

CS426/525
Fall 2021
Project 4
Due 09/01/2022

In this project, you will implement a basic k-mer search program. As in previous projects, you will implement both serial and parallel versions of this program. For the parallel part, you will use CUDA to program in GPU.

Problem Statement

The *k*-mers of a string are basically the substrings of length *k*. The k-mer search problem is defined as follows:

- There are three inputs: a long *reference* string, multiple shorter *read* strings, and an integer *k* value.
- For the given *k* value, we divide each read string into k-mers (substrings of length *k*) and look for these k-mers on the reference string. We keep two variables for each k-mer: hit count and the locations of hits on reference. When a k-mer is found on the reference string, its hit count is incremented and the location on the reference string is recorded.

Here is an example, using the reference string *AGAGCCGAGACATTAG*, two read strings *AGATAGA* and *GATTACA*, and *k*=3. In this case, we have following k-mers for these read strings:

- *AGATAGA*: AGA, GAT, ATA, TAG, AGA
- *GATTACA*: GAT, ATT, TTA, TAC, ACA

When we look for these k-mers on our reference string *AGAGCCGAGACATTAG*, we have the following hit counts and the locations for these reads:

- *AGATAGA*:
 - AGA(count=2, at index 0,7)
 - GAT(count=0 at index -1)
 - ATA(count=0 at index -1)
 - TAG(count=1 at index 13)
 - AGA(count=2, at index 0,7)

Total k-mer hit count = 7, notice that we counted AGA twice, so we compute for each non-unique k-mers. Similarly, for our second read:

- *GATTACA*:
 - GAT(count=0 at index -1)
 - ATT(count=1 at index 11)
 - TTA(count=1 at index 12)
 - TAC(count=0 at index -1)
 - ACA(count=1 at index 9)

Total k-mer hit count= 3

Also notice we stated that index is -1 if count is 0.

Implementation Details

You should have separate files for serial and parallel implementations named as `kmer_serial.cu` (though you won't use GPU for the serial one, keep the file extension as `.cu` because we are using `nvcc` compiler) and `kmer_parallel.cu` as in order. Prepare a `Makefile` that will compile your code along with any necessary files (such as `util.cu`) that will produce 2 executables, `kmer_serial` and `kmer_parallel`. The input format for both programs is that they take 4 arguments in order:

1. reference file name
2. reads file name
3. k
4. output file name

Example:

```
$ ./kmer_serial reference_file.txt read_file.txt 3 output_file.txt
```

Format of the reference file is a single line consisting of a string of letters "G", "C", "A", and "T". In the read file, each read sequence is separated by a new line. We provided a `read_file` function for you to read these reads files in `util.cu`, you may use it. There are some other utility functions that you may find useful. The programs should calculate the total count of k-mers for each read and find the first of the index values in the reference that match with the k-mers. These values should be written into the output file indicated by the fourth argument. Output file format should be as follows: For each read sequence a single line consisting of the total count values succeeded by a space and the list of first index values for each k-mer separated by space should be printed. If a k-mer count in a read is 0, its index should be -1. The read sequences should be in the order they are in the input file. See the sample input and output files we shared for these input and output files.

There are many ways of solving this problem in both serial and parallel versions. For the serial one, a brute force solution might be tempting and easier to implement at first. However, a simple parallel implementation would easily beat the brute force solution in performance. If you implemented the brute force solution, you should implement at least one alternative of the serial program for a "fair" comparison between the parallel and serial execution. Here are some ideas:

- You can use hash-tables to not recompute already computed k-mers, or
- you might pre-search for all possible k-mers in the reference, or
- you might use Bloom Filters, De Bruijn graphs etc.

You may put alternative serial implementations in separate files but the efficient one should be in the `kmer_serial.cu` file, or you can keep them in the same file and comment out the brute force algorithm in your submission. Also don't forget to discuss it in your report.

Assumptions:

- For all string sequences, all letters can only consist of the letters "G", "C", "A", and "T".
- A reference string can be up to 1.000.000 characters, a single read sequence can be at most 200 characters. You can assume that all reads in the same file have the same length.
- There can be up to $1024 \times 20 = 20480$ reads in a read file.
- k value is less than the length of all read sequences and the reference sequence, and
- $2 \leq k \leq 14$

Language

You should use C or C++ language with CUDA (You can refer to course slides for specifications). You may use the following command to compile your files:

`nvcc filename.cu -o program`

Machines

You will need to use a machine with an NVIDIA GPU card. If you want to use a GPU available on our servers, email semi.yenimol@bilkent.edu.tr to ask for an account. Note that one GPU card cannot be used by two programs at the same time, thus it may affect your performance results. We may arrange a schedule for performance measurements.

Testing & Performance

You may simply use the `time` command in Linux to get the total execution time of the programs for a comparison. Run your programs with various sizes of reference string, various numbers of read sequences, and different k values. We also provided a large reference and read file for you to use, you can also generate your own inputs. In addition, use `nvprof` tool to analyze your parallel program running under GPU. You should get the result of the profiling tool with a few number of executions with various inputs. Put these results to your submission, also discuss it in your report.

Report

You should write a short report. **Reports should be in .pdf format**, submissions with wrong format will get 0. Your report should include:

- Your system specifications with following format:
 - **Operating System:**
 - **CPU:**
 - **GPU:**
 - **RAM:**
 - **HDD/SSD:**
- Detailed description of your implementations
- Detailed description of nvprof's profiling outputs.
- Analysis of your parallel implementation (communication cost, expected speedup etc.)
- Plots for execution times/speedups with various inputs
- Discussion of your results, don't forget to use nvprof's profiling outputs in your discussion.

Grading

- Serial implementations: 25
- Parallel implementation: 40
- Nvprof outputs: 5
- Report: 30

Hints for CUDA:

Try NOT to use dynamic arrays and data structures for kernel functions but use fixed ones even if it might cost you to repeat some computations. It might be better to flatten 2D arrays into 1D ones as it is easier for allocation in CUDA. Try to divide the work into small pieces as much as possible for a better GPU performance.

Submission

Put all relevant code, makefile, nvprof outputs and your report into a zip file.

Name the zip file: `yourname_lastname_p4.zip`

Submit to the respective assignment on Moodle.