# CS426/CS525

# Fall 2023

# Project 4

# Due 08/01/2024

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 *GCCAGAGCC* and *GATTACA*, and k=3. In this case, we have following k-mers for these read strings:

- *GCCAGAGCC: GCC, CCA, CAG, AGA, GAG, AGC, GCC*
- *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:

- *GCCAGAGCC:*
  - GCC(count=1 at index 3)
  - CCA(count=0)
  - CAG(count=0)
  - AGA(count=2, at index 0,7)
  - GAG(count=2 at index 1, 6)
  - AGC(count=1 at index 2)
  - GCC(count=1 at index 3)

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

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

Total k-mer hit count= 3

**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. Write a compile script compile.sh that will compile your code along with any necessary files (such util files) that will produce 2 executables, kmer_serial and kmer_parallel. The input format for both programs is that they take 4 arguments:

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

Format of the reference file is a single line 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. The programs should calculate the total count of k-mers for each read and write them into the output file indicated by the fourth argument. Output file format should be the total count values of each read sequence in their order in the input file. See the sample input and output files we shared for these input and output files.

**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*20=20480 reads in a read file.
- k value is less than the length of all read sequences and the reference sequence.

**Language**

You should use 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
```

## Machines

You will need to use a machine with an NVIDIA GPU card. If you want to use a GPU available in TRUBA or NVIDIA, check them out. Note that one GPU card cannot be used by two programs at the same time, thus it may affect your performance results.

## 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 number 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 put the result of the profiling tool to your submission as well and discuss it in your report.

There are many ways of solving this problem in both serial and parallel versions. For serial one, a brute force solution might be tempting and easier to implement at first but a parallel implementation would easily beat the brute force solution in performance. Try to implement at least one alternative of the serial program for a “fair” comparison between the parallel and serial execution. 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 are free to choose any string-matching algorithm you like for both parallel and serial implementation, there are many efficient algorithms that you might take a look at such as Knuth-Morris-Pratt’s algorithm, Boyer-Moore algorithm, Rabin-Karp algorithm, suffix trees, Aho-Corasick algorithm. You may put alternative serial implementation(s) in separate file(s) and put the one you liked most into the kmer_serial.cu file, or you can keep them in the same file and comment out the other algorithms in your submission. Also, don’t forget to 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:

- Detailed description of your implementations
- Detailed description of nvprof’s profiling outputs.
- Plots for execution times 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, compile script and your report into a zip file.

Name the zip file: `yourname_lastname_p4.zip`

Submit to the respective assignment on Moodle.