CS3210 Assignment 2 Vijayaraghavan Vishnuprasath A0252513N

1.1) I used for loop over the sample and another nested for loop over the signature to brute force and check for a match. Along this for loop, I would keep track of the running score tally. If the sample and signature nucleotide don't agree, I would continue to the next for loop. If a match is found, it places the score in the matches array in the position corresponding to this sample and signature and returns.
To avoid complexity, I padded all signatures and samples to be same length of 10240 and 200000 respectively. I padded with the nucleotide X and i set its score to be 0 so that it doesn't contribute to the score of the match.

1.2) Each thread in my program finds matches between sample i and signature j.
Each thread block is responsible for finding matches between sample i and signature j for
blockId.x * blockDim.x <= i < (blockId.x + 1)* blockDim.x
blockId.y * blockDim.y <= j < (blockId.y + 1) * blockDim.y.
The kernels is made of enough blocks such that, gridDim.x * blockDim.x = 2048 and gridDim.y * blockDim.y = 1024 which are the number of samples and signatures respectively.

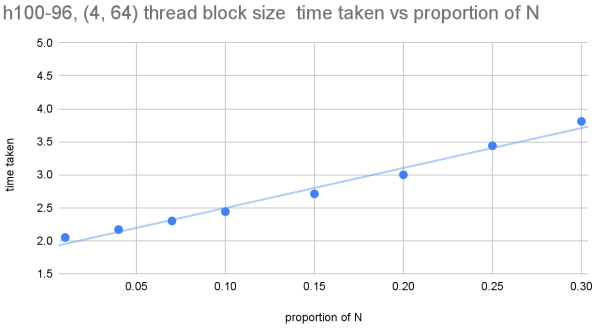
1.3) I chose grid size to be (512,16) and blockDim to be (4, 64). So each threadBlock is responsible for 4 samples and 64 signatures. I chose this number by timing how long it takes for different kernel sizes. The time taken in seconds is summarised in the table below for H100-96 gpu.
The x column shows blockDim.y / number of signatures in each thread block. The y column shoes blockDim.x / number of sample in each thread block.

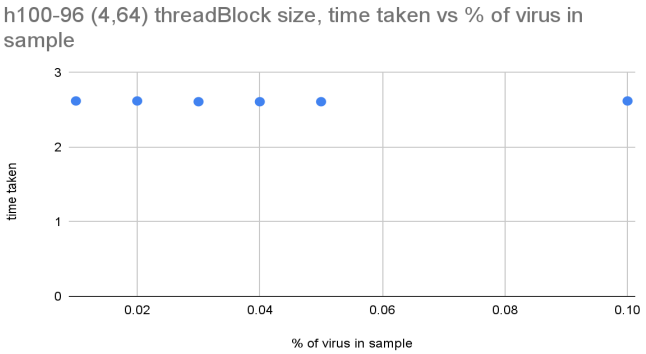
Column 1	1	2	4	8	16	32	64	128	256	512	1024
1	-	-	-	-	-	4.1	4.1	4	4.05	4	4
2	-	-	-	-	2.9	2.92	2.93	2.92	2.91	2.97	-
4	-	-	-	2.55	2.55	2.53	2.5	2.5	2.55	-	-
8	-	-	2.52	2.6	2.6	2.59	2.6	2.6	-	-	-
16	-	3.23	3.9	3.55	3.25	3.24	3.22	-	-	-	-
32	5.5	8.5	8.2	9.2	6.2	5.1	-	-	-	-	-

Hence, I chose (4, 64) as blockDim and accordingly (512, 16) as gridDim.

1.4) I allocated a char samples and a char scores 1d array in gpu of size numSamples*200000, then allocated a char signatures 1d array of size numSignatures*10240. Then I copied the data into these three arrays in row-major order and padded the remaining elements if necessary. I also allocated a matches 1d array of size numSamples * numSignatures. These four arrays were passed into the kernel for matching.

2.1)





In general, the time taken increases as the proportion of N characters increases from 1% to 20%. It also appears to be a linear relationship.
The time taken also doesn't seem to be affected as % virus in sample increases from 1% to 10%.

2.2) When the proportion of N in the samples and nucleotide increases, there is higher chance that two characters in the sample and signatures match. Since, the percentage of virus was kept the same across the different runs at 1%, the program has to loop through more characters before finding out there is no match between the sample and signature and breaking the loop. Hence, the program takes a longer time to complete.

Even though, the program breaks early when it has found a match. This would imply that if higher percentage of samples have virus, the program would be faster. However, as each thread runs as part of warp, even if that thread completes its execution, another thread can't take its place. Hence, the running time of the program is not affected by the percentage of virus in the samples. However, if the percentage of virus in samples is so high that all signatures of threads in entire warps contain virus, then the entire warp might break early which can then be replaced by another warp. This might potentially speed up the program.

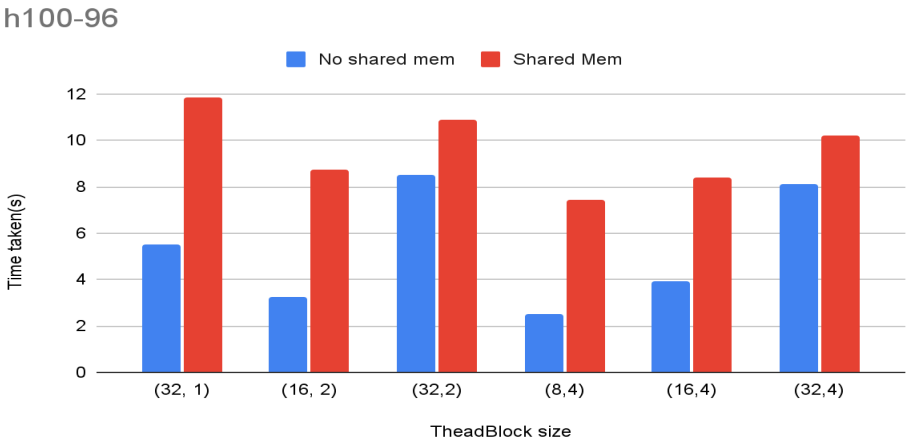
Even though, I didn't test the effect of various lengths of samples and signatures on running time, I believe it shouldn't have much of an effect on running time in my program. As I padded all samples and signatures to 200,000 and 10240 in length.

3) **Optimisation 1: Shared Memory**
For each thread, it loops over every character of sample and for each character as starting position it loops over the signature sequence to find a match. Hence, I tried storing either the sample or the signature in shared memory. Since, sequence of sample is 200000 long, each sample would take up 200kb. One sm in A100-40 only has 164kb of shared memory. So, I tried with the signatures instead.

Hypothesis: Storing the signatures needed in a thread block in shared memory would speed up execution.

Method: Before, finding the match, I used the threads of a thread block, to load the blockDim.y signatures into shared memory. Then, I synced the threads before finding the matches using signature data from shared memory. This might be faster as instead using costly global memory access, the signatures are loaded from shared memory which is quicker.
Even though the signatures were stored in row major order in global memory, in the shared memory I stored it in column major order. This would reduce the chance of bank conflict as within a warp, all the threads are trying to access the j'th character of their signature at the same time. In column major order, all the j'th characters would be adjacent.

Results: The left column shows the thread block sizes used, the middle and right column shows time taken without and with shared memory respectively. Unfortunately, the hypothesis was wrong.



Possible Explanations:
As each sample takes up 10kb shared memory space, this would limit the number of resident thread blocks on each sm. Hence, when one warp becomes idle during the global memory access, there would be a higher chance of another warp not available to run. Hence, it would limit occupancy. I used ncu to profile occupancy in a100-40 xpgp node.

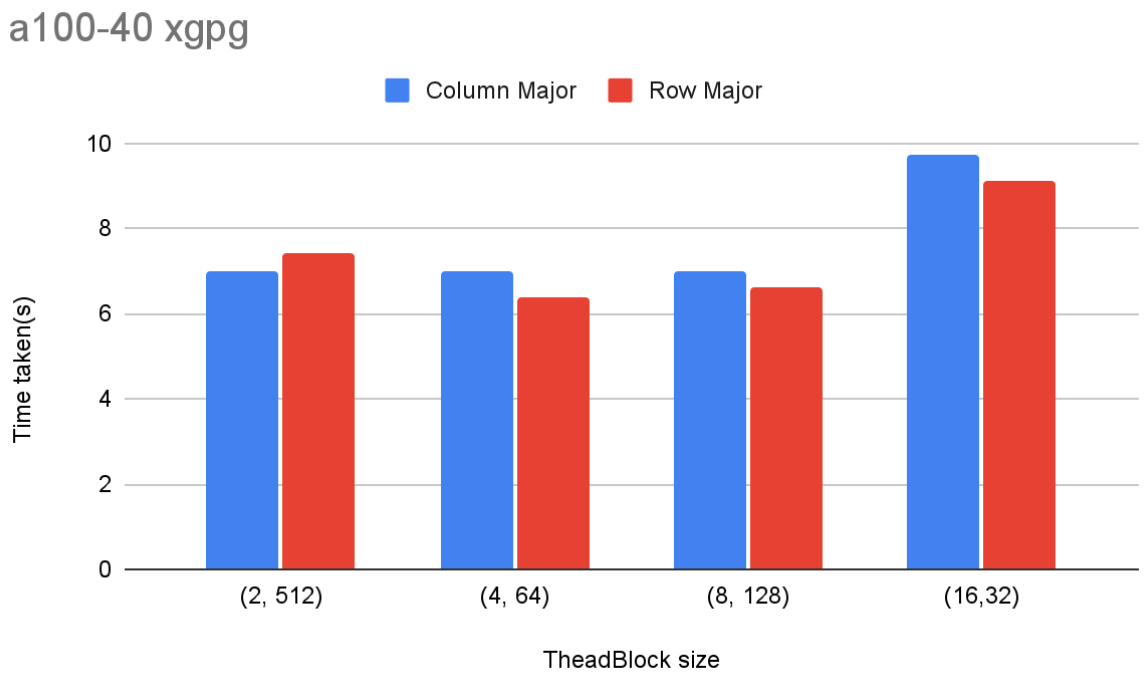
For shared memory with thread block size (8,4) which was the fastest, the occupancy was 6.25%.
For non-shared memory with thread block size (64, 4) the occupancy was 89%.
The data for this can be found in the google sheet in appendix.
So, I didn't use shared memory.

3) **Optimisation 2: Column Major Ordering of signatures.**
Since, each warp tries to execute in lock step, all threads will try to read the j'th character of their signature at the same time. If the signature array in global memory of Gpu is row ordered, then all these characters would be far away from each other. Hence, coalescing cannot be used.
However, if the signatures array in the global memory was in column major format, the j'th characters of all signatures would be contiguous. Hence, coalescing can be used which will reduce the number of global memory access.

Hypothesis: If signatures was in column major format, the kernel would execute faster than if it was in row major format due to above mentioned reasons.

Method: Before copying signatures array into gpu, I changed the signatures to a column major format and also changed how it was accessed in kernel code.

Result: There isn't much different between the row major and column major approaches. Sometimes, the row major approach is even faster than column major.



Possible Explanation:
Even though, all the global memory access from a warp cannot be coalesced, 32 bytes is still read for each global read. This is also cached in L1. So when the warp reads j + 1 character, the information is most likely in the cache. This helps the row major approach utilise the cache more.
In contrast, the column major approach would utilise the cache much less as when the j'th character is read by a warp, all the adjacent characters, which are coalesced, are also j'th character of other signatures. Hence, when reading j+1 character the warp has to most likely read it from global memory again. I used ncu to check cache throughput.
The row major approach had a L1 cache throughput of 89% while row major approach only had 42%. This data is in the google sheets in the appendix. So, I kept row major approach.

Appendix:

- Occupancy was profiled using ncu –sections Occupancy ./matcher ….
- Cache throughput was profiled using ncu –sections SpeedOfLight ./matcher….
- The naive implementation with 1d grid and 1d kernel is the first commit in the github repository with commit message naive
- The 2d grid and 2d threadblock are found from second commit onwards. To change the size of the threadblock, change the values samplesPerBlock and signaturesperBlock at the top of the file.
- The branch “Shared_mem” is the optimisation 1. Since, shared memory was allocated statically, it cannot be more than 48kb as such signaturesPerBlock should be less than or equal to 4.
- The branch “Column_major” is the optimisation 2. To change signatures array in kernel and host code from row to column major change columnMajorSignatures from false to true. Since, shared memory is not used here, signaturesPerBlock is not constrained to 4. Only the numbers of threads in each threadBlock should be less than or equal to 1024 as per Cuda.
- The last commit in the main branch is the submission commit.
- The raw data can be found in this google sheet. The run configuration such as gpu and node information and other inputs like kernel size is found in the sheets. https://docs.google.com/spreadsheets/d/1qMM8ksmXhb9BvePK0T7nzp6Q7Ggcs7G9opMacLvbYng/edit?usp=sharing