Nvidia’s Compute Unified Device Architecture (CUDA)

The CUDA Paradigm

CUDA wants you to break the problem up into Pieces

A Block is made up of a Grid of Threads

Scheduling

Organization: Blocks are Arranged in Grids

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

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

void

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

If you were writing in CUDA, you would say:

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

The threads in a block each have Thread ID numbers within the Block

The hardware implements low-overhead Warp switching

SM threaded Warp scheduler

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 (including textures)
- The CPU can read and write global and constant memories

**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

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

```c
// A CUDA Thread needs to know where it Lives in its “Community” of Threads and Blocks

int blockThreads = blockIdx.x*blockDim.x;
int gid = blockThreads + threadIdx.x;
```

Types of CUDA Functions

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

Note: “__” is 2 underscore characters
The C/C++ Program Calls a CUDA Kernel using a Special Syntax

```
KernelFunction<<< NumBlocks, NumThreadsPerBlock >>>( arg1, arg2, ... );
```

These are called “chevrons”

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.

---

One of my own Experiments with Number of Threads Per Block

```
KernelFunction<<< NumBlocks, NumThreadsPerBlock >>>( arg1, arg2, ... );
```

Performance

Dataset Size

Number of Threads per Block

```
NumBlocks = DataSetSize / NumThreadsPerBlock
```

---

Getting CUDA Programs to Run under Linux

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

```
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
```

We also have the CUDA-11 and CUDA-12 tools loaded for your use. You can use them if you want. But, given the wide breadth of different Nvidia cards around campus, CUDA-10 seems to be the one that will run everywhere! I recommend you use it.

---

Getting CUDA Programs to Run under Visual Studio

1. Install Visual Studio if you haven’t already. If you are an OSU student, go to:

   [https://azureforeducation.microsoft.com/devtools](https://azureforeducation.microsoft.com/devtools)

   Click the blue Sign In button on the right.

   Login using your ONID@oregonstate.edu username and password.

2. Install the CUDA toolkit. It is available here:

Then, in this templates box, type: CUDA

After a few seconds, you will then see this. Click Next.

1. Navigate to the folder you want to contain this project folder.
2. Give the name you want for the folder and project
3. Leave this box checked.
4. Click Create

Using CUDA and OpenMP Together

This is the Makefile we use on Linux:

```
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
-Xcompiler -fopenmp
```

Or, on Linux, but without the Makefile syntax:

```
$(CUDA_NVCC) -o arrayMul arrayMul.cu -Xcompiler -fopenmp
```

Or, in Visual Studio:

1. Go to the Project menu → Project Properties
2. Change the setting Configuration Properties → C/C++ → Language → OpenMP Support to “Yes (openmp)"

We also have the CUDA-11 and CUDA-12 tools loaded for your use. You can use them if you want. But, given the wide breadth of different Nvidia cards around campus, CUDA-10 seems to be the one that will run everywhere! I recommend you use it.
Using Multiple GPU Cards with CUDA

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

...  

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