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. We have provided a sample file: “cuda\_test.cu.”
* Add #include<cuda.h>

**Construction:**

* Declare any data types you would like to send back and forth between the host and the GPU as pointers
* You can allocate memory for them 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. The return value indicates whether the cudaMalloc call succeeded or failed.
* Do not forget the & in the cudaMalloc statement
* We cast a void\*\* because that is the data type cudaMalloc is expecting
* To transfer memory between the 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. cudaMemcpyDeviceToDevice copies data from a GPU to another location on the same GPU.
* 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*

}

* Functions are 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 which 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

**Running the Program:**

* Depending on the video-card you are using, you may have to run the program from the root terminal to execute non-trivial CUDA programs. This is because when your program executes a kernel function, the computer starts a timer, and will terminate your kernel if it monopolizes the GPU for too long a period of time (on our system, a span of about 40 seconds). This will always be an issue if your GPU is used for video display. If this is not a problem on your system or for your purposes, you can skip to the second-to- last step in this section.
* From the command prompt, login as root by executing the command **su**, then typing the password for the root user.
* Execute the command **telinit 3**. This should bring you to the root terminal. If it does not, hit ctrl+alt+backspace.
* Navigate to the directory where your program is located.
* To execute the program, just run it as normal. For example run ./test from the command line if you used the above command to compile the program.
* To bring yourself back to the gui, execute the command **startx**.

**Debugging Tips:**

* Because the kernel code is executed on the GPU, you do not have the luxury of directly writing values to stdout from the kernel to catch simple errors. It is imperative that you write and test your kernel function in manageable chunks.
* If the kernel fails to complete execution, your next CUDA call may return a cudaError\_t unrelated to the actual call. In this case, the cudaError\_t will reflect the error that occurred in the kernel rather than the CUDA function call.
  + **Example**:

launch\_Kernel<<<num\_Blocks, threads\_Per\_Block>>>(arguments);

cudaError\_t err = cudaMemcpy(dest, src, size, cudaMemcpyDeviceToHost);

*//”err” might contain an error resulting from launch\_Kernel if that function //encountered an error (as opposed to cudaMemcpy causing the error)*

* If in the above example, the cudaError\_t is an “unspecified launch error”, it likely means that the launch\_Kernel function segfaulted, or that your kernel was terminated by an outside process (see “Running the program”).
* In all cases, consider executing your kernel with only a few blocks and a few threads per block at first, perhaps even with only one block and thread for the first test run. You can always try with more later.