**ECE569 - Spring 2022**

**Homework 4 – Histogram Generation**

1. **Objective**

The purpose of this lab is to implement “three” versions of the histogram generation algorithm for an input array of integers within a given range. Each integer will map into a single bin, so the values will range from 0 to (NUM\_BINS - 1). The histogram bins will use unsigned 32-bit counters that must be saturated at 127, meaning all bins with values larger than 127 need to be clipped to 127. This clipping operation is a key step during histogram equalization process. The input length can be assumed to be less than 2^32. NUM\_BINS is fixed at 4096 for this homework.

In your implementation, execution will be split into two phases (two kernel launches). First kernel runs the histogram without saturation, and the second kernel applies clipping.

**Logistics**

In the assignment package you will find the “hw4” folder along with the “CmakeLists.txt”. Place them into your “labs” folder.

In the assignment package you will find “run\_hw4.slurm” file. Copy it to your “build-dir” folder.

1. **Task 1: Instructions for histogram generation**

You will complete three versions of the histogram.

* Version 0: global memory only with coalesced memory access using interleaved partitioning (striding)
* Version 1: shared memory with privatization
* Version 2: shared memory with your choice of optimization approach

You will find the template source code named “kernel.cu” in hw4/Histogram. In the same folder you will find the “solution.cu”. The “solution.cu” file includes the main function and all timing measurements for each kernel call. Do not modify the “solution.cu”. **For versions 0 and 1, only add your code into kernel.cu file for each kernel as specified below and do not modify any other part of the code.** Edit only “kernel.cu” file to implement the following two kernels:

// version 0

// global memory only interleaved version

// include comments describing your approach

\_\_global\_\_ void histogram\_global\_kernel(unsigned int \*input, unsigned int \*bins,

unsigned int num\_elements,

unsigned int num\_bins) {

// insert your code here

}

// version 1

// shared memory privatized version

// include comments describing your approach

\_\_global\_\_ void histogram\_shared\_kernel(unsigned int \*input, unsigned int \*bins,

unsigned int num\_elements,

unsigned int num\_bins) {

// insert your code here

}

You also need to implement the clipping function (line 38 in kernel.cu)

// clipping function

// resets bins that have value larger than 127 to 127.

// that is if bin[i]>127 then bin[i]=127

\_\_global\_\_ void convert\_kernel(unsigned int \*bins, unsigned int num\_bins) {

// insert your code here

}

**For version2 add your code into kernel.cu file as specified below. You are allowed to modify the provided “solution.cu” in case your implementation approach involves pre/post-processing on the host or adding other kernels.**

// version 2

// your method of optimization using shared memory

// include DETAILED comments describing your approach

\_\_global\_\_ void histogram\_shared\_optimized(unsigned int \*input, unsigned int \*bins,

unsigned int num\_elements,

unsigned int num\_bins) {

// insert your code here

}}

**Version 2: Options for your method of optimization using shared memory:**

For this you will need to conduct a survey on other techniques. A few keywords for you as a starting point are listed here:

* Sorting based approach
* Reduce by key
* Compression before reduction

**Compiling and Execution Instructions**

You will need to compile your code from the “build-dir” folder similar to the process described in earlier assignments. Change your home directory to ece569/build\_dir

**$ cd ece569/build\_dir**

Then run the following commands:

**$module load cuda11/11.0**

**$CC=gcc cmake3 ../labs**

**$make**

The makefile compiles and generates the executable “Histogram\_Solution” in the “build-dir” directory. The executable generated as a result of compiling the code can be run using the following command:

./ Histogram\_Solution -e <expected.raw> \  
 -i <input.raw> <version\_number> -t integral\_vector

where <expected.raw> is the expected output, <input.raw> is the input dataset, **<version\_number>** is the histogram version to execute where 0 is global, 1 is shared privatized, 2 is shared with your choice of optimized version. The datasets can be generated using the dataset generator built as part of the compilation process.

Histogram\_DatasetGenerator in the build-dir generates the data set.

$./ Histogram\_DatasetGenerator

then go into the “Histogram” folder to access the test files.

Example test run:

Refer to the run\_hw4.slurm for running all test cases.

* You need to create the “Histogram\_output” folder in your “build\_dir” directory. The script will search for the “Histogram\_output” folder to write outputs for all the test cases.
* You need to update the paths in the slurm script.
* Script is configured to run one kernel version at a time as a command line parameter.

In each output file you will find the timing and correctness information for your kernel.

**Make sure that you are still in the “build-dir” directory, then type the following commands to submit your jobs.**

$ srun run\_hw4.slurm

Note that the slurm script is configured to test version 0 (line 53). You need to change the version number by replacing command line parameter of “0” with 1 or 2.

1. **Analysis**

**Experiment1: Run “data set 6” for 10 times and calculate the average execution time for each version of the kernel. We are interested in time spent on the kernel excluding data transfer overhead. main() function has the timing code in place already and you will observe the collected time in the output file for each test case. Indicate the total execution time for each run.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Execution Time based on data set 6 (500,000 elements, random, 4096 bins, clipped at 127** | | | |
| **Time** | **Version0- Global Memory-Interleaved** | **Version1- Shared-Memory-Privatized** | **Version2-Shared-Memory-Optimized** |
| **1** |  |  |  |
| **2** |  |  |  |
| **3** |  |  |  |
| **4** |  |  |  |
| **5** |  |  |  |
| **6** |  |  |  |
| **7** |  |  |  |
| **8** |  |  |  |
| **9** |  |  |  |
| **10** |  |  |  |
| **Average** |  |  |  |

**Experiment2: Now create a data set of size 500,000 with each element having the same value. Run for 10 times and calculate the average execution time for each version of the kernel. Indicate the total execution time for each.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Execution Time based on data set 6 (500,000 elements same value, 4096 bins, clipped at 127** | | | |
| **Time** | **Version0- Global Memory-Interleaved** | **Version1- Shared-Memory-Privatized** | **Version2-Shared-Memory-Optimized** |
| **1** |  |  |  |
| **2** |  |  |  |
| **3** |  |  |  |
| **4** |  |  |  |
| **5** |  |  |  |
| **6** |  |  |  |
| **7** |  |  |  |
| **8** |  |  |  |
| **9** |  |  |  |
| **10** |  |  |  |
| **Average** |  |  |  |

**Analysis-1: Discuss your findings based on experiments 1 and 2 for Versions 0 and 1, Global vs Shared memory implementation.(1 page limit)**

**Answer:**

**V0 vs V1 on Experiment1, how much and privatization**

**V0 vs V1 on Experiment 2, how much with privatization**

**Experiment1 vs Experiment2 on V0, impact of input data on V0**

**Experiment1 vs Experiment2 on V1, impact of input data on V1**

**Analysis-2: Explain your implementation approach for Version-2. Discuss the superiority of your approach (Version 2) compared to versions 0 and 1. What is the parallelization opportunity that you exploited or what drawbacks of Versions 0 and 1 did you resolve? Does performance vary based on the nature of the distribution of the data? For which test cases does Version-2 perform better than Versions 0 and 1? Discuss any relevant execution time trends. (1.5 page limit)**

**Answer:**

**Short Answer**

**How many global memory reads are being performed by each kernel? Explain.**

**Answer:**

**How many global memory writes are being performed by each kernel? Explain.**

**Answer:**

**How many atomic operations are being performed by each kernel? Explain.**

**Answer:**

1. **Submission Instructions**

**It is critical that you follow the following steps for submission. Please refer to penalty points.**

1. Create a folder and name it as your “net id”.
   1. Include **kernel.cu** and **solution.cu** filesin the **“netid”** folder

**Note that, when testing version 0 and version 1, I will only use your “kernel.cu” submission and compile your kernel.cu with my version of the solution.cu.**

**When testing for kernel version 2, I will use both your submitted kernel.cu and solution.cu files. You can work in group of 2 or 3 members for version 2. In the solution.cu. Please include names of the group members in solution.cu file.**

1. Zip your **netid** folder **(netid.zip)**
2. Create a **hw4\_netid** folder
   1. Include **netid.zip**
   2. Include **ECE569\_hw4.docx** or **ECE\_hw4.pdf** file with part C (analysis)
3. Zip **hw4\_netid** folder
4. Submit **hw4\_netid.zip** to the designated D2L hw4 folde**r.**
5. **Grading Criteria**

Coding Style and Commenting the kernels: 30

Histogram kernel 0: global interleaved - private test cases 35

Histogram kernel 1: shared privatized - private test cases 70

Histogram kernel 2: shared optimized - private test cases 100

Analysis 65

Total 300 points

Penalty Conditions:

* 15% points penalty will be applied if you do not follow the submission instructions.
* 15% points penalty per day for late submission.
* 30% penalty when Version 2 is not faster than Version 0 and Version 1 for private test cases with input sizes >= 100,000.