CS3210 Assignment 02

Student: Nguyen Quang Truong - A0293470B
Student: Nguyen Truc Nhu Binh - A0293856L

I. Introduction

1. Algorithm

The algorithm is straightforward: For every pair of $(sample,\ signature)$, attempt to find the first occurrence of the signature in the sample. If the first occurrence is found, immediately calculate the quality score.

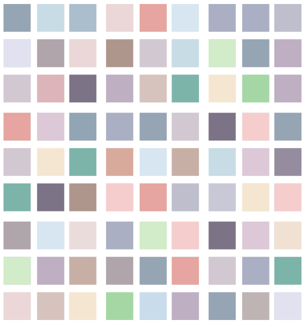
The pattern matching algorithm is as follows:

- Let the length of the sample and the signature be n and m, respectively;
- Find the first index i such that the substring $sample[i; i + m)$ $(i + m \leq n)$ matches the signature;
- If such index i exists, calculate the quality score with the given formula for the substring $quality[i; i + m)$.

2. Parallelization strategy & Kernel Configuration

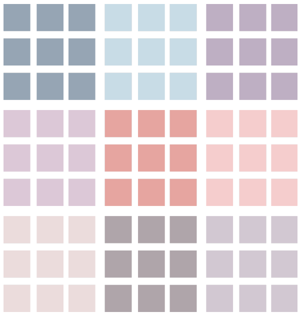
The pattern matching processes between each pair of $(sample,\ signature)$ are independent of other pairs. Thus, each processing unit on the GPU shall independently run the pattern matching algorithm and calculate the quality score.

An approach to implement the above idea is by creating a 2D grid of $(|all\_samples|,\ |all\_signatures|)$. Each block will process one pair of $(sample,\ signature)$. However, there is only one thread per block. Assuming each warp contains 32 threads, only one thread is used, leaving the remaining 31 threads idle.



Each color is a block. In this case, each block works on one pair of (i, j) only.

One way to solve this problem, mentioned in the CUDA C++ Programming Guide, is to choose a constant T, and each block will use $T \times T$ threads instead. This means that each block can handle $T \times T$ pairs of $(sample, signature)$. By delegating T samples and T signatures for a block, the grid dimension decreases to $(ceil(\frac{|all\_samples|}{T}),\ ceil(\frac{|all\_signatures|}{T}))$.



Each color is a block. In this case, each block works on $T \times T$ *pairs of* (i, j) $(T = 3)$.

Each block has a maximum of 1024 threads. Thus, $T = \sqrt{1024} = 32$.

3. Memory management

To perform pattern matching on the GPU, data has to be transferred to the kernel. Since it is complicated to copy `klibpp::KSeq` objects into the kernel, we instead allocated separate buffers for each field, essentially converting an Array of Structures (AoS) into a Structure of Arrays (SoA). To be precise, we concatenate all samples into one large buffer, and do the same for the qualities, the signatures, etc.

From our experiment, allocating to the managed memory and copying from host memory to device memory **only differ by less than a second, and should not really matter**. In our implementation, the best strategy we have found is:

- Allocating larger buffers (e.g. samples, qualities, etc.) to the managed memory;
- Constructing smaller buffers (e.g. signatures) on host memory and copying them to device memory.

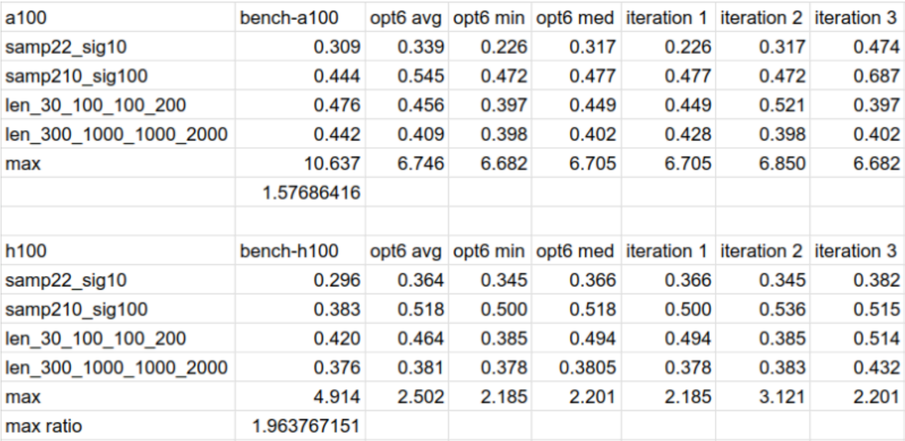
This strategy provides a good balance between memory access and memory allocation overhead.

II. Analyzing

There are 5 different types of tests. Unless otherwise specified, the parameters are the same as stated in the assignment description.

- Default (`max`);
- 22 samples (2 viruses) and 10 signatures (`samp22_sig10`);
- 210 samples (10 viruses) and 100 signatures (`samp210_sig100`);

- Sample length in range $[100; 200]$, signature length in range $[30; 100]$ (`len_30_100_100_200`);
- Sample length in range $[1000; 2000]$, signature length in range $[300; 1000]$ (`len_300_1000_1000_2000`).



a100	bench-a100	opt6 avg	opt6 min	opt6 med	iteration 1	iteration 2	iteration 3
samp22_sig10	0.309	0.339	0.226	0.317	0.226	0.317	0.474
samp210_sig100	0.444	0.545	0.472	0.477	0.477	0.472	0.687
len_30_100_100_200	0.476	0.456	0.397	0.449	0.449	0.521	0.397
len_300_1000_1000_2000	0.442	0.409	0.398	0.402	0.428	0.398	0.402
max	10.637	6.746	6.682	6.705	6.705	6.850	6.682
	1.57686416						
h100	bench-h100	opt6 avg	opt6 min	opt6 med	iteration 1	iteration 2	iteration 3
samp22_sig10	0.296	0.364	0.345	0.366	0.366	0.345	0.382
samp210_sig100	0.383	0.518	0.500	0.518	0.500	0.536	0.515
len_30_100_100_200	0.420	0.464	0.385	0.494	0.494	0.385	0.514
len_300_1000_1000_2000	0.376	0.381	0.378	0.3805	0.378	0.383	0.432
max	4.914	2.502	2.185	2.201	2.185	3.121	2.201
max ratio	1.963767151						

It can be seen that all the tests (except for the default test) are run in under 1 second, and there is no significant difference between our solution and the model one in terms of run time. On the default test, however, both solutions require much more time to compute, and the difference appears. Our solution is 1.6 times faster on A100-40 and twice as fast on H100-96 when compared to the model solution.

This suggests that the difficulty of the problem requires every constraint to be difficult. Reducing just a constraint (e.g. lowering the number of sequences, or their lengths) may reduce the difficulty of the original problem.

III. Optimizations

0. Not-so-optimized solution

Our first so-called "optimization" turned out to be a huge pessimization. The idea was to loop through each pair of $(sample,\ signature)$ in the host, then a kernel will be launched for each pair to pattern match.

In the kernel, let the length of the sample and the signature be n and m, respectively. To find the index i such that the substring $sample[i; i + m)$ $(i + m \leq n)$ matches the signature, there are $n - m + 1$ possible value of i. Thus, each block will determine whether its index may be the correct index. To further "parallelize" this determination process, each substring will be broken into T separate parts, and the check whether a substring is matched can be run in $O(\frac{m}{T})$.

It performed much worse than the single-threaded CPU version (and thus we will not include the benchmark for this attempt). During profiling, we noticed that there were too many `cudaKernelLaunch` (may reach up to 2,200,000 calls in the worst case). Profiling the model solution showed that it only called `cudaKernelLaunch` once.

1. **First optimization**

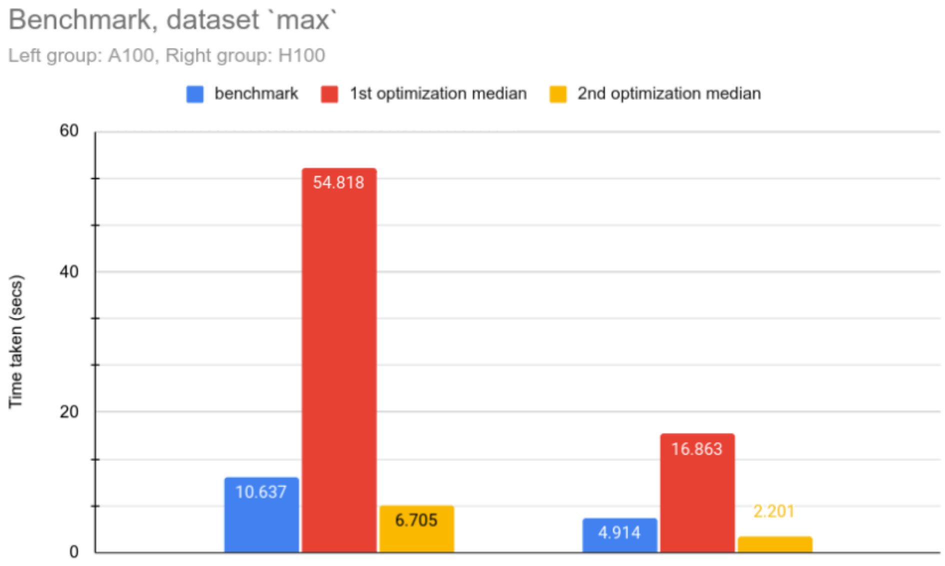
The idea stemmed from the fact that we needed to reduce the calls to `cudaKernelLaunch` as few as possible. After noticing the moderately small number of sequences, it was clear that those limits could act as the grid size.

Our first proper optimization was by creating the 2D grid of $(|all\_samples|,\ |all\_signatures|)$, with each block given one thread to perform pattern matching, as described in the Parallelization Strategy section.

2. **Second optimization**

The second optimization was by utilizing more threads in a block, also described in the Parallelization Strategy section.

3. **Benchmark result**



Each combination (GPU, optimization) of test configurations is run for 3 iterations. The time shown on the graph is the median.

IV. AI Usage Declaration

For this assignment, ChatGPT 4o was used to assist on the knowledge of the CUDA API.

Appendix

- Benchmark logs:
https://drive.google.com/file/d/1MRp2yofAATQQavOd6mfru5AMgOOizPS_/view?usp=sharing
- Google Sheets of the benchmark:
https://docs.google.com/spreadsheets/d/1NyQu_xMwX2of0WsPm2nPHTH0YNmHhshyeH8EkbpzSok/edit?usp=sharing
- Tests used for benchmarking:
https://drive.google.com/drive/folders/1T7ggN9rhxrxn-a7gTSvZ8VIefUQjHAva?usp=drive_link