CPSC/ECE 4780/6780

General-Purpose Computation on Graphical Processing Units (GPGPU)

Lecture 3: Introduction to CUDA

Recaps from Last Lecture

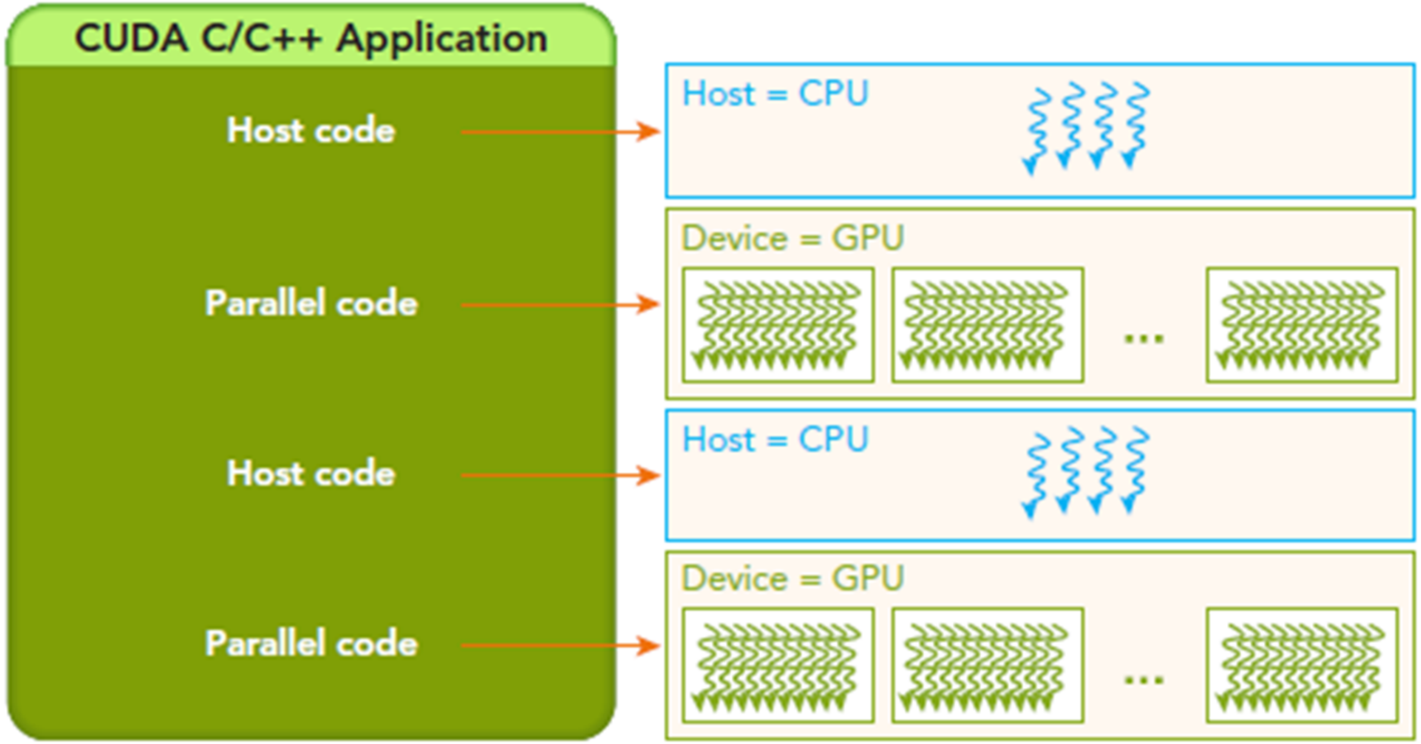
- What is GPU?
- History of GPUs
- Architecture of GPU
- CPU/GPU comparisons
- Why should we use GPUs?
- CPU+GPU acceleration
- GPGPU programming

What is CUDA?

- CUDA – “Compute Unified Device Architecture”
- General-purpose parallel computing platform and programming model
- Created by NVIDIA first in 2007
- Written mostly like C

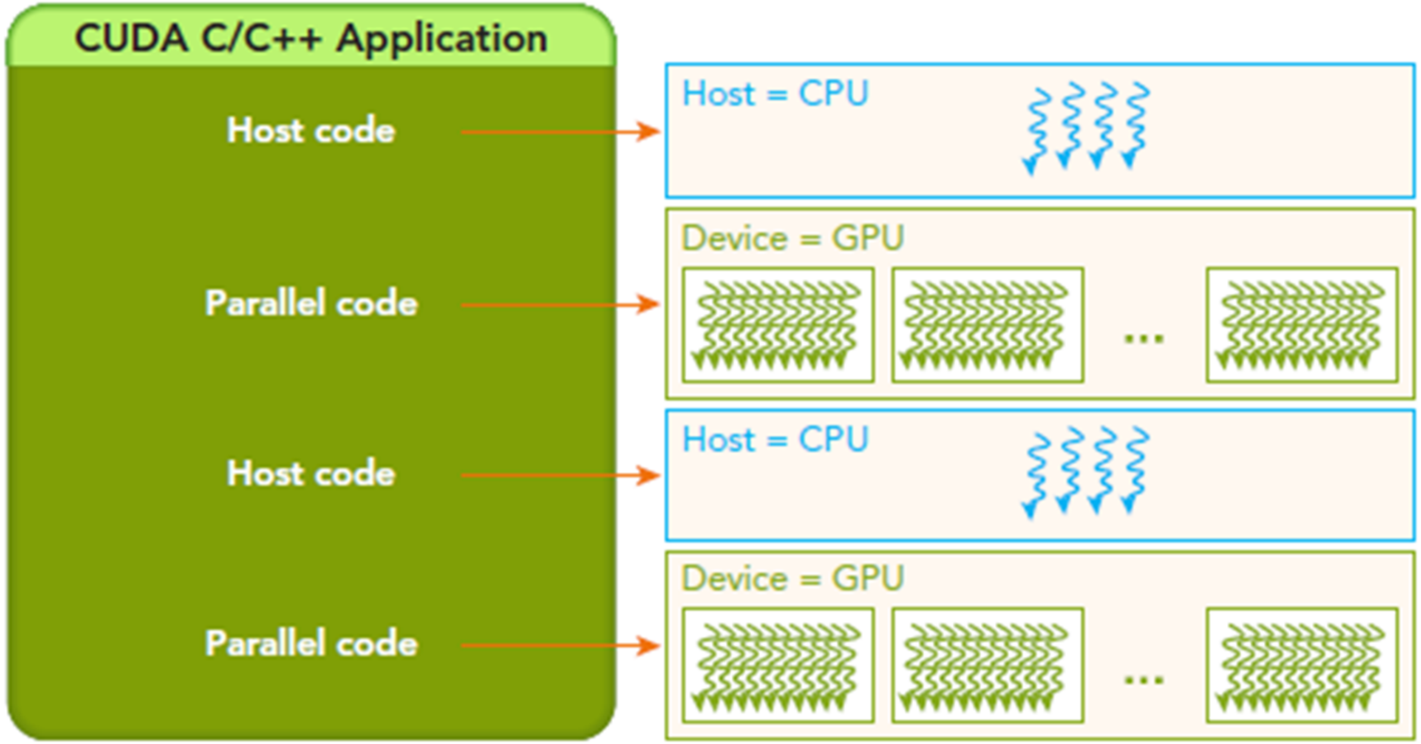
CUDA Programming Structure

- Integrated host + device application C program
  - Host – CPU and its memory
    - Serial or modestly parallel parts
    - Written in ANSI C
  - Device – GPU and its memory
    - Highly parallel parts
    - Written in CUDA C
    - “Kernel”



Processing Flow of a CUDA Program

- Copy input data from CPU memory to GPU memory
- Invoke kernels to operate on the data stored in GPU memory
- Copy data back from GPU memory to CPU memory



Memory Management and Data Transfer

- Host and device memory are separate entities
  - Host pointers point to CPU memory
    - May be passed to/from device code
    - May not be dereferenced in device code
  - Device pointers point to GPU memory
    - May be passed to/from host code
    - May not be dereferenced in host code

STANDARD C FUNCTIONS	CUDA C FUNCTIONS
`malloc`	`cudaMalloc`
`memcpy`	`cudaMemcpy`
`memset`	`cudaMemset`
`free`	`cudaFree`

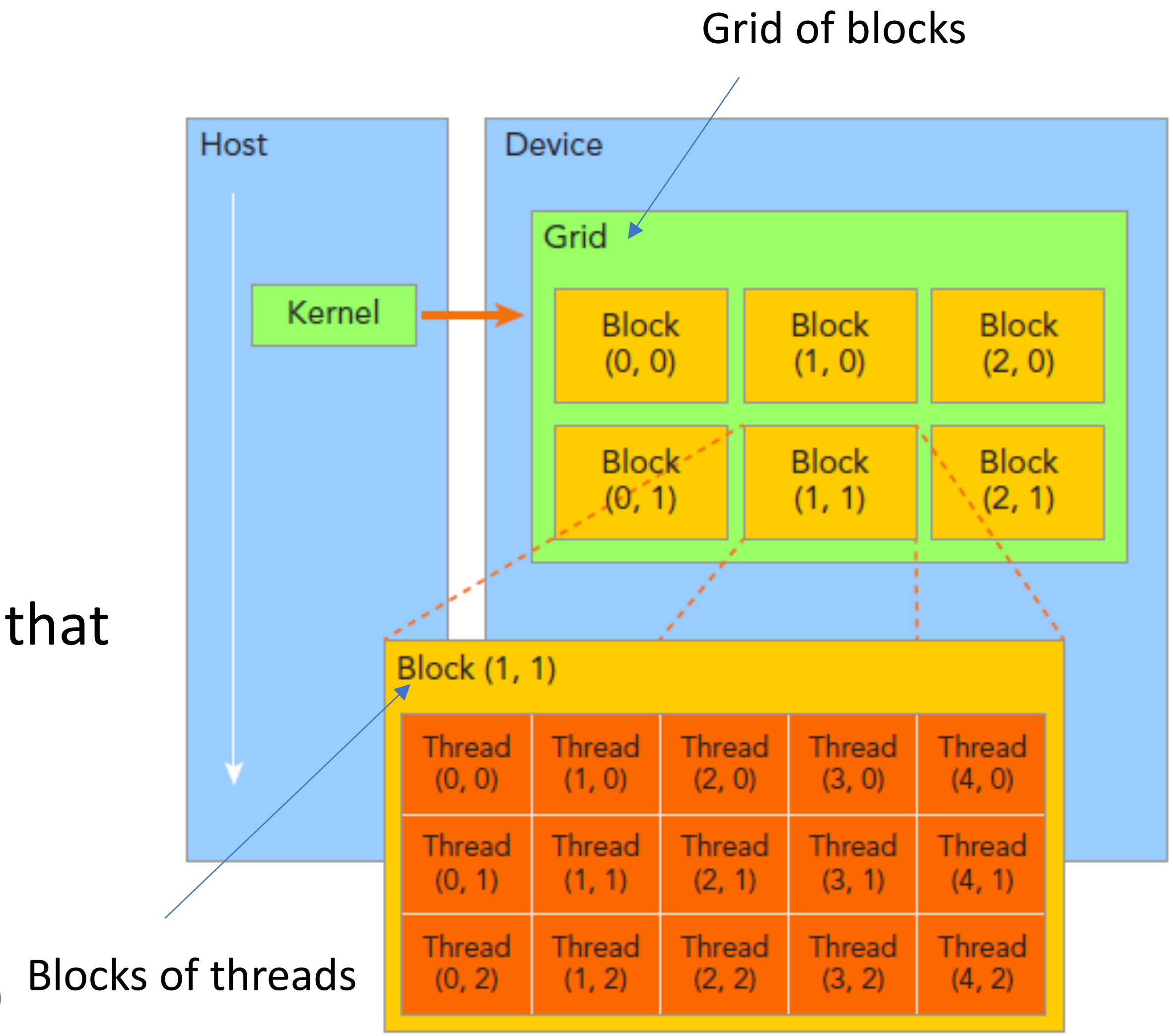
Host and device memory functions

CUDA Function Declaration

- CUDA extensions to C functional declaration
    - __global__: indicates a CUDA kernel function
        - executed on the device
        - Only callable from the host
        - Must have a void return type
    - __device__: indicates a CUDA device function
        - Executed on the device
        - Only callable rom the device
    - __host__: indicates a CUDA host function
        - Executed on the host
        - Only callable from the host

Organizing Threads

- Two-level thread hierarchy
  - Grids of blocks
  - Blocks of threads
- All threads in a grid share the same global memory space
- A thread block is a group of threads that can cooperate with each other by:
  - Block-local synchronization
  - Block-local shared memory
- Threads coordinates:
  - blockIdx (block index within a grid)
  - threadIdx (thread index within a block)
  - Type: unit3 (.x, .y, .z)

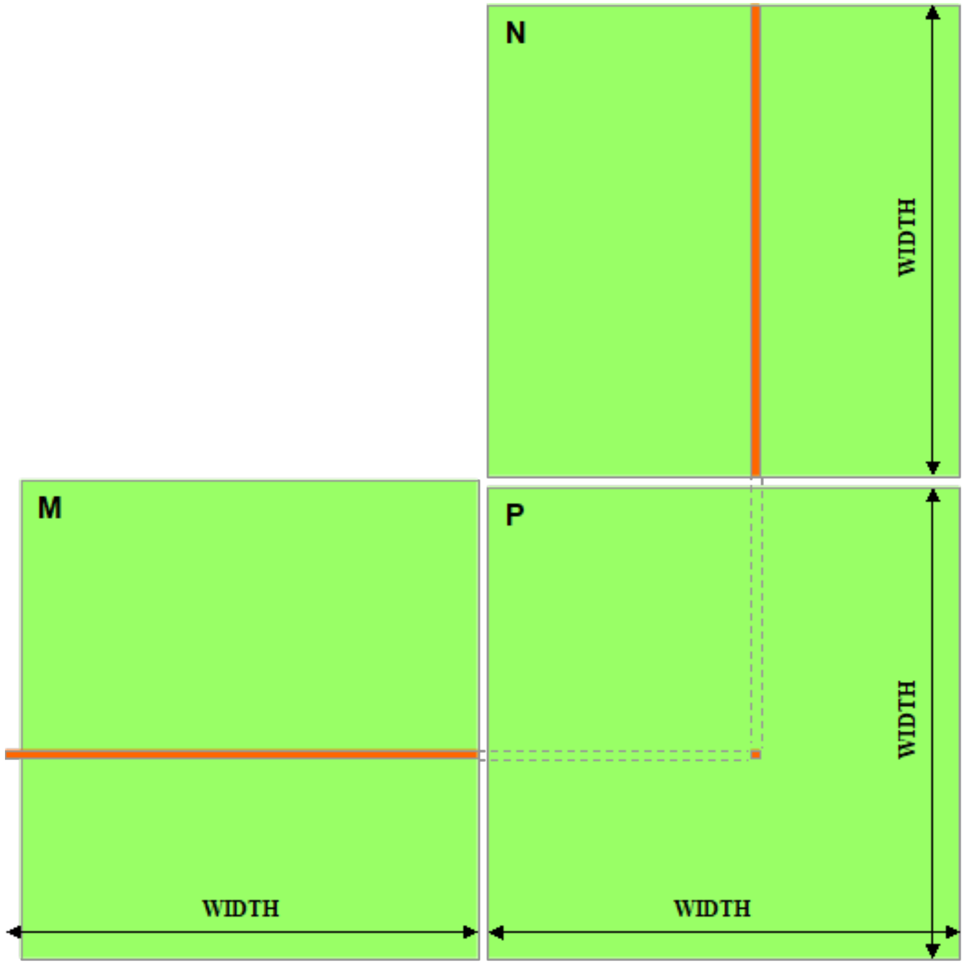


A thread hierarchy structure with a 2D grid containing 2D blocks

Matrix Multiplication on CPU

- M * N => P

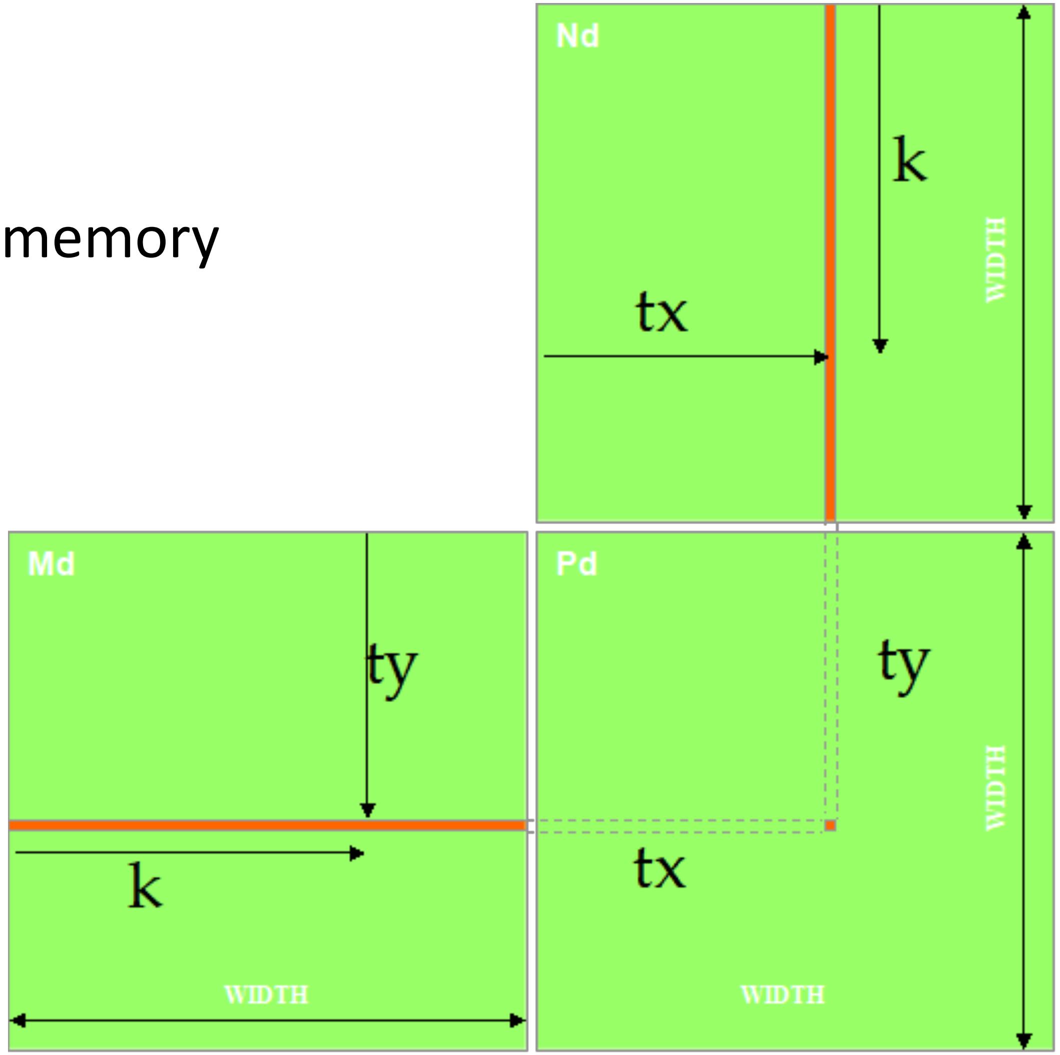
```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
   for (int i = 0; i < Width; ++i)
      for (int j = 0; j < Width; ++j) {
         double sum = 0;
         for (int k = 0; k < Width; ++k) {
            double a = M[i * width + k];
            double b = N[k * width + j];
            sum += a * b;
         }
         P[i * Width + j] = sum;
      }
```



Matrix Multiplication on GPU

- One thread calculates one element of P
- M and N are loaded width times from global memory

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
   int tx = threadIdx.x;
   int ty = threadIdx.y;
   float Pvalue = 0;
   for (int k = 0; k < Width; ++k) {
      float Melement = Md[threadIdx.y*Width+k];
      float Nelement = Nd[k*Width+threadIdx.x];
      Pvalue += Melement * Nelement;
   }
  Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```



CUDA Device Properties

- Get the count of CUDA devices
  - int count;
  - cudaGetDeviceCount(&count);
- Query relevant information of a device
  - cudaDeviceProp prop;
  - cudaGetDeviceProperties(&prop, i);
- Set device property and choose a proper device
  - int dev;
  - cudaDeviceProp prop;
  - prop.major = 1;
  - prop.minor = 3
  - cudaChooseDevice(&dev, &prop);
  - cudaSetDevice(dev);

Coding Examples

- Coding
  - First CUDA program: Hello World
  - Add two numbers
  - Add two vectors
    - By blockIdx
    - By threadIdx
    - Combined
  - Query device property
- Compilation: use nvcc
- Makefile

Palmetto is comprised of 2021 compute nodes (totalling 23072 CPU cores), and features:

- 2021 compute nodes, totaling 23072 cores
- 595 nodes equipped with 2x NVIDIA Tesla GPUs (2 per node); 103 nodes each have 2x NVIDIA Tesla V100 GPUs (2 per node)

Login with command:

- ssh username@login.palmetto.clemson.edu

https://www.palmetto.clemson.edu/palmetto/userguide_palmetto_overview.html

Request a specific GPU (m2075, m2070q, k20, k40, p100, or v100) on Palmetto with command:

- qsub -I -l select=1:ncpus=1:ngpus=1:gpu_model=k20:mem=2gb,walltime=2:00:00

https://www.palmetto.clemson.edu/palmetto/userguide_howto_use_gpus.html

Load CUDA module:

- module load cuda-toolkit

GPU device query:

- /software/cuda-toolkit/8.0.44/samples/1_Utilities/deviceQuery/deviceQuery

Compile .cu code with “nvcc”, e.g.,

- nvcc helloWorld.cu
- nvcc –o helloWorld helloWorld.cu

Assignment #1: The Big Dot

- The dot product of two vectors $a = (a_{0,}\ a_{1,}\ \ldots, a_{n-1})$ and $b = (b_{0,}\ b_{1,}\ \ldots, b_{n-1})$, written $a \cdot b$, is simply the sum of the component-by-component products:

$$a \cdot b = \sum_{i=0}^{n-1} a_i \times b_i$$

Dot products are used extensively in computing and have a wide range of applications. For instance, in 3D graphics (n = 3), we often make use of the fact that $a \cdot b = |a||b|cos\theta$, where | | denotes vector length and θ is the angle between the two vectors.

Assignment #1: The Big Dot

- Write CUDA code to compute in parallel the dot product of two (possibly large N = 100,000, or N = 1024*1024) random single precision floating point vectors;
- Write two functions to compute the results on the CPU and GPU, and compare the two results to check for correctness (1.0e-6);
  - float *CPU_big_dot(float *A, float *B, int N);
  - float *GPU_big_dot(float *A, float *B, int N);
- Print performance statistics with timer function;
  - CPU: Tcpu = Total computation time for CPU_big_dot();
  - GPU: Tgpu = Total computation time for GPU_big_dot();
    - Memory allocation and data transfer from CPU to GPU time
    - Kernel execution time
    - Data transfer from GPU to CPU time
  - Speedup = GPU/CPU
- Analyze the performance results in a few sentences.
  - Which one runs faster?
  - What's the reason for that? Problem size, overhead, etc.

Assignment #1: The Big Dot

- Timer functions
  - #include <sys/time.h>
  - long long start_timer() {

    ```
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000 + tv.tv_usec;
    ```

    }
  - long long stop_timer(long long start_time, char *name) {

    ```
    struct timeval tv;
    gettimeofday(&tv, NULL);
    long long end_time = tv.tv_sec * 1000000 + tv.tv_usec;
    Printf(“%s: %.5f sec\n”, name, ((float) (end_time – start_time)) / (1000 * 1000));
    return end_time – start_time;
    ```

    }