**LAB 4: GPUs and CUDA**

**GOAL**

Research into parallel algorithms often involves the development of multiple algorithms and implementations for solving the same 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.

**SUBMISSION**

· Submit your lab answers/results as Word documents (.doc, .docx, .xls) attachments to Canvas.

· Submit all of your codes (make sure they can be compiled without errors, runnable and produce exact results that you will report for this lab).

· You can zip your submission.

· This lab is worth 10 points

**Part 1. Optimizing Parallel Sum Reduction in CUDA**

In this part, we are going to study basic CUDA programming via two parallel algorithms (using global and shared memory) to calculate the sum of all elements of a one-dimensional array (i.e. given input array A = [1,2,3,4], the output result is sum = 1 + 2+ 3 + 4 = 10.)

Three versions are given in this lab:

1) seq\_sum.cpp: the sequential version written in C++.

2) gpu\_global\_mem.cu: the parallel version was written in C++ and CUDA and using GPU’s global memory

2) gpu\_shared\_mem.cu: the parallel version was written in C++ and CUDA and using GPU’s shared memory

**Read and understand each given version, to get familiar with CUDA, please use the following materials: (that have working links)**

<https://code.google.com/p/stanford-cs193g-sp2010/wiki/GettingStartedWithCUDA>

<http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf>

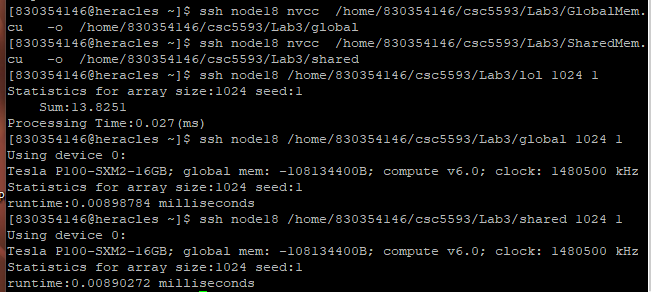
**Complete the following exercises:**

**Exercise 1 (1pt): Compile and run the three different sum reduction versions and collect runtime of each version on Heracles.** To compile and execute CUDA program on Heracles, please refer to Section I (Note that GPUs reside in node 18.) Run each program with the command:

ssh node18 /home/830354146/csc5593/Lab3

C++ syntax compiler

g++ -o lol cou\_seq.cpp



Nvcc -c name.cpp name.cu name2.cu -o executable

Format: executable file <size of array> <random seed>

You can use 1 for random seed

Arguments:

size of array - This is the size of the array to be generated and processed

random seed - This integer will be used to seed the random number

generator that will generate the contents of the array

Fill the runtime for each array size in Table I below, calculate speed up of parallel version in Table 2, and plot the graph of runtime (y-axis) vs array size (x-axis). **For simplicity, array sizes are assumed to be power of 2.**

**Table 1. Runtime**

| Array size | cpu\_seq | gpu\_global\_mem | gpu\_shared\_mem |
| --- | --- | --- | --- |
| 1024 | 0.038 ms | 0.00898754 ms | 0.00890272 ms |
| 4096 | 0.128 ms | 0.0116742 ms | 0.0120173 ms |
| 16384 | 0.511 ms | 0.0092672 ms | 0.00907648 ms |
| 65536 | 2.053 ms | 0.0116646 ms | 0.0087952 ms |
| 262144 | 8.398 ms | 0.0195008 ms | 0.0110397 ms |
| 1048576 | 28.488 ms | 0.0564179 ms | 0.0110259 ms |

**Table 2. Speedup**

| Array size | gpu\_global\_mem | gpu\_shared\_mem |
| --- | --- | --- |
| 1024 | 4.228075758 | 4.268358434 |
| 4096 | 10.96434873 | 10.65131103 |
| 16384 | 55.14071133 | 56.29935834 |
| 65536 | 176.0026062 | 233.4227761 |
| 262144 | 430.648999 | 760.7090772 |
| 1048576 | 504.9461253 | 2583.734661 |

**Plot your graph here:**

| **Chart Chart** |
| --- |

**Exercise 2 (5 pts): Answer the following questions.**

**a) Describes the main difference between the three given programs. Which one achieves the best runtime with which array size and why?**

| Array size | CPU Time | GPU Global Memory | GPU Shared Memory | Fastest Time |
| --- | --- | --- | --- | --- |
| 1024 | 0.038 | 0.00898754 | 0.00890272 | 0.00890272 |
| 4096 | 0.128 | 0.0116742 | 0.0120173 | 0.0116742 |
| 16384 | 0.511 | 0.0092672 | 0.00907648 | 0.00907648 |
| 65536 | 2.053 | 0.0116646 | 0.0087952 | 0.0087952 |
| 262144 | 8.398 | 0.0195008 | 0.0110397 | 0.0110397 |
| 1048576 | 28.488 | 0.0564179 | 0.0110259 | 0.0110259 |

For the most part, fastest runtime goes to GPU Shared Memory.

Looking at the global and shared memory codes, the shared global/shared memory code is a little different. The shared data needs to load the shared data from global memory and make sure that the entire block is loaded. Shared memory also needs to set the number of blocks and threads before running the kernel and global sets it with every run. I would assume that this would be the cause in a lightly quicker runtime since it only sets it once vs setting every time.

The difference is also in the kernel loops, where the shared\_memory\_sum multiplies num\_threads \*sizeof(float)

b) Try to print thread id in *gpu\_shared\_mem.cu \_sum* version using *printf* command. Does the thread id appear in order? why or why not? (To observe the results, you can use small array size (e.g. 4) and a small number of threads (e.g. 4) and set #NUM\_TRIALS = 1 inside the code.)

They do appear to be in order to a certain extent. It will run in order, like from 367-383 and then jump up to thread 864. This is probably due to it being multithreaded, while one thread is busy, the next one will run.

c) With array size 1048576, how much memory (in bytes) is allocated in each block in the *shared\_sum* version? Show your calculations.

Malloc shows = 0x1cb1cb0

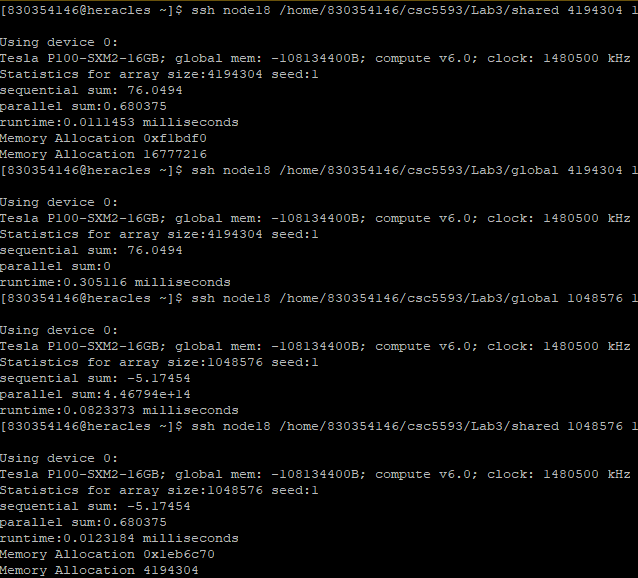
array\_size\*sizeof(float) = 4194304 MB

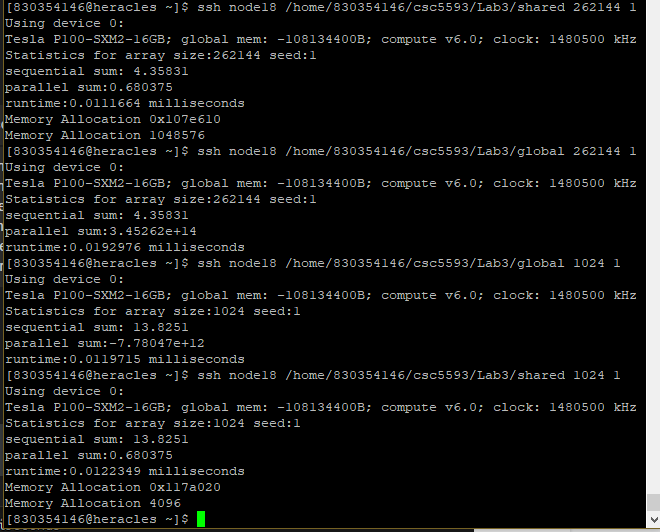
d) Do the *gpu\_global\_mem.cu and gpu\_shared\_mem.cu* produce correct results with array sizes larger than 1048576? Justify your answers by observing results of these versions. Attach here screenshots of your results.

1048576 = 1024^2

4194304 = 2048^2

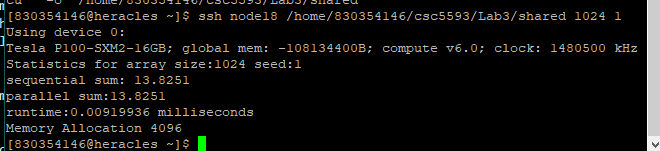
16777216 = 4096^2





No, arrays larger and smaller than 1048576 do not produce the same parallel sum. Sequential sum is consistent across the tested memory sizes

e) Modify *gpu\_shared\_mem.cu* to deal with the issue in d). Submit your modified version on Canvas. Attach here screenshots of your results.



Parallel and Sequential sum add up now. Added the two int variables, int num\_threads and int num\_blocks into the Kernel 1 loop.

**Part 2. (4 pts) Implement a sequential and a parallel CUDA program to calculate the minimum, maximum element and standard variation of an array**.

· (1 pt) Verify the correctness of each of your program on small array size.

· (1 pt) Report your results in Table 3. The array sizes are assumed to be power of 2.

· (2 pts) Achieve better runtime for CUDA version compared to sequential version.

**Table 3. Runtime**

| Array size | Sequential version | CUDA parallel version | Speed up |
| --- | --- | --- | --- |
| 1024 |  |  |  |
| 4096 |  |  |  |
| 16384 |  |  |  |
| 65536 |  |  |  |
| 262144 |  |  |  |
| 1048576 |  |  |  |

**Learn about standard variation here:**

· 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>

**SECTION I: COMPILING AND RUNNING PROGRAMS ON HERACLES**

· Hardware Information about Heracles:

<http://pds.ucdenver.edu/webclass/Heracles_Architecture.html>

· Compiling CUDA programs on Heracles

<http://pds.ucdenver.edu/webclass/Heracles-Compiling%20Cuda%20code.html>

· Running program on Heracles

<http://pds.ucdenver.edu/webclass/Heracles-RunningPrograms.html>

CUDA programs are only executable on node 18. You can ssh to node 18 first, and then execute your CUDA program in this node.