CS 6023 GPU Programming
Assignment 3

Handout date: 16/03/2019
Due date: 24/03/2019 (end of day)

General instructions:

1. The goal of this assignment is to let you explore ways to improve the running time of a CUDA program.
2. This is an individual assignment. Collaborations and discussions with others are not encouraged. However, if you do collaborate with anyone, you are requested to explicitly state the same in your submitted report.
3. You have to use C with CUDA for your implementation.
4. You have to turn in three components: (a) well-documented code (.cu files for each programming question), (b) output files (.txt for each relevant programming question), and (c) a detailed report (single .pdf file) of the results of the experiment. All these are to be submitted on Moodle. No hard copy is required.
5. Code will be checked for functional correctness, quality of code, and performance. Report will be verified for completeness and to-the-point answers to the subjective questions posed.
6. If you have doubts please raise them in the Moodle Q&A forum, and follow the same regularly.
7. Codes generating incorrect output will be awarded zero marks.
8. We provide you with code snippets for taking inputs. Please include these code snippets as it is in your source code. Modifying the code snippets may result in an incorrect output and lead to consequences mentioned in 7.

Submission instructions:

1. Make a report PDF as *<username>_report.pdf*, where <username> is the username used by you to login to the GPU cluster. For many it may be your roll number.
2. Put all files inside a folder named your username and compress the folder into a tarball in the following format : *username_A03.tar.gz*
3. Upload the tarball in Moodle. For instructions on how to do this contact the TAs.

N-Count-Grams:

An N-Gram is a contiguous sequence of words/items from a particular sample space (most prominently from speech or text data). N-Count-Grams of a text are our own created definition to mean sequences of lengths of contiguous words in a window of size N moving in a sentence. For example, let us consider the sentence *"The quick brown fox jumps over the lazy dog"*. For this, we have the following N-grams:

1-grams: "the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"
2-grams: "the quick", "quick brown", "brown fox", "fox jumps", "jumps over", and so on.
3-grams: "the quick brown", "quick brown fox", "brown fox jumps", "fox jumps over", and so on.
9-gram will be the whole sentence itself.

Corresponding to N-grams we have N-count-grams which provide the number of characters in the words of the N-grams. For example, for the N-grams listed above the corresponding N-count-grams respectively are:

1-count-grams: `{3}, {5}, {5}, {3}, {5}, {4}, {3}, {4}, {3}`
2-count-grams: `{3, 5}, {5, 5}, {5, 3}, {3,5}, {5, 4}`
3-count-grams: `{3, 5, 5}, {5, 5, 3}, {5, 3, 5}, {3, 5, 4}`
9-count gram: `{3, 5, 5, 3, 5, 4, 3, 4, 3}`

So, for a sentence with `M` words, there are a total of `[M - (N - 1)]` N-count-grams, including possible repetitions.

While we are working with N-count-grams, N-grams of texts can be very useful to predict the next word in a sentence by calculating the probabilities of different words that can come after a particular sequence of words. You can learn more about it in the following wiki article: https://en.wikipedia.org/wiki/N-gram

Based on the concept of N-Count-Gram explained above, you need to solve six questions. For each of those questions, you have to take input from a text comprising of many words.

Input/Output Instructions:

The command-line arguments should look like the following:

```
$ ./a.out <M> <N> <filename>
```

Where, <M> = number of words to take input,
<N> = value for N-count-gram,
<filename> = name of the input text file with extension.

Use the following code snippet to take the input in a 1D array:

```
#include <stdio.h>
#define MAXWORDS 5000000

char words[MAXWORDS * 10];    // Word length capped to 10

void readInput(int M, char * filename) {
      FILE * ipf = fopen(filename, "r");
      int totalWordCount = 0;
      while (fscanf(ipf, "%s",  &words[totalWordCount*10]) != EOF
                && totalWordCount < M
                && totalWordCount < MAXWORDS) {
            totalWordCount += 1;
      }
      fclose(ipf);
}

int main(int argc, char * argv[]) {
      int M = atoi(argv[1]);
      int N = atoi(argv[2]);
      char * filename = argv[3];
      readInput(M, filename);
      // do computation
      return 0;
}
```

Output format (say, for N = 2) in `<username>_out_2.txt` in lexicographic order:

```
<word1_len_1> <word2_len_1> <count_1>
<word1_len_2> <word2_len_2> <count_2>
...
<word1_len_i> <word2_len_i> <count_i>
```

Questions:

1. **(0.3 + 0.2 = 0.5 marks)** In order to get the results for N-count-grams, we need to get the lengths of all the words in the input text. Write a CUDA kernel to calculate the lengths of all the words in parallel. You should send all the text data to the GPU and perform length calculation there only (i.e., no computation on the host, except taking inputs).

   Note: The executable for this question does not need the value N, but still is taken as input to keep everything consistent.

   Submit the source code as `<username>_1.cu`

Generate output as `<username>_out_1.txt`, containing the same text represented in lengths (one in each line).
Report your configuration and execution time of the kernel in `<username>_report.pdf`

2. **(1 + 0.5 = 1.5 Marks)** Write a CUDA kernel to calculate the N-count-gram data for all the text in the input. You should use GPU global memory to store all the histogram bins and update the values atomically for result integrity.

   Submit the source code as `<username>_2.cu`
   Generate output as `<username>_out_2.txt`, according to the format described in the 'Input/Output Instructions' section.
   Report your configuration and kernel execution time for `N = {1, 2, 3, 4, 5}` in `<username>_report.pdf`

3. **(1.5 + 0.5 = 2 Marks)** Accessing the global memory for each update, along with atomic instructions, will significantly reduce the performance of the program. So, an alternative solution is to use shared memory for the repeated updates. When all the shared-memory copies are updated, the final global-memory copy of the bins is updated atomically by a representative thread inside each block.

   Write a CUDA kernel to perform the histogram computation using shared memory. Since shared memory is limited, test your kernel for N = {1, 2, 3} only.

   Submit the source code as `<username>_3.cu`
   Generate output as `<username>_out_3.txt`, according to the format described in the 'Input/Output Instructions' section.
   Report your configuration and kernel execution time for `N = {1, 2, 3}` in `<username>_report.pdf`

4. **(2 + 1 = 3 Marks)** Analysis of the histogram bins shows that not all bins are equally valued. This means that some bins become hotspots and some bins are rarely touched. For large values of N, not all bins can be stored into the shared memory. Naturally, an optimization can be done for our problem where the most accessed bins are stored in the fast shared memory, while the rest of the bins can be updated directly to the main memory.

   Write a CUDA kernel that uses shared memory to cache the bins that are frequently accessed, and works for N = {4, 5}.

   Note: The choice of the bins (i.e., which ones to keep in the shared memory) can be static or dynamic or based on heuristics.

   Submit the source code as `<username>_4.cu`
   Generate output as `<username>_out_4.txt`, according to the format described in the 'Input/Output Instructions' section.
   Report your configuration and kernel execution time for `N = {4, 5}` in `<username>_report.pdf`

5. **(2 + 1 = 3 Marks)** The idea of privatization can be applied to much smaller granularities. So, instead of sharing some bins with all the threads of a block, we can introduce smaller groups having local copies of the bins. After the computations are done, the final result from each group can be updated into the global copy.

   Write a CUDA kernel that computes the histogram values with 8 threads per group, each group updating the local copies of the bins.

   Submit the source code as `<username>_5.cu`
   Generate output as `<username>_out_5.txt`, according to the format described in the 'Input/Output Instructions' section.
   Report your configuration and kernel execution time for `N = {1, 2, 3, 4, 5}` in `<username>_report.pdf`

6. **(Bonus: 0.5 + 0.5 = 1 Mark)** The idea of Q5 can be extended to the lowest level of granularity, i.e., each thread maintains its own copy of the histogram bins. This means that the threads no longer need to use atomic updates when computing the results.

   Write a CUDA kernel implementing the aforementioned idea.

   Submit the source code as `<username>_6.cu`
   Generate output as `<username>_out_6.txt`, according to the format described in the 'Input/Output Instructions' section.
   Report your configuration and kernel execution time for `N = {1, 2, 3, 4}` in `<username>_report.pdf`

As an example for the sentence *"The quick brown fox jumps over the lazy dog",* and for N = 2 the output of the file could have the following entries (among others):

3 5 2
5 3 1
5 4 1
5 5 1

Note from the example above that the N-count-gram `{3, 5}` is different from `{5, 3}`, i.e., the ordering of the numbers matters in defining distinct N-grams.

Note 1: Each question has two components for marking - first for correctness and second for performance above a certain threshold. Marks for the second part will be awarded only if it passes the correctness test.
Note 2: You should print the kernel execution times to stdout. Correctness will be checked only on the output files.
Note 3: Even if the value for a bin is zero, you should print the line.
Note 4: Do not modify the input text files (e.g., removing punctuations).