Jerome Mueller

Zach Goodwyn

**Cuda Tutorial for Current CentOS System**

**Setup:**

* Make sure CUDA is installed on the system and keep track of where
* Create a file with the extension .cu
* Add #include<cuda.h>

**Construction:**

* Declare any data types you would like to send back and forth between the host and the GPU need to be pointers
* You can allocate memory for them using on the host using malloc like any c program
* To allocate memory on the GPU use cudaMalloc

**Example:**

* + Host, float\* variable\_Name = (float\*)malloc(num\_Elements \* size\_of\_type)
  + GPU, cudaMalloc((void\*\*)&var\_Name, (num\_Elements \* size\_of\_type))
* When allocating with cudaMalloc the return value is not the pointer
* Do not forget the & in the cudaMalloc statement
* We cast a void\*\* because that is the data type it is expecting
* To transfer memory between to host and the GPU use cudaMemcpy(dest\_Var\*, source\_Var\*, size\_mem\_trans, trans\_Type)

**Example:**

* int\* host\_Data;
* int\* GPU\_Data;
* cudaMemcpy(GPU\_data, host\_Data, (sizeof(int)\*num\_Elements), cudaMemcpyHostToDevice);
* //Have GPU operate on data
* cudaMemcpy(host\_Data, GPU\_Data, (sizeof(int)\*num\_Elements), cudaMemcpyDeviceToHost);
* cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are built in CUDA variables along with cudaMemcpyHostToHost and cudaMemcpyDeviceToDevice. The later does not work across GPUs, but only to the same one
* Any Pointers declared using cudaMalloc can be freed using cudaFree(var\*)

**Functions:**

* Any function that will be executed in parallel needs to be declared as \_\_global\_\_ before its return type
* Note that there are double underscores on both sides

**Example Declaration:**

* \_\_global\_\_ void multiFoo(var\*)
* {
* //do stuff in parallel
* }
* Called like any other function except there are <<< and >>> surrounding the number of blocks and the number of threads per block
* This is placed between the function name and the argument list

**Example Call:**

* multiFoo<<<1,5>>>(var\*);
* //this launches one block with five threads, each thread executes multiFoo
* within the function you can access threadIdx.x and blockIdx.x wich contain the thread ID and the block ID unique to each thread

**Compilation:**

* Compilation must be done with the NVCC compiler which is included in the CUDA toolkit
* To execute this give the pathname where the complier resides followed by the name of your CUDA file and any other relevant parameters or flag
* The default location is /user/local/cuda-5.0/bin/nvcc

**Example:**

* /user/local/cuda-5.0/bin/nvcc test.cu –o test