# LAB 4: GPUs and CUDA

## GOAL

Research into parallel algorithms often involves development of multiple algorithms and implementations for solving a given problem. These variations are then tested and compared to each other in order to highlight certain computational and architectural features. This lab is meant as a small demonstration of this process. You can always refer to the reduction problems that we discussed in class to remind yourself of some of the parallelization challenges, whether there is enough parallelism that can be exploited in an algorithm, are there different amount of parallelism in different algorithm designs, when will it be effective, etc.

A lecture about GPU/CUDA will be given to introduce you to this lab.

## SUBMISSION

- Provide answers for each question in **REQUIREMENTS** section. Place your answers in a word document (.docx) and name it as "Lab4_LastName_FirstName.docx."
- Submit your document and codes on Canvas (do not zip/compress your files).
- This lab is worth 40 points + 5 extra points.

## REQUIREMENTS

1. **(10 pts)** Understand the given CUDA codes for matrix multiplication programs (mm-cuda1 and mm-cuda2) and answers the following questions:

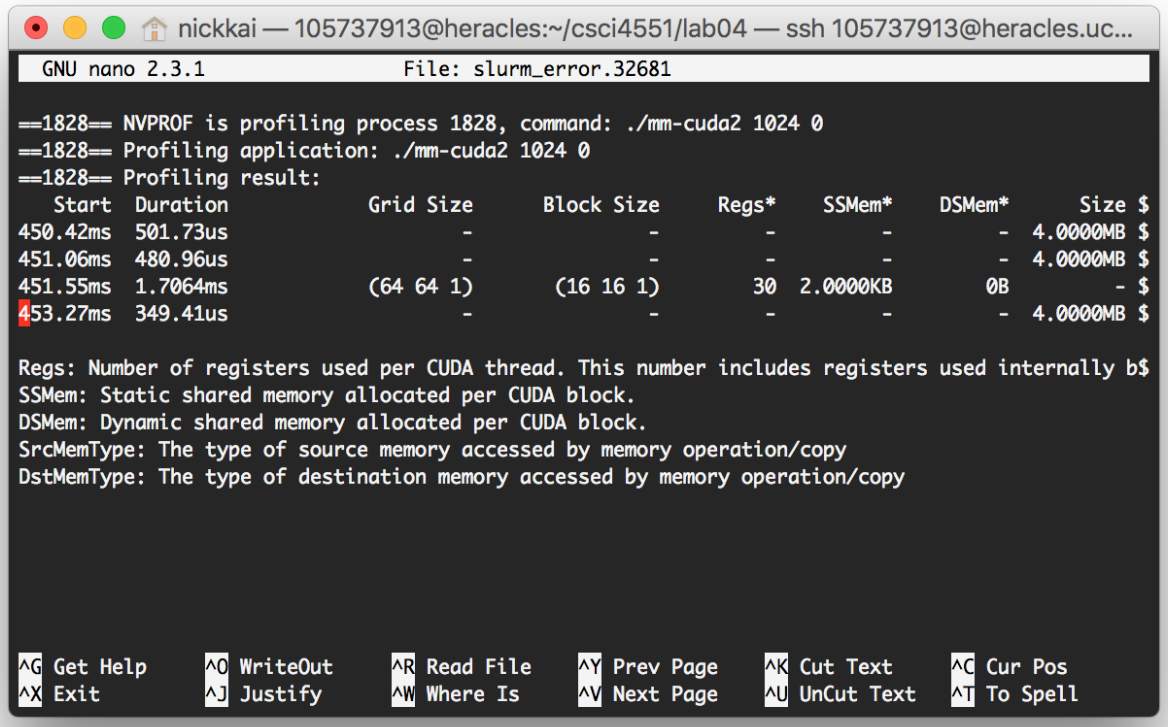
a. **(5 pts)** Describes the main difference between the two given programs.

The main difference between the two given programs whether the program utilizes shared memory or not. The program mm-cuda1 computes the matrix multiplication with no shared memory while the program mm-cuda2 computes the matrix multiplication with shared memory.

b. **(5 pts)** With array size 1024x1024, how much shared memory (in bytes) is allocated in each block in the **mm-cuda2** version? To earn full credits, you need to show your calculations manually and use Nvidia profiling to verify the results. Attach a screenshot of your profiling results. See Sections 1-3 for compiling/running/profiling instructions.

There are 4 bytes in floats and we have declared two floats which are float a and float b. We define these as: #define TILE 16 __shared__ float A[TILE][TILE]; __shared__ float B[TILE][TILE];

We will have 16 x 16 = 256. We have it both for float A and float B so it would be 256 x 2 = 512. Now since we have the total amount for "A" and "B" which is 512, we will multiply it by 4 bytes to get the total number of bytes allocated in each block. So we will do $(512 * 4) = 2048$ bytes. This shows that there are 2048 bytes allocated in each block.




So the Nvidia profiling shows that there are 2000 bytes of shared memory allocated in each block. My manual calculation shows that there are 2048 bytes allocated for each block and the Nvidia profiling shows that there are 2000 bytes allocated for each block. It is very similar and that there is only a difference of 48 bytes between my calculation and the Nvidia profiling.

2. **(10 pts)** Run the given matrix multiplication programs and complete Table 1 below with runtime and speedup (compared to its sequential time on CPU) below.

**Table 1**

| Matrix size | Runtime (seconds) | | | Speedup | |
|---|---|---|---|---|---|
| | mm-seq | mm-cuda1 | mm-cuda2 | mm-cuda1 | mm-cuda2 |
| 192 | 0.04 | 0.23 | 0.28 | 0.173 | 0.143 |
| 1008 | 3.22 | 0.27 | 0.27 | 11.93 | 11.93 |
| 2016 | 27.54 | 0.48 | 0.38 | 57.38 | 72.47 |
| 4080 | 325.15 | 1.55 | 0.82 | 209.77 | 396.52 |

$$\text{Speedup} = \frac{Time_{Sequential}}{Time_{Parallel}}$$

3. **(20 pts)** Given in this lab is a sequential version of a program (**stats.cpp**) that calculates the minimum, maximum element, and standard variance of an array on CPU. Implement a CUDA version of **stats.cpp** that calculates minimum, maximum element, and standard variance on GPU. Name your file as **stats_cuda.cu** and follow instructions in Section 2 and 3 to compile and run your program. (Hint: you can develop three CUDA kernels to perform each task separately, and re-use any codes from **stats.cpp** if needed.) Report your results in Table 2.

| Table 2 | | | |
|---|---|---|---|
| Array size | Sequential version | CUDA parallel version | Speed up |
| 1024 | **0.123 ms** | **0.360 ms** | **0.341** |
| 4096 | **0.206 ms** | **0.425 ms** | **0.485** |
| 16384 | **0.632 ms** | **0.711 ms** | **0.889** |
| 65536 | **2.249 ms** | **1.974 ms** | **1.139** |
| 262144 | **7.883 ms** | **6.345 ms** | **1.242** |
| 1048576 | **32.103 ms** | **25.281 ms** | **1.270** |

In my code, I noticed that the smaller the array size I had, the worst performance it would have compared to the sequential version. As the array size gets larger, the performance improves and would have faster speeds then the sequential version. When we get to an array size of 65536 and larger, CUDA version will have faster speeds.

4. **(extra 5 pts)** Implement another CUDA version to further speed up the **mm-cuda2** program. Name your program as **mm-cuda3.cpp** and follow instructions in Section 2 and 3 to compile and run your program. Let yourself a chance to learn other advanced optimization techniques and apply these techniques to speed up the given program (mm-cuda2) in this lab. Some of useful materials are listed below in Section 4.

a. **(2 pts)** Provide a proof of correctness of your program by attaching here a screenshot of your results with small matrix sizes. Explain the optimization methods you use.

**b.** **(3 pts)** Report the runtime and speed up (compared to its sequential time on CPU) in Table 3. To gain credits, your program must gain better speedup than **mm-cuda2** and explain the optimization techniques you use.

| **Table 3:** | | | |
|---|---|---|---|
| **Matrix size** | **Runtime (seconds)** | | **Speedup** |
| | **mm-seq** | **mm-cuda3** | **mm-cuda3** |
| 192 | | | |
| 1008 | | | |

| | | | |
|---|---|---|---|
| 2016 | | | |
| 4080 | | | |

## SECTION 1: SOURCE CODES

- **mm-cuda1.cu** *(Matrix Multiplication – CUDA Version 1)*
- **mm-cuda2.cu** *(Matrix Multiplication –CUDA Version 2)*
- **stats.cpp** *(Standard Variance – C++ Version)*

## SECTION 2: COMPILING

Please follow instructions in Lab 1 and make sure the source codes are copied to your home directory
Compiling the given codes using following commands:

```
$ ssh node18 nvcc -arch=sm_30 /mypath/mm-cuda1.cu -o /mypath/mm-cuda1
$ ssh node18 nvcc -arch=sm_30 /mypath/mm-cuda1.cu -o /mypath/mm-cuda2
$ g++ -O stats.cpp -o stats
$ g++ -O mm-seq.cpp -o mm-seq
```

Compiler your stats-cuda.cu

```
$ ssh node18 nvcc -arch=sm_30 /mypath/stats-cuda.cu -o /mypath/stats-cuda
```

For more information, please read:

- **Compiling sequential programs on Heracles (C, C++ and Fortran)**
  http://pds.ucdenver.edu/webclass/Compiling%20C_C++%20and%20Fortran%20programs.html
- **Compiling CUDA programs on Heracles**
  http://pds.ucdenver.edu/webclass/Heracles-Compiling%20Cuda%20code.html

## SECTION 3: RUNNING PROGRAMS ON HERACLES

Use the following Slurm scripts to run each code

- **mm-cuda1_slurm.sh** (script to **mm-cuda1** CUDA code on GPU)
- **mm-cuda2_slurm.sh** (script to **mm-cuda2 CUDA** code on GPU)
- **mm-cuda3-cuda_slurm.sh** (script to run your code (**mm-cuda3.cu)** on GPU)
- **stats_slurm.sh** (script to run **stats.cpp on CPU**)
- **mm-seq_slurm.sh** (script to **mm-seq.cpp on CPU**)
- **stats-cuda_slurm.sh** (script to run your code (**stats-cuda.cu)** on GPU)

For matrix multiplication program, from master node, execute the following command:

```
$  sbatch scriptName argument1 argument2
```

where,
argument1 is the dimension of the matrix
argument2: is the option to specify if input/output matrices are printed or not (1=print; 0= not print).
Program only prints if the matrix size is less than 10.

For standard variance program, from master node, execute the following command:
$ sbatch scriptName argument1 argument2 argument3
where,
argument1 : size of array - This is the size of the array to be generated and processed
argument2 : random seed - This integer will be used to seed the random number generator that will generate the contents of the array
argument3 : is the option to specify if input/output matrices are printed or not (1=print; 0= not print). Program only prints if the matrix size is less than 10.

To profile your nvidia code (e.g. mm-cuda2, you need to uncomment this line (line 22): /usr/local/cuda/bin/nvprof ./mm-cuda2 "$@" and comment out (line 23)./mm-cuda2 "$@" inside CUDA Slurm script. The slurm script at lines 22 and 23 should look as follows:

```
/usr/local/cuda/bin/nvprof ./mm-cuda2 "$@"       ## use this command for profiling
## ./mm-cuda2 "$@"
```

Switch back if you no longer want to use Nvidia profiling. You can apply this modification for other CUDA slurm scripts (e.g. **mm-cuda1_slurm.sh** and **mm-cuda3_slurm.sh)** if you want to profile. The profiling results are sometimes printed out into slurm_error.jobid file.
You can further add nvprof arguments to gather the profiling metrics that you want. To learn more about nvprof argurmenst, run:
$ ssh node18 /usr/local/cuda/bin/nvprof --help

## SECTION 4: USEFUL MATERIALS

- **CUDA Instruction and Examples**
  http://developer.download.nvidia.com/books/cuda-by-example/cuda-by-example-sample.pdf
  http://geco.mines.edu/tesla/cuda_tutorial_mio/
  http://docs.nvidia.com/cuda/#axzz3UHN4KyJS
  https://code.google.com/p/stanford-cs193g-sp2010/wiki/GettingStartedWithCUDA
- **Profile your cuda code by using NVIDIA nvprof**
  i) **nvprof** profiling tool can collect and view profiling data from the command-line.
  https://docs.nvidia.com/cuda/profiler-users-guide/index.html#profiling-modes
  ii) **To see a list of all available events on a particular NVIDIA GPU, type:**
  nvprof --query-events
- **Parallel Reduction**
  http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf
- **Mean**
  - The mean of a set of values is also known as the 'average' of those values.
  http://en.wikipedia.org/wiki/Mean
- **Standard Deviation**
  - Measures the degree of dispersion over a set of values
  http://en.wikipedia.org/wiki/Standard_deviation
  - Here's a good example:
  http://en.wikipedia.org/wiki/Standard_deviation#Basic_examples

```
ssh node18 nvcc -arch=sm_30 /home/105737913/csci4551/lab04/mm-cuda2.cu -o
/home/105737913/csci4551/lab04/mm-cuda2
```