Crystin VanWagenen

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.

In this lab, you will build two programs, one sequential version for the CPU and one parallel version for the GPU. Afterwards, you will gather data and answer questions about your implementations.

SUBMISSION

- Rename this doc **to** **CSC5593_Lab4_yourlastFirstname.docx** (e.g. CSC5593_Lab2_LinckIris.docx).
- This is not a group assignment.
- **Provide answers for requirements 1, 2, 3 and 4.**
- For requirements #1 and #2, submission will include your modified versions of **stats_s.c** and **stats_gpu.cu**. Note that these files must compile on Hydra or Heracles using the instructions provided.
- Each program will be tested on Hydra or Heracles by the grader, you should say in which machine you used as well all the steps for compiling and run your code.
- For requirements #3 and #4, submission will include a document with your results/answers.
- Email your lab answers/results as Word documents (.doc, .docx, .xls) attachments. If the lab assignment requires program implementation, send the code (program and input) files as attachment as well as the commands for compiling and running them. Send an email to Iris and Manh at pdslab@ucdenver.edu **with subject "CSC5593 Lab 4"**;
- Send your questions to Iris and Manh at pdslab@ucdenver.edu **with subject "CSC5593 Questions Lab 4"**;
- Comments and grades will be added to your documents **so do not submit your homework as a PDF.**

ADVICE

- Develop your sequential version immediately. You shouldn't need much more than 20 lines of code to complete it.
- After your sequential version is complete, iteratively develop your CUDA version as you learn. Try to solve little problems first before trying to tackle your entire solution.
- Some of the questions in requirement #4 might help you design your code for requirement #2. You can answer some of them before completing #1 and #2.
- Do not procrastinate on this lab.

NOTES

- **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

- **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

REQUIREMENTS

1. Starting with **stats_s.c**, build a sequential program that calculates the minimum, maximum, mean, and standard deviation of an array of doubles. The structure of this code has already been written so you can focus on your algorithm. You should only need to place your code between the tags, “CALCULATE VALUES FOR MIN, MAX, MEAN, and STDDEV”.

   You should be able to compile this version using the following compilation instruction:

   ```
   gcc stats_s.c -o stats_s -lm
   ```

   Your program **must** compile and run on Hydra or Heracles. Identify which cluster you used.

   Here are some sample executions. You can use these as a basis of comparison for your solution. Given the use of each seed you see below (the seed is the second argument, I've colored them blue), your minimum and maximum values should match exactly for the given seed. Your mean and standard deviation values should be close to the actual – say within 0.01%. I've omitted the runtimes.

   ```
   [hydra ~]$ bpsh 5 ./stats_s
   Format: stats_s <size of array> <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
                     to be processed
   [hydra ~]$ bpsh 5 ./stats_s 100 7983

   Statistics for array ( 100, 7983 ):
       Minimum = 1110.371668, Maximum = 992249.621540
       Mean = 526604.645223, Standard Deviation = 313024.981570
   ```

```
 [hydra ~]$ bpsh 5 ./stats_s 1000 2854

 Statistics for array ( 1000, 2854 ):
    Minimum = 400.610734, Maximum = 999633.841682
    Mean = 511140.894837, Standard Deviation = 283314.411939

 [hydra ~]$ bpsh 5 ./stats_s 20000 5701

Statistics for array ( 20000, 5701 ):
    Minimum = 52.004121, Maximum = 999982.517213
    Mean = 499311.049550, Standard Deviation = 287313.034641

 [hydra ~]$ bpsh 5 ./stats_s 3000000 11056

Statistics for array ( 3000000, 11056 ):
    Minimum = 0.168104, Maximum = 999999.904074
    Mean = 499816.343646, Standard Deviation = 288702.699507
```

```
Statistics for array ( 100, 7983 ):
    Minimum = 1110.371668, Maximum = 992249.621540
    Mean = 526604.645223, Standard Deviation = 311455.924163
Processing Time: 0.1870 milliseconds
[crystinrodrick7@hydra csc5593]$ bpsh 12 ./stats_s 1000 2854
Statistics for array ( 1000, 2854 ):
    Minimum = 400.610734, Maximum = 999633.841682
    Mean = 511140.894837, Standard Deviation = 283172.719301
Processing Time: 0.2680 milliseconds
[crystinrodrick7@hydra csc5593]$ bpsh 5 ./stats_s 1000 2854
Statistics for array ( 1000, 2854 ):
    Minimum = 400.610734, Maximum = 999633.841682
    Mean = 511140.894837, Standard Deviation = 283172.719301
Processing Time: 0.2560 milliseconds
[crystinrodrick7@hydra csc5593]$ bpsh 5 ./stats_s 100 7983
Statistics for array ( 100, 7983 ):
    Minimum = 1110.371668, Maximum = 992249.621540
    Mean = 526604.645223, Standard Deviation = 311455.924163
Processing Time: 0.1630 milliseconds
[crystinrodrick7@hydra csc5593]$ bpsh 5 ./stats_s 20000 5701
Statistics for array ( 20000, 5701 ):
    Minimum = 52.004121, Maximum = 999982.517213
    Mean = 499311.049550, Standard Deviation = 287305.851725
Processing Time: 2.7840 milliseconds
[crystinrodrick7@hydra csc5593]$ bpsh 5 ./stats_s 3000000 11056
Statistics for array ( 3000000, 11056 ):
    Minimum = 0.168104, Maximum = 999999.904074
    Mean = 499816.343646, Standard Deviation = 288702.651390
Processing Time: 146.8700 milliseconds
```

2. Starting with **stats_gpu.cu**, build a GPU CUDA program that calculates the minimum, maximum, mean, and standard deviation of an array of doubles. The structure of this code has already been written so you can focus on your algorithm. For the main function, place your code only between the tags, “CALCULATE VALUES FOR MIN, MAX, MEAN, and STDDEV”. Place your GPU kernel functions between the tags, “PLACE GPU KERNELS HERE”. The helpers HANDLE_ERROR and checkCUDAError are meant to help you check errors returned

from the GPU. Use the HANDLE_ERROR macro when doing any GPU operation that is not a kernel call. Use the checkCUDAError function to evaluate errors after a kernel call. Note that your GPU version should outperform the sequential version when array sizes are large enough.

Here is an example of using HANDLE_ERROR:

```
HANDLE_ERROR(
       cudaMalloc( (void**) &dev_block_sums, num_blocks * sizeof( double ) )
  );
```

Here is an example of using checkCUDAError:

```
kernel_calculate_min_max_sum<<<num_blocks,num_threads>>>( dev_array,
                                                           array_size,
                                                           dev_block_mins,
                                                           dev_block_maxs,
                                                           dev_block_sums );
checkCUDAError( "kernel_calculate_min_max_sum", true );
```

Your implementation should use parallel reduction on the GPU in order to determine the minimum, maximum, sum (needed for the mean) and standard deviation sum (needed for standard deviation) values. All reduction to obtain minimum, maximum, sum, and standard deviation sum must be done on the GPU. You can transform the sum into the mean and the standard deviation sum into the standard deviation on the CPU. See the **Notes** above for help with parallel reduction.

Hint: Be sure to use shared memory for the reduction. Be aware that doing a parallel reduction over an array whose size is not a power of two requires some thought.

I've added a helper GPU device function: __device__ double device_pow(double x, double y). Use this function, device_pow(), in place of pow() in your CUDA kernel code. You will need it when you calculate the standard deviation sum. It is defined in order to circumvent function override problems.

Try to limit your kernel launches. However, if you find that you exceed 4 separate kernel launches, then you are probably not thinking of the problem properly. I needed to define 3 kernel functions and I used 4 kernel launches. You could use less, but dimensioning would be tricky.

You should be able to compile this version using the following compilation instruction:

```
nvcc -arch=sm_21 stats_gpu.cu -o stats_gpu
```

Your program **must** compile and run on Hydra or Heracles.

Here are some sample executions. You can use these as a basis of comparison for your solution. Given the use of each seed you see below (the seed is the second argument, I've colored them blue), your minimum and maximum values should match exactly for the given seed. Your mean and standard deviation values should be close to the actual - say within 0.01%. I've omitted the runtimes.

```
[hydra ~]$ bpsh 12 ./stats_gpu
Format: stats_gpu <size of array> <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
                  to be processed
[hydra ~]$ bpsh 12 ./stats_gpu 100 7983
Statistics for array ( 100, 7983 ):
    Minimum = 1110.371668, Maximum = 992249.621540
    Mean = 526604.645223, Standard Deviation = 311455.924163
[hydra ~]$ bpsh 12 ./stats_gpu 1000 2854
Statistics for array ( 1000, 2854 ):
    Minimum = 400.610734, Maximum = 999633.841682
    Mean = 511140.894837, Standard Deviation = 283172.719301
[hydra ~]$ bpsh 12 ./stats_gpu 20000 5701
Statistics for array ( 20000, 5701 ):
    Minimum = 52.004121, Maximum = 999982.517213
    Mean = 499311.049550, Standard Deviation = 287305.851725
[hydra ~]$ bpsh 12 ./stats_gpu 3000000 11056
Statistics for array ( 3000000, 11056 ):
    Minimum = 0.168104, Maximum = 999999.904074
    Mean = 499816.343646, Standard Deviation = 288702.651390
```

3. Using the seed value of 343, execute the sequential (from #1) and GPU CUDA (from #2) versions of your program for the following array sizes: 100, 1000, 10000, 100000, 1000000, 10000000, 10000100. Provide a **table** and a **graph** showing the runtime results.

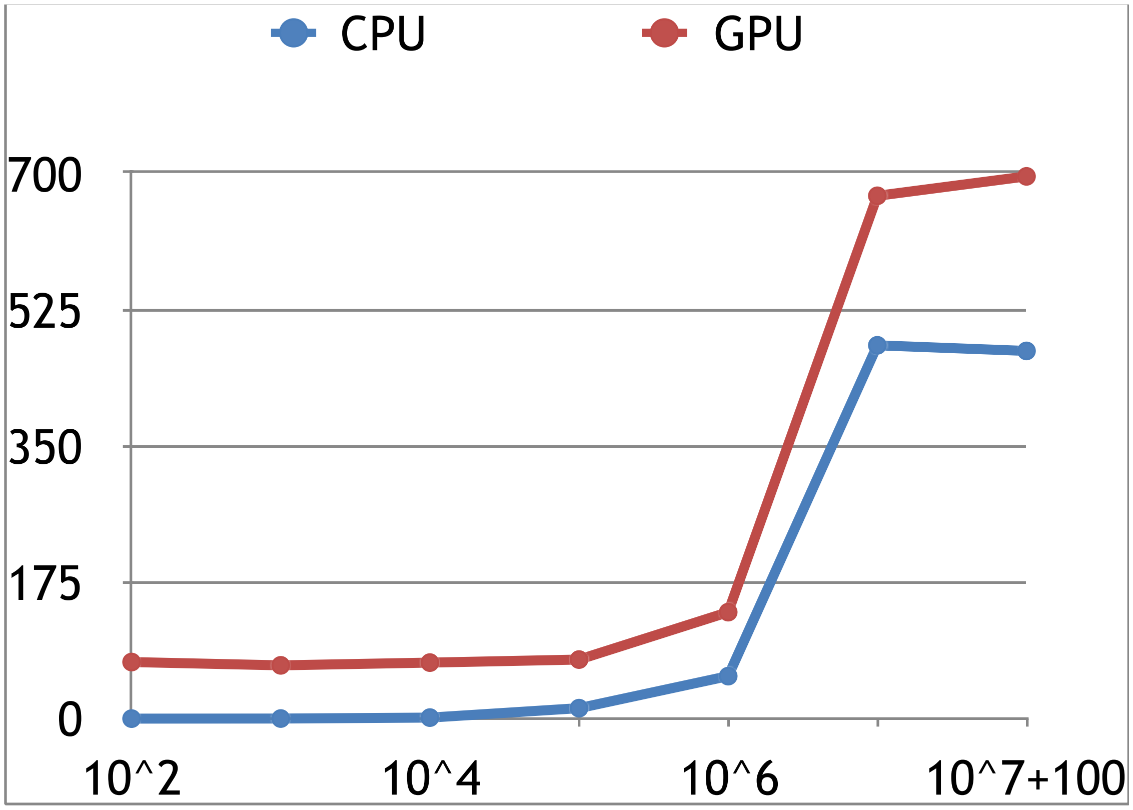
```
MINMAX Statistics for array ( 100, 343 ):
MIN = 13863.650623
MAX = 984520.023216
RUNTIME = 108.4250 milliseconds
MEAN = 509951.130697

MAX Statistics for array ( 100000, 343 ):
MIN = 1708.979719
MAX = 999288.109131
RUNTIME = 75.8940 milliseconds
MEAN = 2639.243915
```

a. Table with runtime results

Size	Runtime (Sequential)	Runtime (GPU CUDA)
100 (10^2)	0.148	72.714996
1000 (10^3)	0.259	68.488998
10000 (10^4)	1.46	72.080002
10000 (10^5)	13.562	75.8940
100000 (10^6)	54.6740	136.748993
1000000 (10^7)	478.92	670.669006

10000100 (10^7 + 100)	471.718	695.6440

b. Plot your graph here



4. **Answer the following questions:**

   a. Describe how you implemented parallel reduction. How are you handling the reduction of arrays whose size is not a power of 2?

I implemented the parallel reduction by addressing the global index and the local index, to help split off parallel processes and then sync the threads together at the end. It is handled the same way whether or not the array is of power 2.

   b. Compare the performance of the sequential and GPU versions when the array size is 1000. Which performs better? Why?

The runtime for the sequential performs better. This may be in part due to the size not being a power of two.

c. How did you dimension (i.e. how many blocks and threads are used) your GPU computation? Explain. How many blocks and threads are used when array_size = 100? When array_size = 10,000? When array_size = 40,000,000?

The number of threads per block are 512, independent of the array_size. The number of blocks depended on the array_size. It would be (array_size + 511)/512. So for array_size 100, there would be 1.193359375 blocks. For 10000, 20.529296875 blocks. For 40,000,000, there would be 78125.998046875.

d. Given that the memory bandwidth of the GPU is 144 Gb/s and the PCIe bus bandwidth is 2.25 Gb/s, how many milliseconds should it take to transfer an array with 100,000,000 doubles to the GPU? Show your calculation. How could you measure this performance?

e. Given that the size of the GPU global memory is 3Gb, what is the maximum array size that you can process? Show your calculation. If you had to process a longer array, how could you change your program to accomplish this?

The maximum array_size would be 54772, by taking the square root of 3Gb. In order to process a longer array, you could split the program into two chunks, each finding the solution for the problem, and storing the result and then combining to find the overall result.

f. Parallel reduction almost invariable requires the use of GPU shared memory. Look through your GPU code, find the kernel function that uses the most shared memory. For which kernel function are you using the most shared memory? How much shared memory, in kB, does this kernel function use? Given that the use of shared memory displaces L1 cache and that they share 48 kB, how much L1 cache is available for this kernel function? Does this concern you? Why or why not?

*The sum kernel function is using the most shared memory because it is storing every value in the array. Therefore the amount of shared memory is dependent on the array_size. So for each array_size, we have array_size * sizeof(double).This means not a lot of cache is available for there kernel function. This shouldn't matter too much because of compiler optimization techniques that are out there.*

SOURCE CODE

The source code file **stats_s.c** is the framework for the sequential implementation of the minimum, maximum, mean, and standard deviation. Modify this file to create your sequential program.

The source code file **stats_gpu.cu** is the framework for the GPU CUDA implementation of the minimum, maximum, mean, and standard deviation. Modify this file to create your GPU CUDA program.

COMPILING AND RUNNING THE PROGRAMS

Step 1: Copy source code files to Hydra

- Logon the cluster (see Lab 1)
- Make a folder named csc5593 in your home directory using the **mkdir** command.

```
mkdir /path/to/directory
```

For example:

```
mkdir /home/john/csc5593
```

- Copy the source code files to csc5593. If you are using MAC or Linux, you can copy these files using **scp** or **sftp** command. If you are using Windows, you can use Bitvise, WinSCP, or SSH Secure Shell to login and copy files from your PC to the cluster.
- Change the working directory to csc5593 using **cd** command

```
cd /path/to/directory
```

For example:

```
cd /home/john/csc5593
```

Step 2: Compile source code files

Compiling the C/C++ , opemMP and cuda Programs on Hydra

http://pds.ucdenver.edu/webclass/Compiling%20C_C++%20and%20Fortran%20programs.html

http://pds.ucdenver.edu/webclass/Compiling%20openMP%20programs.html

http://pds.ucdenver.edu/webclass/Compiling%20Cuda%20code.html

Step 3: Run the program.

The first step when running your program on a compute node is to find an open node to run on. See the page on Monitoring the Hydra Cluster for how to monitor the compute nodes on the cluster. Try to select a node that shows the lowest usage (0% if possible).

In order to check node status on Hydra, visit:

http://pds.ucdenver.edu/webclass/Monitoring%20the%20Cluster.html

Afterwards, you can choose to execute your program immediately or with the batching system.

To run your program immediately, visit: http://pds.ucdenver.edu/webclass/Running%20programs%20on%20Hydra.html

To run your program via scheduling job on Hydra,

visit: http://pds.ucdenver.edu/webclass/at%20-%20Command%20for%20scheduling%20programs.html

The advantage of using **at (command for scheduling job)** is that you can log off and walk away from your computer while your results are generating. You cannot log off or end your terminal session when you execute immediately. Remember to check node status before executing your program.

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

The first step when running your program on a compute node on Heracles is to find an open node to run on. Visit https://heracles.ucdenver.pvt/mcms/ for monitoring the compute nodes on Heracles. Try to select a node that shows the lowest usage.

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

Scheduling jobs on Heracles by using SLURM

http://pds.ucdenver.edu/webclass/Heracles-RunningPrograms%20Slurm.html