CS3210 Assignment 2

Part 1: A brief description of implementation

1. Algorithm used by your program

The matching process begins by sliding the signature across the sample sequence to check for character matching while treating ‘N’ as a wildcard. If a match is found, the thread calculates a match score based on the quality scores of the sample qual, which contains the phred+33 in ASCII format. The quality score is derived by adjusting ASCII-encoded values and averaging them across the matched portion of the sequence. Once the match score is computed, it is stored in a global memory array at a position corresponding to the specific sample-signature pair.

2. Parallelisation strategy

We parallelised the matching process by assigning the samples and signature pairing to separate threads. Since each pair will have independent matches or no-match result and their own match scores, this removes the need to loop through the samples and signatures. This independence ensures that it is completely safe to parallelise the matching and can calculate the match scores for the various pairs to fully utilise the GPU processing power.

3. Choice and justification for grid and block dimensions

Block Dimension: 1 dimension with 256 threads per block.
A value that is a multiple of 32 is selected as a warp that contains 32 threads, so if it is not a multiple of 32, there will be threads that are not utilised, which wastes computational resources on the CUDA architecture. There weren't any significant time changes from 64 threads, so we just used a number with the lowest average timings, which is 256 for our case.

Grid Dimension: 1 dimension with ceil((number of pairs of signature and samples) / block size) per grid. The value is calculated to ensure that there are enough blocks to handle all the sample and signature pairs without any hardcoding of values.

4. How memory is handled and how shared memory is used in kernel

Memory is handled by splitting our structs into arrays to transfer the relevant data, specifically the size of the sample sequences and signature sequences, the starting indexes of the sample sequences and signature sequences and also the char* arrays of sample sequences and quality scores in (ASCII+33) and signatures. After the cuda kernel had finished executing the logic, we transferred the match scores back to the host to form MatchResults structs and add them to the matches vector.

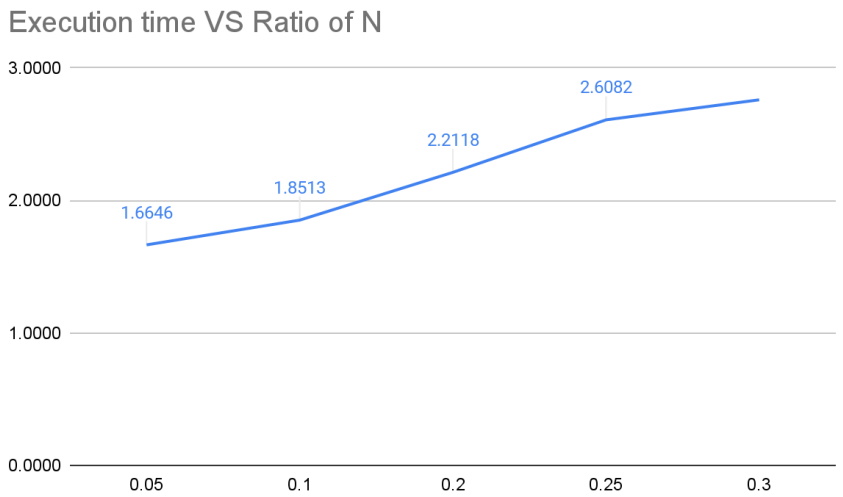
Part 2: Factors of the input file

1. How do different factors of the input overall runtime of the program?

a.Sequence lengths: From the data in the third part of our appendix, the overall runtime of the program increases as sequence lengths increase. The sequence lengths include both the length of samples and the length of signatures.

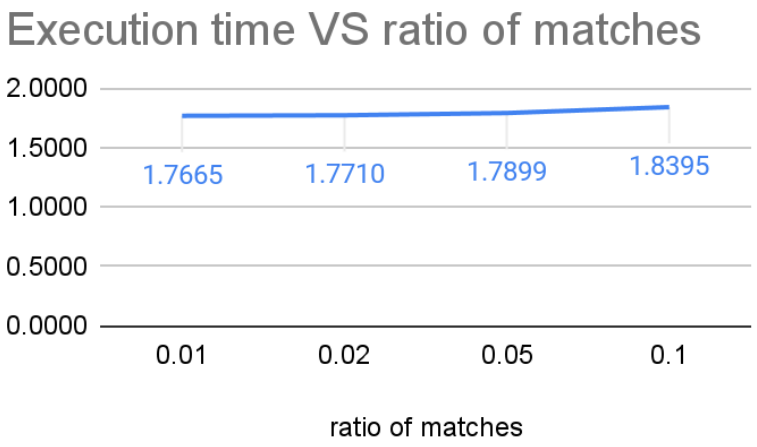
b. #sample and signatures: From the data in the third part of our appendix, the overall runtime of the program increases as the number of samples and number of signatures increase.

c. Percentage of N: To evaluate the effect of the percentage of N, I ran the program several times with all other settings the same, except that I changed the percentage of N in both samples and signatures. Below is the aggregated line graph.



As shown in the graph, the execution time increases as the ratio of N increase.

d. Percentage of signatures matching samples: Also to evaluate this, I ran the program several times with all other settings the same, except that I changed the Percentage of signatures matching samples. Below is the aggregate line graph.



As shown in the graph, the execution time slightly increases as the ratio of signatures matching samples increase

2. How can you explain these observations?

For factors a and b, they both affect the overall runtime by changing the total number of comparisons between samples and signatures. More specifically, as their numbers increase, the number of comparisons potentially needed will increase, thus increasing the runtime.

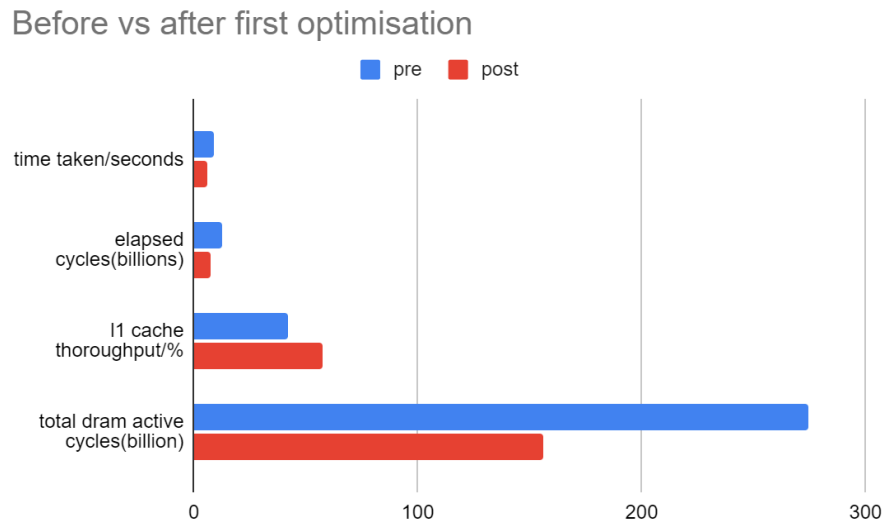
For factor c, since our algorithm for matching will, for each interval for each sample on a specific signature, terminate once there's a mismatch. Increasing the ratio of N will make it less possible to terminate the comparison and continue the process, thus increasing the runtime.

For factor d, similar to c, as the percentage of signatures matching the sample increases, the chances that the comparison can be terminated reduces and therefore increases the runtime.

Part 3: Performance optimisations

1. Reducing the number of for loop iterations to be sampleSize - signatureSize instead of checking for out-of-bounds inside the if-else clause together with other matching conditions

	time taken	elapsed cycles(billion)	l1 cache thoroughput(percentage)	total dram active cycles	
pre	9.07	12.79	41.94	274.28 billion	
post	5.71	7.29	57.63	156.31 billion	

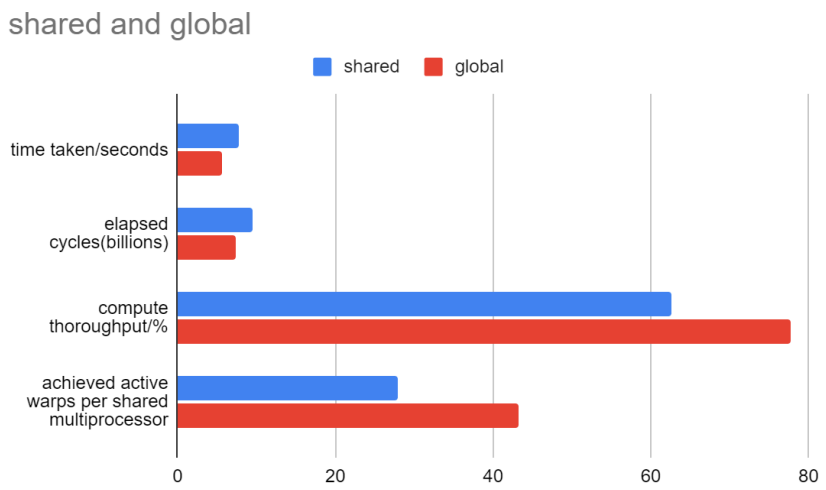


This is because other matching conditions require accessing global memory by checking the out-of-bounds samples when signatureSize + current index of sample > sample length in the if-else clause. This increases the number of accesses to global memory, which means there are more DRAM elapsed cycles, as accessing global memory in GPU is slower than GPU executions of other instructions. In addition, the cache throughput is higher in the optimised solution, which reduces access to global memory and results in lower DRAM elapsed cycles. In overall, this optimisation reduces the bottleneck of memory by reducing memory accesses.

2. Attempted to use shared memory to store the most frequently accessed parts of the sample

The most frequently accessed parts of the sample are the front parts of the sample, hence I made it such that the first 1024 characters are occupied and accessed by the shared memory while the others are accessed by global memory. However, this approach failed as it resulted in the program being slower than before.

	time taken	elapsed cycles	compute thoroughput	achieved active warps per shared multiprocessor
shared	7.83	9.43	62.56	27.93
global	5.71	7.29	77.79	43.17



A possibility is that the shared memory is being initialised, loaded and used only once, that is because for the algorithm, the data is already split to 1 thread per sample and signature pair. Hence the shared memory that is loaded will be used only once. The overhead of copying data into shared memory does not justify the faster performance due to accessing the faster shared memory. Hence, this will result in a slower performance as significant time is taken to load data from the global memory to the shared memory. This also reduces the compute throughput because a significant time is used to load memory from global to shared memory instead of executing computational instructions. There could also be possibly higher latency as the GPU is unable to optimise mixed memory access instead of only having global memory access. The active number of warps per shared multiprocessor is also reduced possibly due to shared memory constraints and threads being idle waiting for the slower global memory access instead of the faster shared memory access.

APPENDIX

1. Inputs used to reproduce results

The plot for execution time vs ratio of N, the input files are generated below:

./gen_sig 1000 3000 10000 **0.1** > sig.fasta

./gen_sample sig.fasta 2000 20 1 2 100000 200000 10 30 **0.1** > samp.fastq

The bolded number of where I change to produce different ratio of N

The plot for execution time vs ratio of matches, the input files are generated below:

./gen_sig 1000 3000 10000 0.1 > sig.fasta

./gen_sample sig.fasta 2000 **20** 1 2 100000 200000 10 30 0.1 > samp.fastq

The bolded number indicates the number of samples having signatures, therefore changing the ratio of matching signatures while other settings same

2. Nodes used for testing and performance measurements

The plot for execution time vs ratio of N is generated using node xgpi and slurm GPU h100-96, the command we use is:

srun --ntasks 1 --cpus-per-task 1 --cpu-bind=core --mem 20G --gpus h100-96 --constraint xgpi ./matcher samp.fastq sig.fasta

The plot for execution time vs ratio of matches is generated using node xgpi and slurm GPU h100-96, the command we use is:

srun --ntasks 1 --cpus-per-task 1 --cpu-bind=core --mem 20G --gpus h100-96 --constraint xgpi ./matcher samp.fastq sig.fasta

3. Data for part 2

The plot for execution time vs signature length, the input files are generated below:

./gen_sig 1000 **3000** 10000 0.1 > sig.fasta

./gen_sample sig.fasta 2000 20 1 2 100000 200000 10 30 0.1 > samp.fastq

The bolded number of where I change to produce different length of signatures

Sequence lengths vs runtime

Keeping minimum sample length at 100000

Minimum Signature length	3000	5000	7000	9000
Time taken using h100-96 architecture/s	1.78	1.91	2.04	2.13

The plot for execution time vs sample length, the input files are generated below:
./gen_sig 1000 3000 10000 0.1 > sig.fasta
./gen_sample sig.fasta 2000 20 1 2 **100000** 200000 10 30 0.1 > samp.fastq
The bolded number of where I change to produce different length of signatures

Keeping minimum signature length at 3000

Minimum Sample length	100000	125000	150000	175000
Time taken using h100-96 architecture	1.78	1.89	2.00	2.07

Number of sample and signature vs runtime

The plot for execution time vs number of signatures, the input files are generated below:
./gen_sig **1000** 3000 10000 0.1 > sig.fasta
./gen_sample sig.fasta 2000 20 1 2 100000 200000 10 30 0.1 > samp.fastq
The bolded number of where I change to produce different number of signatures

Keeping number of samples at 2020

Number of signature	500	600	700	800	900	1000
Time taken using h100-96 architecture	1.29	1.46	1.56	1.64	1.71	1.78

The plot for execution time vs number of samples, the input files are generated below:
./gen_sig 1000 3000 10000 0.1 > sig.fasta
./gen_sample sig.fasta **2000 20** 1 2 100000 200000 10 30 0.1 > samp.fastq
The bolded number of where I change to produce different number of samples, keeping the percentage of non-viral and viral samples consistent by using a ratio of 100:1 for non viral to viral samples

Keeping number of signatures at 1000

Number of samples	1212	1616	2020
Time taken using h100-96 architecture	1.36	1.59	1.78