香港中文大學
The Chinese University of Hong Kong

CMSC5743 Lab 03
CUDA Tutorial Materials

Yang BAI
Department of Computer Science & Engineering
Chinese University of Hong Kong
ybai@cse.cuhk.edu.hk

October 29, 2021

Outline

CUDA Programming Language

- Heterogeneous Computing
- Host and Device
- CUDA C/C++

Vector Addition

Coding Style and Organization

- Host Code Initialization
- Device Code Initialization
- Kernel Code
- Check Your Results
- Free Source on Host
- Free Source on Device

Host Code Initialization

```
int* a;
int* b;
int* c;
int* dev_a;
int* dev_b;
int* dev_c;

a = (int* )malloc(sizeof(int) * N);
b = (int* )malloc(sizeof(int) * N);
c = (int* )malloc(sizeof(int) * N);

for ( int i = 0;i < N;i++ ) {
    a[i] = i;
    b[i] = N - i - 1;
}
```

Device Code Initialization

```
cudaMalloc((void**)&dev_a, N * sizeof(int));
cudaMalloc((void**)&dev_b, N * sizeof(int));
cudaMalloc((void**)&dev_c, N * sizeof(int));

cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice);

add<<<numBlocks, numThreads>>>(dev_a, dev_b, dev_c);
cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost);
```

Kernel Code

```
#include <stdio.h>
#include <iostream>

#define N 128
#define numThreads 128
#define numBlocks 1


__global__ void add(int* a, int* b, int* c) {

    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < N) {
        c[tid] = a[tid] + b[tid];
    }
}
```

Check Your Results

```
bool flag = true;
int tot = 0;
printf("Let us check our results...\n");
for( int i = 0;i < N;i++ ) {
    if (a[i] + b[i] != c[i]) {
        flag = false;
        printf("%d + %d != %d\n", a[i], b[i], c[i]);
    }
    tot += 1;
}

if (flag) {
    printf("success!");
}
```

Free Source on Host and Device

```
free(a);
free(b);
free(c);

cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
```

Tensor Core WMMA

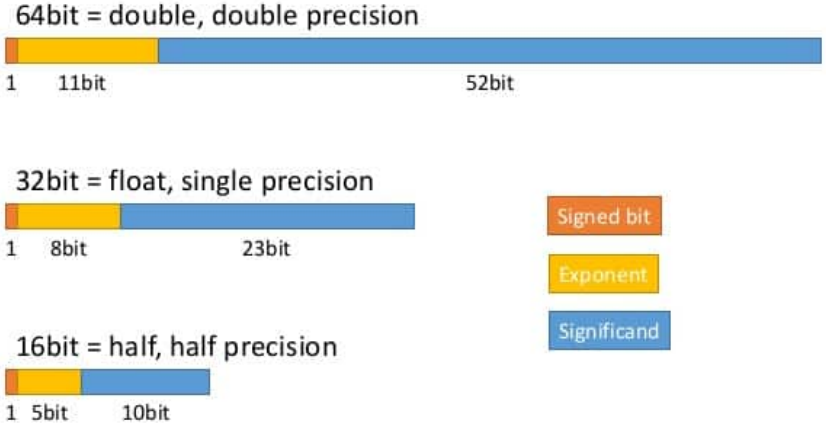
Tensor Core Overview

- Access to Tensor Core by cuBLAS/cuDNN
- Access to Tensor Core by CUTLASS
- Access to Tensor Core by TVM

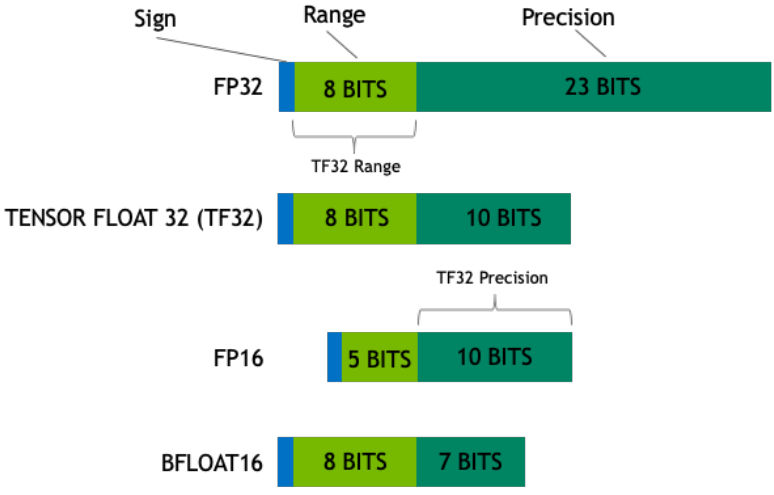
Tensor Core History

- Volta Tensor Core (1st)
  - FP16 supported
  - 8 x 8 x 4 (M x N x K)
- Turing Tensor Core (2nd)
  - FP16 supported
  - 8 x 8 x 4, 16 x 8 x 8 (recommended)
  - INT8, INT4, INT1 supported
  - 8 x 8 x 16, 8 x 8 x 32, 8 x 8 x 128
- Ampere Tensor Core (3rd)
  - new bfloat16 supported, FP16
  - 16 x 8 x 8, 16 x 8 x 16
  - new TF32 supported, 16 x 8 x 4, 16 x 8 x 8
  - new Double supported, 8 x 8 x 4

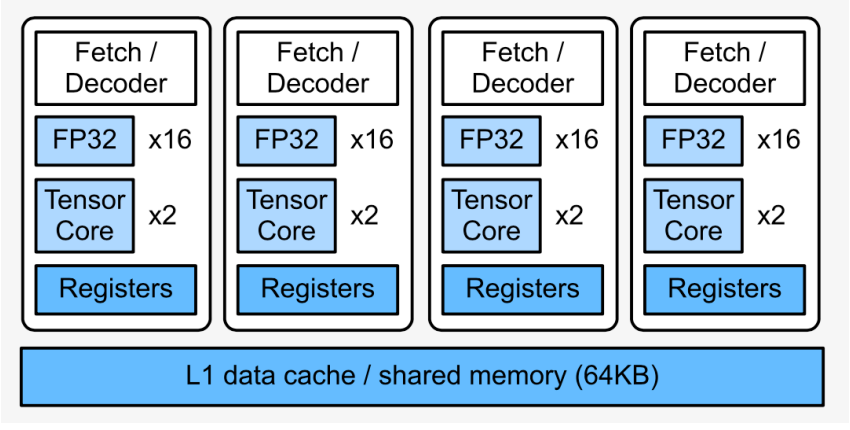
Format of Floating points IEEE754



Data types


Sign
Range
Precision
FP32
8 BITS
23 BITS
TF32 Range
TENSOR FLOAT 32 (TF32)
8 BITS
10 BITS
TF32 Precision
FP16
5 BITS
10 BITS
BFLOAT16
8 BITS
7 BITS

Tesla T4 GPU Architecture



Turing 2080Ti Information

- Architecture: Turing
- SMs: 68
- CUDA Cores/SM: 64
- CUDA Cores/GPU: 4352
- Tensor Cores/SM: 8
- Tensor Cores/GPU: 544

Hierarchical Structure

```
for (int cta_n = 0; cta_n < GemmN; cta_n += CtaTileN) {                  // for each threadblock_y           } threadblock-level concu
  for (int cta_m = 0; cta_m < GemmM; cta_m += CtaTileM) {                //    for each threadblock_x        }

    for (int cta_k = 0; cta_k < GemmK; cta_k += CtaTileK) {              //       "GEMM mainloop" - no unrolling
                                                                        //                     - one iteration of this loop is one
                                                                        //
      for (int warp_n = 0; warp_n < CtaTileN; warp_n += WarpTileN) {    // for each warp_y                  } warp-level parallelism
        for (int warp_m = 0; warp_m < CtaTileM; warp_m += WarpTileM) {  //    for each warp_x               }
                                                                        //
          for (int warp_k = 0; warp_k < CtaTileK; warp_k += WarpTileK) {      //       fully unroll across CtaTileK
                                                                        //         - one iteration of this loop is one "k Group"
                                                                        //
            for (int mma_k = 0; mma_k < WarpTileK; mma_k += MmaK) {     // for each mma instruction         } instruction-level parai
              for (int mma_n = 0; mma_n < WarpTileN; mma_n += MmaN) {   //    for each mma instruction      }
                for (int mma_m = 0; mma_m < WarpTileM; mma_m += MmaM) { //        for each mma instruction  }
                                                                        //
                  mma_instruction(d, a, b, c);                          //            TensorCore matrix computation

                }   // for mma_m
              }   // for mma_n
            }   // for mma_k

          }   // for warp_k
        }   // for warp_m
      }   // for warp_n

    }   // for cta_k
  }   // for cta_m
}   // for cta_n
```

Hierarchical Structure

- The basic triple loop nest computing matrix multiply may be blocked and tiled to match concurrency in hardware, memory locality, and parallel programming models.
- In CUTLASS, GEMM is mapped to NVIDIA GPUs with the structure illustrated by the following loop nest.

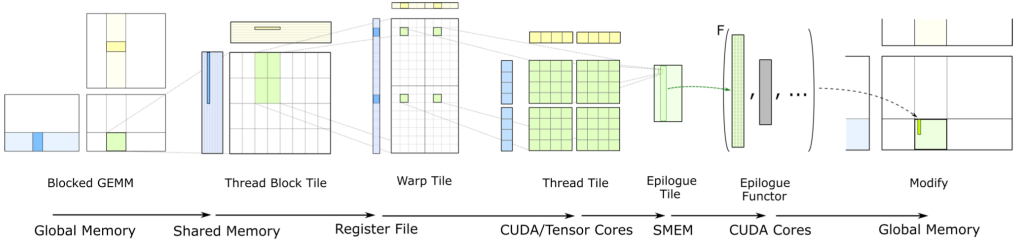
This tiled loop nest targets concurrency among

- threadblocks-level
- warps
- CUDA and Tensor Cores

and takes advantage of memory locality within

- shared memory
- registers

The flow of data in CUTLASS



General Matrix Multiplication

Q1 Learn the code in `./Lab03-CUDA/code` and it contains three folders (vector_add, gemm, wmma)

- Learn the code style and components of `vector_add.cu` file
- Complete all of the code in `gemm` folder
- Try to make your gemm kernel more efficient
  - shared memory
  - tiling size
  - block and thread size

Q2 Learn the `wmma.cu` from the `/Lab03-CUDA/code/wmma` to run it successfully by `compile.sh` script

- Learn the different data type in CUDA programming language such as Float16, Int8
- Learn the basic knowledge of Tensor Core and WMMA in CUDA programming language
- Learn the difference between FLOPs and FLOPS
- Change the tiling size in `wmma.cu` to get the different TFLOPS

THANK YOU!