yan, Wei Wang, Xiaowen Chu: **Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply.**
  In the Proceedings of the 2020 *IEEE International Parallel and Distributed Processing Symposium* (IPDPS), pages 634–643.
- Thomas Faingnaert, Tim Besard, and Bjorn De Sutter: **Flexible Performant GEMM Kernels on GPUs**. *IEEE Transactions on Parallel and Distributed Systems* 33:(9): 2230–2248, 2022.
- Massimiliano Fasi, Nicholas J. Higham, Florent Lopez, Theo Mary, and Mantas Mikaitis:
  Matrix Multiplication in Multiword Arithmetic: Error Analysis and Application to GPU Tensor Cores.
  SIAM Journal on Scientific Computing 45(1): C1–C19, 2023.

exercises

1. Let $\mathbf{x} = (x_0, x_1, x_2, x_3, x_4)$ and $\mathbf{y} = (y_0, y_1, y_2, y_3, y_4)$ be two vectors and consider its convolution $x_0y_4 + x_1y_3 + x_2y_2 + x_3y_1 + x_4y_0$. Demonstrate how to rewrite convolutions as matrix products.
2. Install the Julia package `GemmKernels.jl`.
   Read the paper by Faingnaert et al. and run an example of matrix multiplication with the package.