1

The Compute Unified Device Architecture (CUDA)

Oregon State University

Mike Bailey

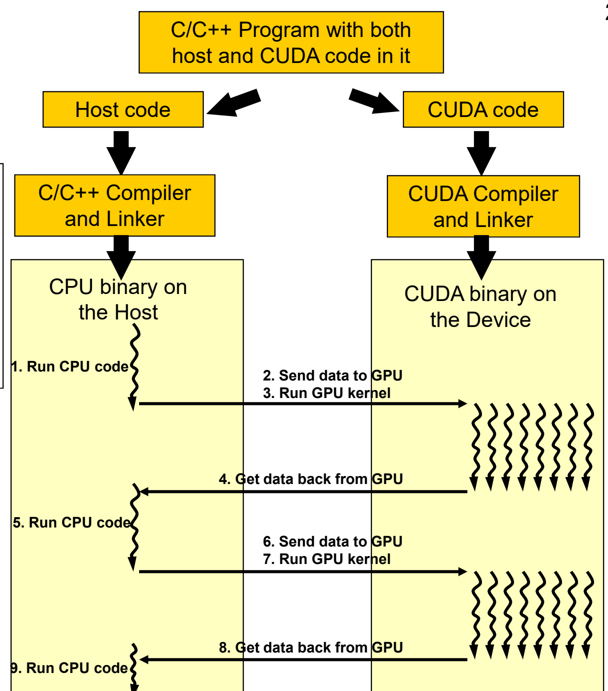
mjb@cs.oregonstate.edu

Oregon State University
Computer Graphics

cuda.pptx

2

The CUDA Paradigm



CUDA is an NVIDIA-only product. It is very popular, and got the whole GPU-as-CPU ball rolling, which has resulted in other packages like OpenCL.

CUDA also comes with several libraries that are highly optimized for applications such as linear algebra and deep learning.

Oregon State University
Computer Graphics

3

CUDA wants you to break the problem up into Pieces

If you were writing in **C/C++,** you would say:

```
void
ArrayMult( int n, float *a, float *b, float *c)
{
        for ( int i = 0;  i < n;  i++ )
                c[ i ] = a[ i ] * b[ i ];
}
```

If you were writing in **CUDA**, you would say:

```
__global__
void
ArrayMult( float *dA, float *dB, float *dC )
{
        int gid = blockIdx.x*blockDim.x + threadIdx.x;
        dC[gid] = dA[gid] * dB[gid];
}
```

Think of this as having an implied for-loop around it, looping through all possible values of *gid*

Oregon State University
Computer Graphics

mjb – March 28, 2020

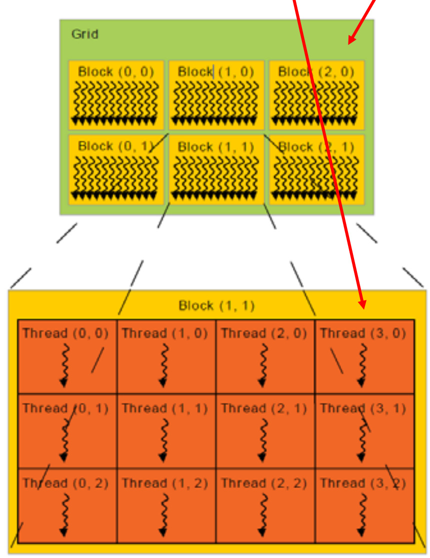
4

Organization: Blocks are Arranged in Grids

- The GPU's workload is divided into a **Grid of Blocks**
- Each Block's workload is divided into a **Grid of Threads**

Grid of Blocks

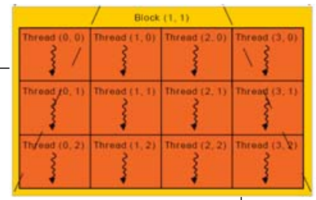
Grid of Threads



Oregon State University
Computer Graphics

mjb – March 28, 2020

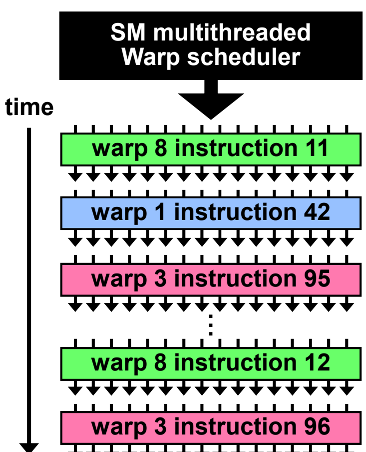
A Block is made up of a Grid of Threads



- The threads in a block each have *Thread ID* numbers within the Block
- Your CUDA program will use these Thread IDs to select work to do and pull the right data from memory
- Threads share data and synchronize while doing their share of the work
- Every **32** threads constitute a *"Warp"*. Each thread in a Warp simultaneously executes the same instruction on different pieces of data.
- But, it is likely that a Warp's execution will need to stop at some point, waiting for a memory access. This would make the execution go idle – bad! So, it is worthwhile to have multiple Warps worth of threads available so that when one Warp blocks, another Warp can be swapped in.
- The threads in a *Thread Block* can cooperate with each other by:
  – Synchronizing their execution
  – Efficiently sharing data through a low latency shared memory
- Threads from different blocks cannot cooperate

Scheduling

- The hardware implements low-overhead Warp switching



– A Warp whose next instruction has operands ready for consumption is eligible to be executed.

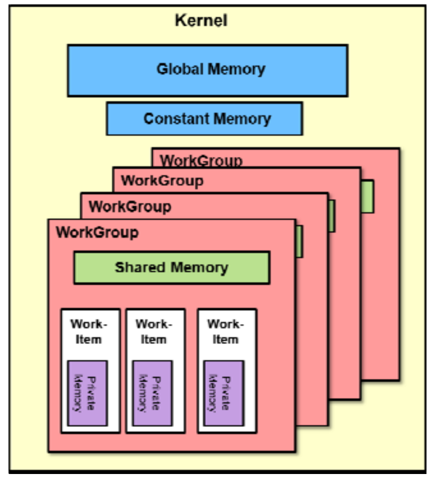
– All threads in one Warp execute the same instruction at any given time, but on different data.

– Threads in different Warps will usually be executing different instructions at any given time

This tells you that there needs to be a bunch of Warps to work on so that something is always ready to run
If you can help it, these should be multiples of 32.

Threads Can Access Various Types of Storage

- Each thread has access to:
  - Its own R/W per-thread registers
  - Its own R/W per-thread private memory
- Each thread has access to:
  - Its block's R/W per-block shared memory
- Each thread has access to:
  - The entire R/W per-grid global memory
  - The entire read-only per-grid constant memory
  - The entire read-only per-grid texture memory
- The CPU can read and write global and, constant memories



Different Types of CUDA Memory

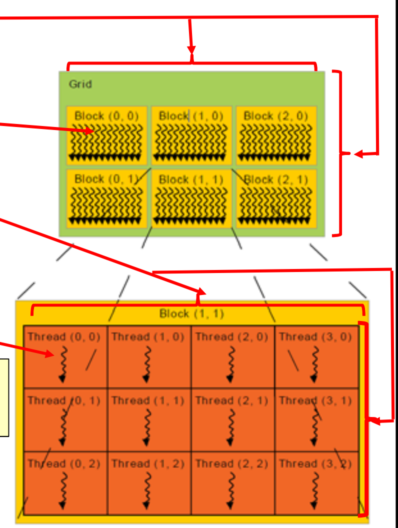
Memory	Location	Who Uses
Registers	On-chip	One thread
Private	On-chip	One thread
Shared	On-chip	All threads in that block
Global	Off-chip	All threads + Host
Constant	Off-chip	All threads + Host

Thread Rules

- Each Thread has its own registers and private memory
- Each Block can use at most some maximum number of registers, divided equally among all Threads
- Threads can share local memory with the other Threads in the same Block
- Threads can synchronize with other Threads in the same Block
- Global and Constant memory is accessible by all Threads in all Blocks
- 192 or 256 are good numbers of Threads per Block (multiples of the Warp size)

A CUDA Thread can Query where it Fits in its "Community" of Threads and Blocks

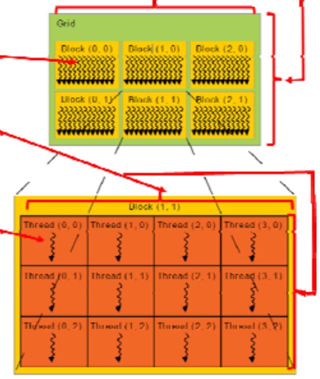
- `dim3 gridDim;`
  - Dimensions of the blocks in this grid
- `dim3 blockIdx;`
  - This block's indexes within this grid
- `dim3 blockDim;`
  - Dimensions of the threads in this block
- `dim3 threadIdx;`
  - This thread's indexes within the block



Note: It is as if dim3 is defined as:
  typedef int[3] dim3;
(it's not really – it is actually defined within the CUDA compiler)

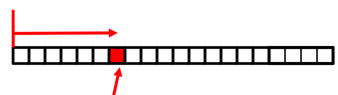
A CUDA Thread needs to know where it Fits in its “Community” of Threads and Blocks

- **dim3 gridDim;**
  - Dimensions of the blocks in this grid
- **dim3 blockIdx;**
  - This block's indexes within this grid
- **dim3 blockDim;**
  - Dimensions of the threads in this block
- **dim3 threadIdx;**
  - This thread's indexes within the block



For a 1D problem:
int blockThreads = blockIdx.x*blockDim.x;
int gid = blockThreads + threadIdx.x;
C[gid] = A[gid]*B[gid];

For a 2D problem:
int blockNum = blockIdx.y*gridDim.x + blockIdx.x;
int blockThreads = blockNum*blockDim.x*blockDim.y;
int gid = blockThreads + threadIdx.y*blockDim.x + threadIdx.x;
C[gid] = A[gid]*B[gid];



Types of CUDA Functions

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	GPU	GPU
`__global__ void KernelFunc()`	GPU	Host
`__host__ float HostFunc()`	Host	Host

`__global__` defines a kernel function – it must return **`void`**

Note: “__” is *2* underscore characters

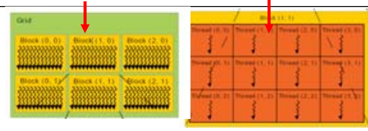
13

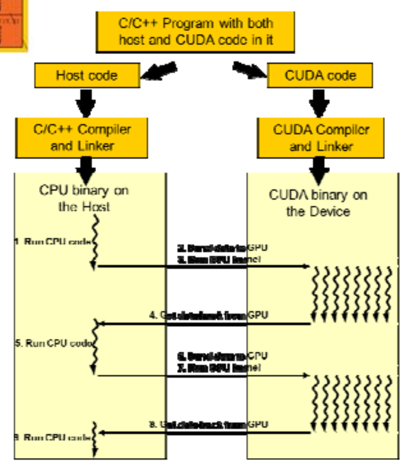
The C/C++ Program Calls a CUDA Kernel using a Special <<<…>>> Syntax

These are called "chevrons"

dim3 dim3

KernelFunction<<< NumBlocks, NumThreadsPerBlock >>>(arg1, arg2, …) ;





Note that this is just like calling the C/C++ function:
KernelFunction(arg1, arg2, …) ;
except that we have designated it to run on the GPU with a particular block/thread configuration.

Oregon State University
Computer Graphics

mjb – March 28, 2020

14

Running a CUDA Program on our Linux systems: The *Makefile* we use

```
CUDA_PATH       =    /usr/local/apps/cuda/cuda-10.1
CUDA_BIN_PATH   =    $(CUDA_PATH)/bin
CUDA_NVCC       =    $(CUDA_BIN_PATH)/nvcc

arrayMul:            arrayMul.cu
                     $(CUDA_NVCC) -o arrayMul  arrayMul.cu
```

This is the path where the CUDA tools are loaded on our Oregon State University systems. Yours is probably different.

Note: if you are trying to run CUDA on your own Visual Studio system, make sure your machine has the CUDA toolkit installed. It is available here:

https://developer.nvidia.com/cuda-downloads

Oregon State University
Computer Graphics

mjb – March 28, 2020

Creating your own CUDA Visual Studio Folder

1. Un-zip the *ArrayMul2019.zip* file into its own folder.
2. *Rename that folder to what you want it to be.*
3. Rename *arrayMul.cu* to whatever you want it to be (keeping the .cu extension). Without the .cu extension, we will call this the **basename**.
4. Rename the *.sln* and *.vcxproj* files to have the same basename as your *.cu* file has.
5. Edit the **.sln* file. Replace all occurrences of "*arrayMul*" to what the **basename** is.
6. Edit the *.*vcxproj* file. Replace all occurrences of "*arrayMul*" with the **basename**. Replace all occurrences of *ArrayMul2019* with whatever you renamed the folder to.
7. In the .cu file, rename the CUDA function from *ArrayMul* to whatever you want it to be. Do this twice, once in the definition of the function and once in the calling of the function.
8. Now modify the CUDA code to perform the computation you require.

Note: if you are trying to run CUDA on your own Visual Studio system, make sure your machine has the CUDA toolkit installed. It is available here:

https://developer.nvidia.com/cuda-downloads

Using Multiple GPU Cards with CUDA


```
int  deviceCount;
cudaGetDeviceCount(  &deviceCount  );

. . .

int device;          // 0 <= device <= deviceCount - 1
cudaSetDevice( device );
```