NVIDIA CUDA Programming Massively Parallel Processors

Hands-On Labs

The IMPACT Research Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

The purpose of the hands-on labs is meant to assist a three-day course for clarifying CUDA programming concepts. It starts from printing “Hello World!” strings, followed by several well-known problems from a simple matrix multiplication to teach basic programming concepts and performance tuning.

Laboratory 0: Setup and Hello World

Objective

This lab is purposed to check your setup environment to make sure you can compile and run CUDA programs.

1. Preliminary work

Step 1:

Use an SSH program to log into qp.ncsa.uiuc.edu, using the training account login and initial password given to you. On the first login you will be required to set your own password, which will remain active during the entire workshop.

After the system accepts your new password, it will end the current login session, and you are required to log in again with your new password to gain access.

Your home directory can be organized any way you like. To unpack the SDK framework including the code of all of the lab assignments, execute the unpack command in the directory you would like the SDK created.

```
%> tar -zxvf ~stratton/GPU_WORKSHOP_SDK.tgz
```

Step 2:

Go to the lab0 directory and make sure it exists and is populated.

```
%> cd NVIDIA_CUDA_SDK/projects/lab0-hello_world
```

There should be two source files at least.

- helloword.cu
- helloword_kernel.cu

2. Make the first CUDA program

Compile using the emurelease or emudebug configuration. The executable will be placed in NVIDIA_CUDA_SDK/bin/linux/emuXXX/lab0-hello_world and you should see emulated 16 threads printing "Hello World!" to the screen, as shown below.

If the program compiles and runs correctly, then you should be prepared to address the future lab assignments.

Note: printf only works in an emulation mode. All printfs and I/O related code has to be removed from kernel code when running on the actual G80 card.

```
%> make emu=1
%> ../../bin/linux/emurelease/lab0-hello_world
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 0, 0}
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 1, 0}
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 0, 1}
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 1, 1}
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 0, 2}
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 1, 2}
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 0, 3}
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 1, 3}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 0, 0}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 1, 0}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 0, 1}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 1, 1}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 0, 2}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 1, 2}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 0, 3}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 1, 3}
```

Laboratory 1: Matrix multiplication using CPU+GPU

Objective

This lab introduces a famous and widely-used example application on the parallel programming field, namely the matrix multiplication. You will complete key portions of the program in the CUDA language to compute this widely-applicable kernel.

The source files for lab 1 are in "NVIDIA_CUDA_SDK/projects/lab1-matrixmul", and should compile completely, although with warnings, as they are.

Modify the given CUDA programs

Step 1:

Edit the runTest(…) function in "matrixmul.cu" to complete the functionality of the matrix multiplication on the host. Follow the comments in the source code to complete these key portions of the host code.

Code segment 1: Copying the input matrix data from the host memory to the device memory.

Code segment 2: Set the kernel launch parameters, and invoke the kernel.

Code segment 3: Copy the result matrix from the device memory to the host memory.

Step 2:

Edit the matrixMul(…) function in "matrixmul_kernel.cu" to complete the functionality of the matrix multiplication on the device. Follow the comments in the source code to complete these key portions of the device code.

Code segment 4:

- Define the output index where the thread should output its data.
- Iterate over elements in the vectors for the thread's dot product.
- Multiply and accumulate elements of M and N

The source code does not need to be changed elsewhere.

Step 3:

Compile using the provided solution files or Makefiles. Simply typing the command "make" in the lab1-matrixmul directory will compile the source program for you. To make sure that the most recent source files are used, we strongly encourage you to execute the command "make clean" before rebuilding your project again with "make".

Submit the executable to the queuing system using the "submitjob" script.

```
%> submitjob ../../bin/linux/release/lab1-matrixmul
```

You will see confirmation of your job submission and a job ID. You can use the commands shown in the confirmation to monitor your job while it is in progress. After the job completes, you should see .oXYZ and .eXYZ files containing the stdout and stderr outputs from your program, respectively. Your stdout file should look like this…

```
Allocate host memory for matrices M and N.
Allocate memory for the result on host side
Initialize the input matrices.
Allocate device memory.
Copy host memory to device.
Allocate device memory for results.
Setup execution parameters.
Execute the kernel.
Check if kernel execution generated an error.
Copy result from device to host.
Processing time:
Do checksum.
sum =
sum should be 2.96393e+17 for matrix_3072.bin.
Clean up memory
```

Check that your checksum result matches the expected result for the input matrix. matrix_3072.bin is used by default. Record your runtime to compare with future lab results.

Laboratory 2: Matrix multiplication using CPU+GPU with tiling and shared memory

Objective

This lab is an enhanced version from the previous one, which adds the features of shared memory and synchronization between threads in a block. The device shared memory is allocated for storing the sub-matrix data for calculation, and threads share memory bandwidth which was previously overtaxed.

The source files for lab 2 are in "NVIDIA_CUDA_SDK/projects/lab2-matrixmul", and should compile completely, although with warnings, as they are.

Modify the given CUDA programs

Step 1:

Edit the matrixMul(…) function in "matrixmul_kernel.cu" to complete the functionality of the matrix multiplication on the device. The host code has already been completed for you. The source code need not be modified elsewhere.

Code segment 1: Determine the update values for the tile indices in the loop.

Code segment 2:

- Load a tile from M and N into the shared memory arrays
- Synchronize the threads
- Multiply the two tiles together, each thread accumulating the partial sum of a single dot product.
- Synchronize again

Step 2:

Compile using the provided solution files or Makefiles. Simply typing the command "make" in the lab2-matrixmul directory will compile the source program for you. To make sure that the most recent source files are used, we strongly encourage you to execute the command "make clean" before rebuilding your project again with "make".

Submit the executable to the queuing system using the "submitjob" script.

```
%> submitjob ../../bin/linux/release/lab2-matrixmul
```

You will see confirmation of your job submission and a job ID. You can use the

commands shown in the confirmation to monitor your job while it is in progress. After the job completes, you should see .oXYZ and .eXYZ files containing the stdout and stderr outputs from your program, respectively. Your stdout file should look like this…

```
Allocate host memory for matrices M and N.
Allocate memory for the result on host side.
Initialize the input matrices.
Allocate device memory.
Copy host memory to device.
Allocate device memory for results.
Setup execution parameters.
Execute the kernel.
Check if kernel execution generated an error.
Copy result from device to host.
Processing time:
Do checksum.
sum =
sum should be 2.96393e+17 for matrix_3072.bin.
Clean up memory
```

Check that your checksum result matches the expected result for the input matrix. matrix_3072.bin is used by default. Record your runtime to compare with future lab results. This program also writes the ouput matrix to a binary file.

Laboratory 3: Matrix Multiplication with Performance Tuning

Objective

This lab provides more flexibility for you to tune the matrix multiplication performance. You can give different values for loop unrolling degree, block size, tiling factors, and turn on/off the spill and prefetch functions to see the performance differences.

You can examine the source code in matrixmul.cu and matrixmul_kernel.cu and see how these switches are applied to change the source code. The given programs are correct. You should be able to compile them without errors.

The source files for lab 3 are in "NVIDIA_CUDA_SDK/projects/lab3-matrixmul", and should compile completely and correctly as they are.

Modify the given CUDA programs

Step 1:

Edit the preprocessor definition in matrixmul.h to vary the kernel optimization parameters. The source code does not need to be edited elsewhere. The project is correct as given, and should compile without errors.

Step 2:

Compile using the provided solution files or Makefiles. Simply typing the command "make" in the lab3-matrixmul directory will compile the source program for you. To make sure that the most recent source files are used, we strongly encourage you to execute the command "make clean" before rebuilding your project again with "make".

Submit the executable to the queuing system using the "submitjob" script.

```
%> submitjob ../../bin/linux/release/lab3-matrixmul
```

You will see confirmation of your job submission and a job ID. You can use the commands shown in the confirmation to monitor your job while it is in progress. After the job completes, you should see .oXYZ and .eXYZ files containing the stdout and stderr outputs from your program, respectively. Your stdout file should look like this…

```
Allocate host memory for matrices A and B.
Allocate memory for the result on host side.
```

```
Initialize the input matrices.
Allocate device memory.
Copy host memory to device.
Allocate device memory for results.
Setup execution parameters.
Execute the kernel.
Check if kernel execution generated an error.
Copy result from device to host.
Processing time:
Do checksum.
sum =
sum should be 2.96393e+17 for matrix_3072.bin.
Clean up memory
```

Check that your checksum result matches the expected result for the input matrix. matrix_3072.bin is used by default. Although the program should function correctly as it is, is also writes the ouput matrix to a binary file, and you may compare it with the same correct output file to verify its correctness.

Step 3:

Find the combination of the parameters that gets the best performance with some experimentation. Record the best performance you can achieve, and try to explain why the selection of parameters you chose was a good one.