Yee Jun Hyeok Bryan A0252218L

Brief Description

Algorithm used by your Program

It is a simple brute-force algorithm that iterates every single element of the sample as a starting point, and iteratively checks each element from this starting point with the corresponding element in a signature for mismatches. If it went through the whole signature with no mismatches, it is a match.

Parallelisation Strategy & Choice and Justification for Grid and Block dimensions

Each block corresponds to a single sample, and each thread in a block corresponds to a single signature. This is because for the above algorithm, each element of the sample is iterated in the outer most for loop as a starting point, resulting in this same starting point being repeatedly accessed for each signature. Hence each signature can correspond to a single thread, comparing with this starting point in lock step to make it more cuda-optimised. Also, this allows coalescing from global memory, since all the threads are accessing this same starting point (and the following elements of the sample).

Moreover, the maximum number of threads per block is 1024[1], disallowing the maximum number of samples of 2200 from being distributed fully as threads into a single block (this is not an issue for signatures with its maximum number being 1000).

The sample DNA sequence should not be split into multiple sub-sequences and put into different blocks, as they would require a significant amount of repeats to be independently calculated (need to repeat an additional 10000 chars (max length for signatures) for every split into a different block), requiring many more accesses from global memory and slowing down the program.

Hence the grid size would be (no. of samples, 1, 1), and the block dimension would be (no. of signatures, 1, 1). The y & z axis do not correspond to anything in this assignment since this is not a 2D/3D task, and are hence set to 1 for ease of calculation.

How you handle Memory in your Program

The sample sequence, sample quality, and signature sequence are all placed in global memory. For the signature sequence, each character is accessed only once per block, and it is hence wasted copying if it is moved into shared memory and can be directly accessed instead. For the sample sequence, while each character is accessed multiple times by different threads in the block, it is cached and is hence not significantly slow. Moreover, the shared memory for the A100 is 164KB[2], making it unable to fully store a sample with max length of 200000, and requiring it to be split into 2. This would result in the same issue as described above with multiple sub-sequences, slowing down the code instead if it were put in shared memory. For sample quality, since it is rarely used (only 1% of samples have viruses), it is left in global memory.

The resulting scores of the calculations from the GPU uses pinned memory to transfer the results back to the host. This is because the large amount of space required to store all possible combinations found (2200 samples * 1000 signatures * sizeof(double)) makes it slow to transfer to the device and back, even though only a small subset (1% of samples) will be used most of the time. Hence it uses pinned memory, where the space is malloc'd within the host, so that no data transfer is necessary to the device and only the required data is sent to the host.

How do different factors of the input affect the overall runtime of the program? How can you explain these observations?

(All the corresponding graphs can be found in the appendix)

As the *sample sequence length* increases, the time taken increases linearly. This is because with the number of samples with matches being at a low 1%, the algorithm will search through 99% of the samples from start to end. Hence, the time taken would scale linearly with the sample sequence length. This can likewise be seen in the *signature sequence length*, where increase in length increases the time taken generally, since the signatures will be compared with from start to end for most samples. The minor fluctuations of ~0.2s is likely because each change in signature must have a newly generated sample, causing randomness in the length of the sample sequence lengths.

As the *number of samples* increases, the time taken linearly increases. This is because most of the samples will be searched from start to end (since most won't have matches), and hence an increase in the number of samples will increase the time taken linearly. The same applies to the *number of signatures* due to the same reasoning.

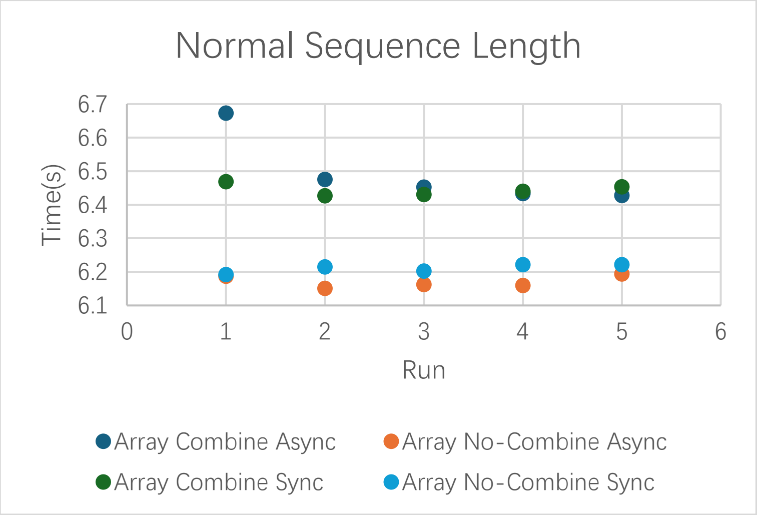
As the *% of N for Samples* increases, the time taken increases at an increasing rate at the start. This is likely because while the percentage of samples with matches remains constant, the number of partial matches with the signatures increases since N can match with any character, increasing the number of characters that need to be checked before determining they are not a match. This is especially true at near 100% for N, causing an accelerating increase. At 100% for N however, there is a steep drop in time taken since it results in 100% of samples matching with any signature, and moreover only requires comparing the number of characters equal to the length of the signature instead of comparing the entire sample. All these can be similarly seen in *% of N for Signatures* for the same reasoning.

As the *% of Samples with Matches* increases, the time taken fluctuates at a range of around ~0.1s with seemingly no relation. This is likely because even though threads can terminate early if it finds a sample-signature match, they still need to run in lockstep with their warps. Hence, unless the entire warp (different signatures) similarly finds a match with the sample, it will still take the same number of cycles to run.

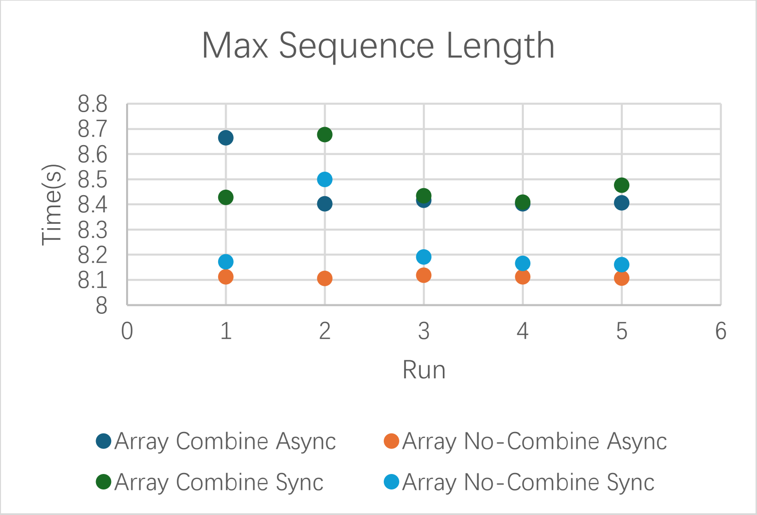
As the *Virus Count per Matching Sample* increases, the time taken mostly linearly increases. This is likely because while the number of samples with virus remains the same at 1%, the number of viruses for each of these virus-containing samples increases, increasing the number of scores and details that need to be returned to the host. Hence, the time taken would increase, especially since memory transfer is the bottleneck in this problem.

Describe at least two specific performance optimisations you attempted with analysis and supporting performance measurements.

1) By combining all the sample sequences into 1 array, all the sample qualities into 1 array, and all the signature sequences into 1 array, all the required data can be sent to the device with just 3 calls to cudaMemcpy(), instead of having to make ~1000 calls for signatures and ~4400 calls for samples (sequence + quality). This is done by fixing each sample to the max possible sequence length of 200000, and each signature to the max possible sequence length of 10000, and filling up the rest of the chars with '\0' to indicate the sequence ending. This makes it significantly more efficient in theory since the time spent making ~5400 separate calls with gaps of time between each transfer is drastically reduced, especially since the biggest overhead is transferring the data to the device.



However in practice, there was a slowdown instead. Since it could be due to cudaMemcpyAsync() causing the time gap between each memcpy to be shorter since they are already in the queue and the function has already been called (less function call overhead), I tested them with the synchronous cudaMemcpy() as well. However, the non-combine version is still faster even without async, as seen in the graph on the left.



Since it could also be due to the increased number of bytes that are transferred for the combined version, I tested them with the max sequence length for both samples (200000) and signatures (10000), guaranteeing that they all send the exact same number of bytes to the device. These can be seen in the graph on the left. However, the non-combine version still wins out the combined version even for this case, showing that the slowdown is not due to the increased data transfer.

The most likely reason is due to the combined version requiring a separate calloc(), memcpy(), and free() within the host, slowing down the runs instead (especially the slow memcpy()). This is unlike the non-combined version, which simply gives a pointer to the existing C-style string array to cudaMemcpyAsync(), requiring almost no extra work within the host. Unfortunately, this cannot be tested for since it is required to malloc to be able to combine the arrays.

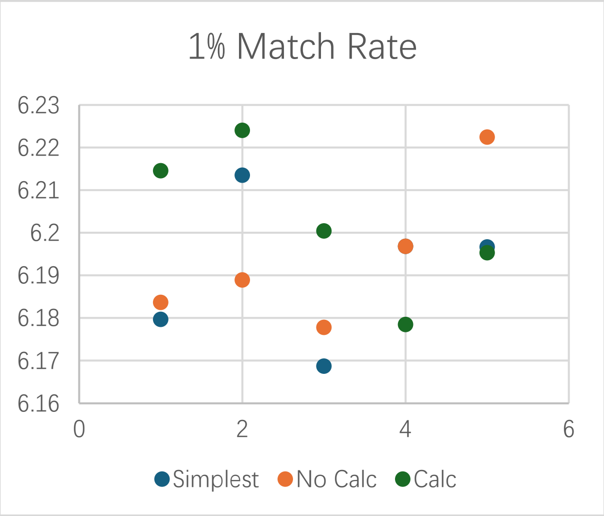
2) Retrieving scores from the device requires creating a large C-style array of size samples.size() * signatures.size() to store all potential matches, as vectors are not supported in cuda. Since each sample-signature pair has a unique index, they do not need any synchronisation to write to this large array. The issue however is how to efficiently know which elements in this large array has the result values, especially since only a small subset (1% of samples) will have matches.

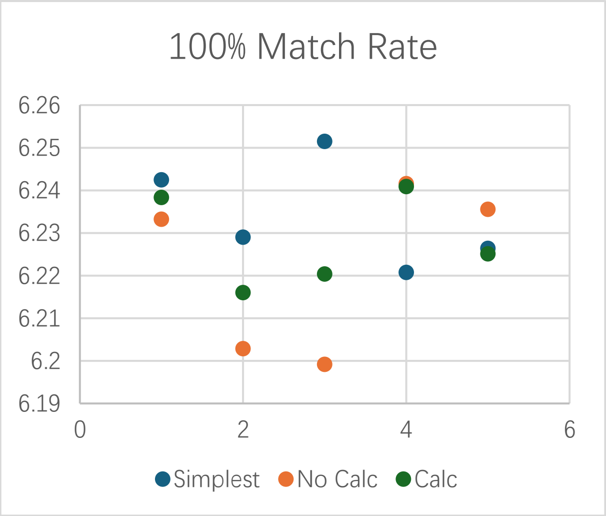
The simplest way is to memset() the large array with 0s, and iterate through the entire array for any non-zero scores, making this slow and inefficient.

In order to speed this up, I used atomicAdd(match_count, 1) recommended by cbuchner1[3] in order to obtain a unique index each time I run the function. It is guaranteed to be unique since no other code modifies match_count (except for the initialisation to 0), the value of match_count can only increase, and the function atomically increments the counter and returns the previous value. The thread is then free to add the required value at this unique index with no synchronisation.

One possible way to use atomicAdd() would be to add the score directly at this unique index, and separately store the sample and signature indexes (needed to obtain their names) using this same unique index. This would require using another 2 arrays of size samples.size() * signatures.size(), but would require no calculation to obtain these indexes.

Another possible way is to add the score at its unique sample-signature pair index (similar to the simplest way), and then make 1 array of size samples.size() * signatures.size() to store this index, using atomicAdd() to append them at incremental indexes. This would require only 1 additional array instead, but would need the host to calculate the sample and signature indexes using the slow divide and modulo operations.





After running all 3 variants with both the default 1% match rate and 100% match rate (to increase the number of matches that need to be returned), neither of the atomicAdd() variants seem to have much of a speed increase as compared to the simplest variant. This is likely because while there is a speedup in terms of not needing to iterate through the large array, there is a slowdown due to requiring the creation of 1 or 2 more similarly large arrays. Moreover, atomicAdd() requires synchronisation with other threads, possibly adding to the slowdown. Since the no calc variant is on average the fastest (albeit by only 0.01s), it is chosen.

APPENDIX

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications, Table 21: Maximum number of threads per block.

[2] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability-8-x, 16.7.3. Shared Memory.

[3] https://forums.developer.nvidia.com/t/threads-branching-and-writing-to-global-memory/66566, Post by: cbuchner1.

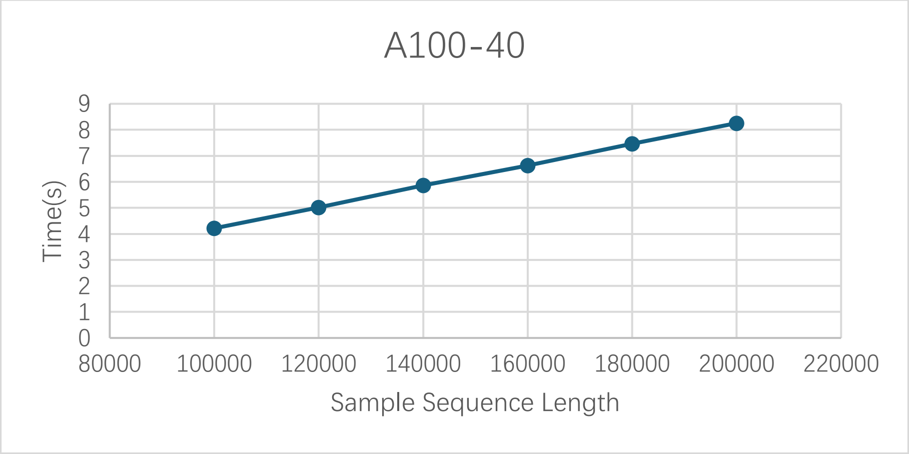
Different factors of the input affecting the overall runtime of the program (graphs):

Different factors of the input affecting the overall runtime of the program (code):

All the code used to generate and run the tests can be found in the files "run_generated.sh" and "generate.sh", and were run on xgph10 a100-40.

All the raw data for this section can be found in "run_generated_truncated.out" and "Data.xlsx". "run_generated_truncated.out" has % of N at 100% removed due to the huge file size (shows every single sample-signature pair).

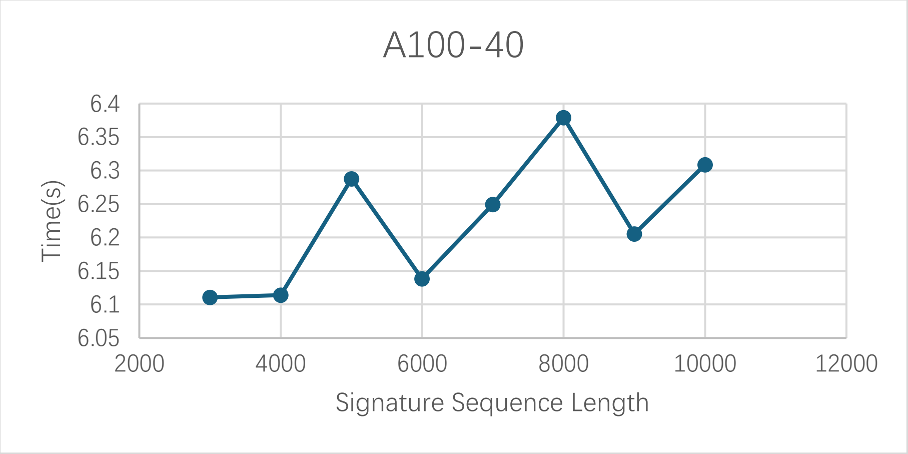
Sample Sequence Length:



```
./gen_sample sig.fasta 2000 20 1 2 <min_len> <max_len> 10 30 0.1
```

where `<min_len> == <max_len>` for the same run.

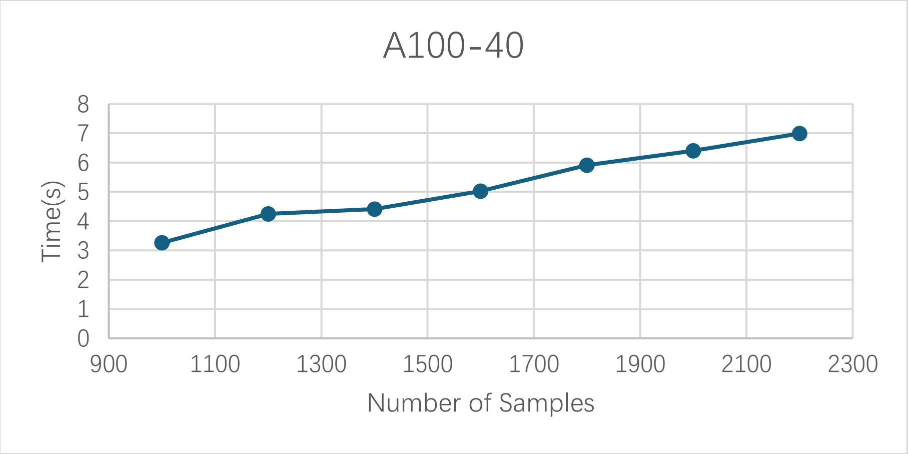
Signature Sequence Length:



```
./gen_sig 1000 <min_len> <max_len> 0.1
```

where `<min_len> == <max_len>` for the same run.

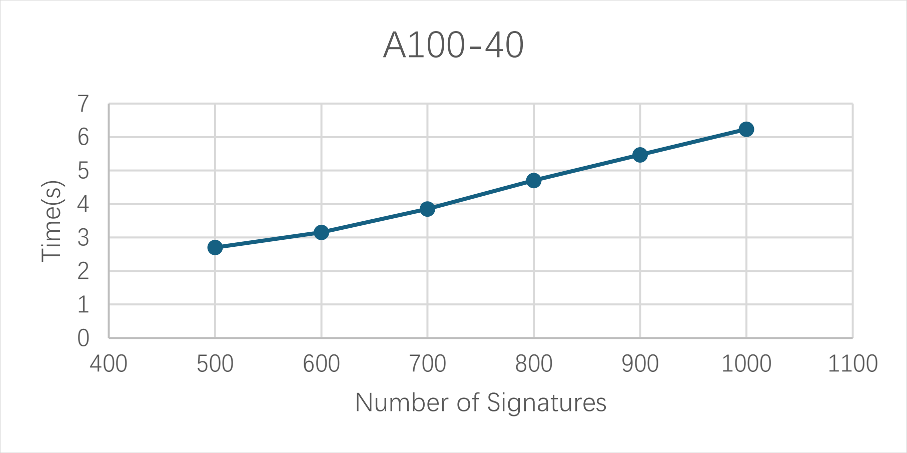
Number of Samples:



```
./gen_sample sig.fasta <num_no_virus> <num_with_virus> 1 2
100000 200000 10 30 0.1
```

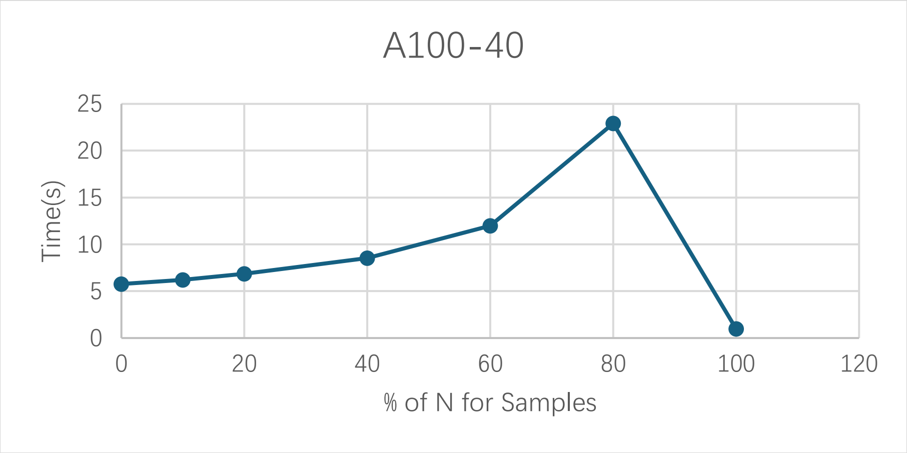
where `<num_no_virus> + <num_with_virus>` changes between runs, while `<num_no_virus> / <num_with_virus>` remains constant at 1%.

Number of Signatures:



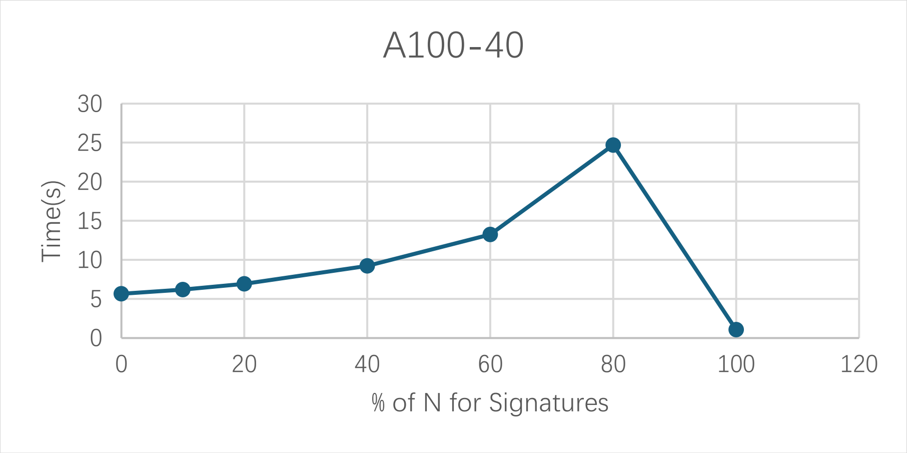
```
./gen_sig <num_sig> 3000 10000 0.1
```

% of N for Samples:



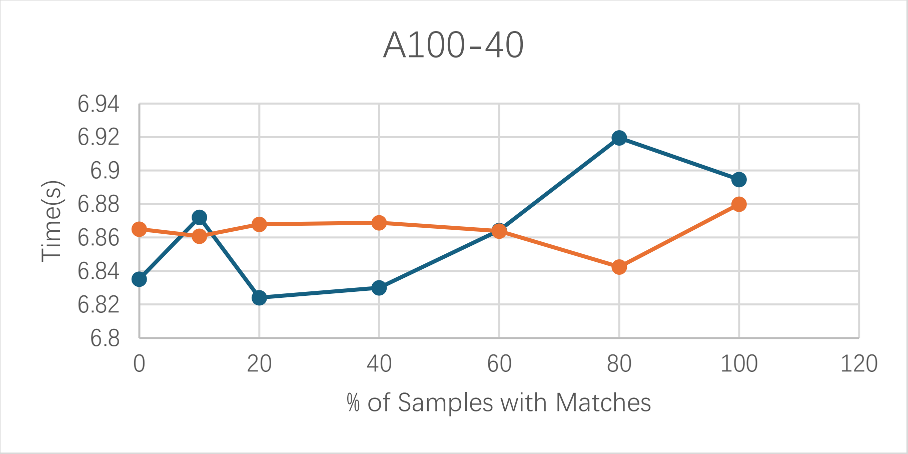
```
./gen_sample sig.fasta 2000 20 1 2 100000 200000 10 30 <n_ratio>
```

% of N for Signatures:



```
./gen_sig 1000 3000 10000 <n_ratio>
```

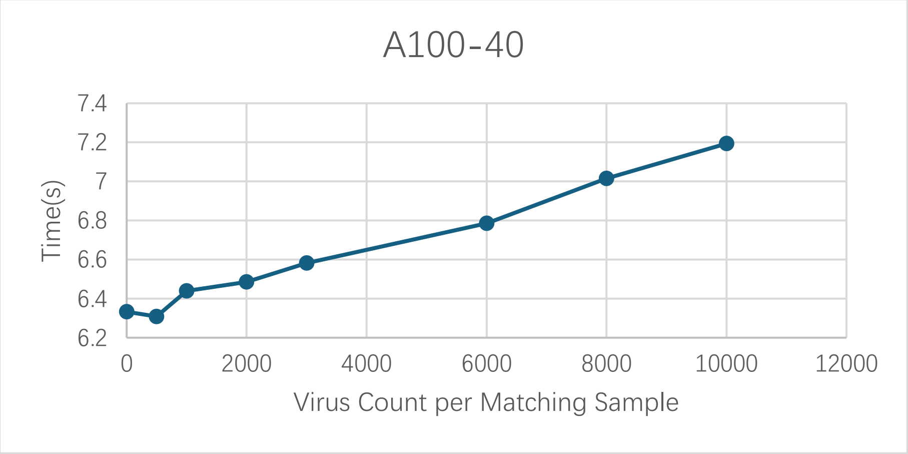
% of Samples with Matches:



```
./gen_sample sig.fasta <num_no_virus> <num_with_virus> 1 2
100000 200000 10 30 0.1
```

where `<num_no_virus> + <num_with_virus>` is fixed at 2200 between runs, while `<num_with_virus> / 2200` changes. This was run twice due to the high fluctuations.

Virus Count per Matching Sample:



```
./gen_sample sig.fasta 2000 20 <min_viruses> <max_viruses>
100000 200000 10 30 0.1
```

where `<min_viruses> == <max_viruses>` for the same run.

Combine Arrays:

***All the raw data can be found in "Data.xlsx" and were run on xgph10 a100-40.**

***Array combine code is found in "kernel_skeleton_arraycombine.cu".**

Normal Sequence Length

```
./gen_sig 1000 3000 10000 0.1
./gen_sample sig.fasta 2000 20 1 2 100000 200000 10 30 0.1
```

The provided sequence range in the pdf document was used for 5 runs.

Max Sequence Length

```
./gen_sig 1000 10000 10000 0.1
./gen_sample sig.fasta 2000 20 1 2 200000 200000 10 30 0.1
```

The max sequence ranges were used for 5 runs.

Retrieve Scores:

All the raw data can be found in "Data.xlsx" and were run on xgph10 a100-40.

Simplest, Calculate, No Calculate codes can be found in "kernel_skeleton_simplest.cu", "kernel_skeleton_calc.cu", and "kernel_skeleton.cu" respectively

1% Match Rate

```
./gen_sig 1000 3000 10000 0.1
./gen_sample sig.fasta 2000 20 1 2 100000 200000 10 30 0.1
```

The provided match rate in the pdf document was used for 5 runs.

100% Match Rate

```
./gen_sig 1000 10000 10000 0.1
./gen_sample sig.fasta 0 2020 1 2 100000 200000 10 30 0.1
```

The max match rate with max signature sequence length was used for 5 runs.