

## 1. Brief description:

### Algorithm:

The program employs a brute-force sliding window approach to determine whether a viral signature appears in a sample. For each sample–signature pair, the algorithm slides the signature over every possible alignment in the sample, comparing each character while allowing for unspecified nucleotides ('N') to match any nucleotide. If a match is found, a score is computed by summing the quality scores over the matched region. To efficiently calculate both the integrity hash and the maximum match score, a parallel prefix-sum (scan) algorithm is used. This technique is in [GPU Gems 3, Chapter 39](#) and implemented similarly to the code in Appendix 1 and this allows us to compute these sums in logarithmic time and get the sum in  $O(1)$ .

### Parallelisation Strategy:

Each sample is assigned to its own CUDA block. Within each block, every thread is responsible for processing one signature. Each thread slides its assigned signature over the sample, comparing character-by-character to determine if and where a match occurs. The prefix-sum algorithm is applied on the quality scores of all samples to quickly compute partial sums. These partial sums are used to obtain both the integrity hash of the entire sample and the maximum match score for any detected match.

### Grid and Block Dimensions:

**Grid dimensions:** The grid is configured with the x-dimension equal to the number of samples, ensuring each sample is processed by a separate block.

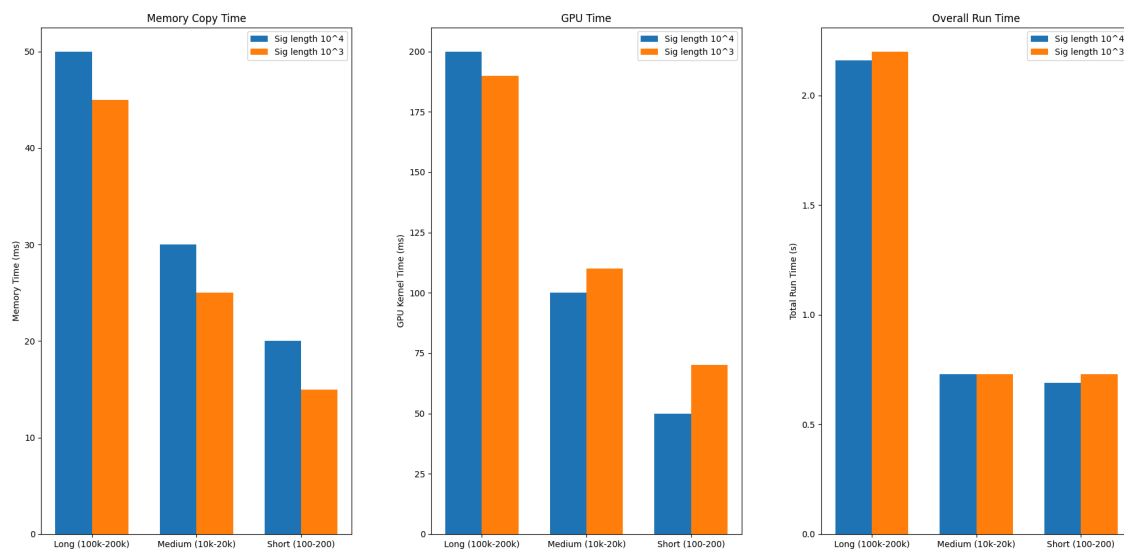
**Block dimensions:** Within each block, the number of threads is set to the number of signatures. This design allows each thread to handle one signature in parallel.

### Memory Handling and Shared Memory Usage:

- The sample sequences and signature sequences are stored in global memory. Signature sequences are flattened as one contiguous block of data in global memory and the same as sample sequences (more explanation on this in part 3). When the comparison is needed, the character of each the sig and sam are loaded to thread variable
- To minimize overhead, memory for samples, signatures, and quality strings is allocated once (or batched) rather than repeatedly within inner loops.
- Shared memory in prefix-sum is used to hold partial sums during the prefix-sum (scan) operations. In the prefix sum algorithm, you need to perform a reduction (summing values) across an array. Reading from global memory is relatively slow compared to shared memory. By loading a block of data from global memory into shared memory, you can perform multiple iterative summations efficiently. Once the data is in shared memory, the threads cooperate to compute the prefix sum. This is done in multiple steps, where each step the block's threads combine elements in pairs by using a tree-based reduction. The result computed in shared memory was written back to global memory to be used after that.

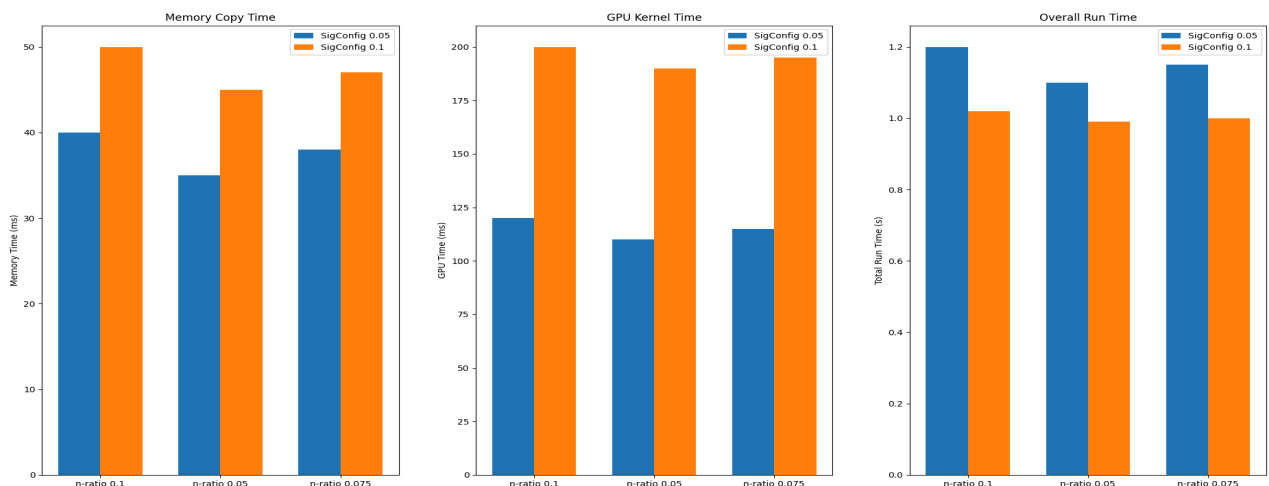
## 2. Different factors of the input that affect the runtime of the program

### Different input for sample and signature sequence length.



The graph (based on raw data on Appendix 2) demonstrates that when running on H100 with 2020 samples and 1000 signatures, increasing the sequence length significantly raises memory copy and GPU processing times, resulting in a longer overall run time. The input configuration determines the lengths of the sample and signature sequences. Longer sequences require more data to be transferred from host to device (increasing memory copy time) and involve more iterations for checking mismatches in the kernel loops (increasing GPU execution time). In contrast, shorter sequences require less memory bandwidth and fewer computational iterations, leading to faster execution.

### Different input for n-ratio

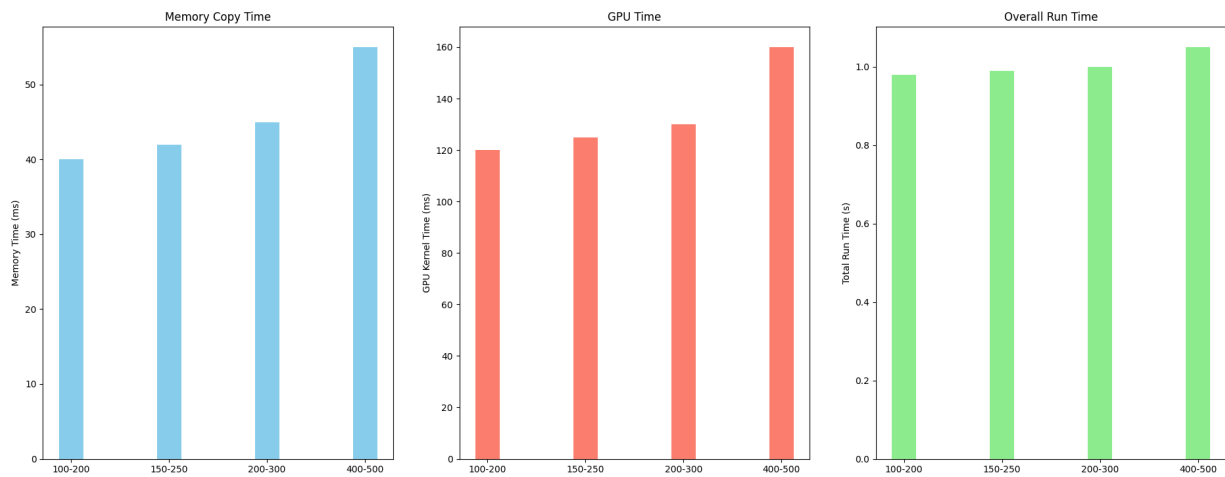


When varying the n-ratio in the input, differences in sequence ambiguity can alter the matching behavior. A lower n-ratio typically means fewer ambiguous bases, which can reduce the number of tolerated mismatches and might allow the kernel to short-circuit the inner loop more often

,especially since the kernel breaks out of the loop immediately when it encounters a non-matching, non-ambiguous character. In contrast, a higher n-ratio may require more careful checking before the break condition is met, resulting in longer GPU kernel times and slightly different memory behavior. The graph (based on raw data on Appendix 2) illustrates these trends by showing how memory time, GPU time, and total run time vary with the sample's n-ratio for two different signature configurations.

### **Difference min, max virus appear in a sample with virus**

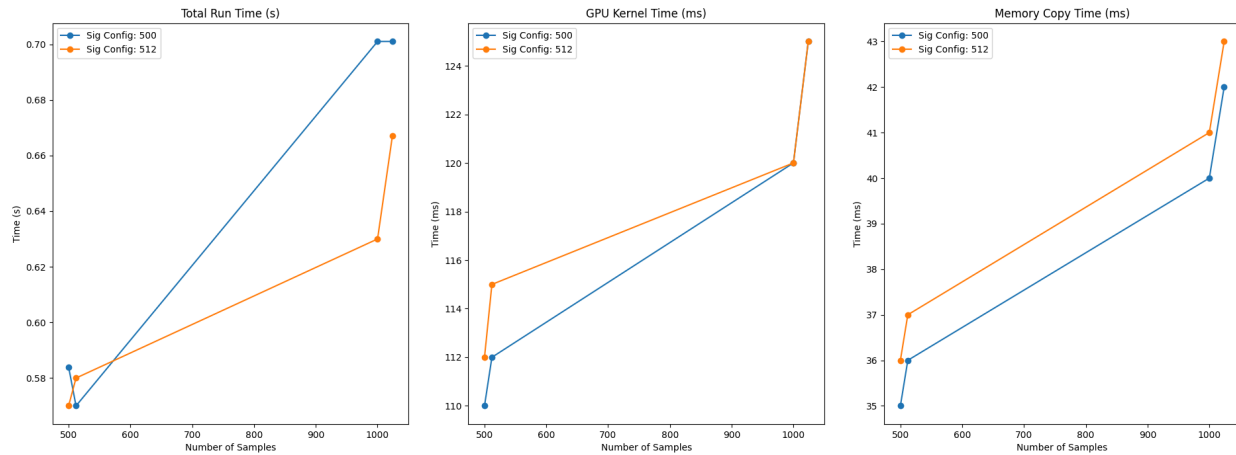
Effect of Increasing Virus Count in Sample (H100-96)



The graph above (based on raw data in Appendix 2) shows that as the virus count per sample increases, both memory copy and GPU kernel execution times increase, leading to a longer overall run time. As the virus count range increases—from 100–200 up to 400–500—the overall processing time increases. This is due to the higher number of ambiguous bases requiring more iterations and comparisons before the kernel's break condition is met and requiring operation to compare and find the best match score, which in turn increases both the memory transfer and GPU execution times, leading to a longer total run time.

### **Difference number of samples and signatures**

Performance Comparison: Varying Number of Samples (Grid Size) and Signatures (Block Size)



The graph above (based on raw data in Appendix 2) shows that while increasing the number of samples (i.e. grid size) increases the total run time due to higher scheduling overhead, using signature configurations that align with the warp size (e.g., 512 threads) improves GPU efficiency and reduces kernel execution time. Increasing the number of samples (which increases the grid size) generally increases the total run time. This is because more blocks need to be scheduled on the GPU. Comparing the two signature configurations, one with 500 signatures and one with 512, we see that using 512 (an exact multiple of 32) tends to offer slightly better or more consistent performance. The extra four threads in a 512-thread block (which forms 16 complete warps) allow for more efficient warp scheduling and memory coalescing, while 500 threads can leave part of a warp under-utilized. Overall performance is affected by both the number of samples and signatures. When the grid (samples) is large, keeping the GPU busy requires having enough blocks to fully occupy all SMs.

### 3. Optimization

We generate a test base on these conditions for test:"

# Signature configurations: "num\_signatures min\_length max\_length n\_ratio"

SIG\_CONFIGS=( "1000 3000 10000 0.1" "1000 300 1000 0.1" )

# Sample configurations: "num\_no\_virus num\_with\_virus min\_viruses max\_viruses min\_length max\_length min\_phred max\_phred n\_ratio"

SAMPLE\_CONFIGS=( "1000 10 100 200 100000 200000 10 30 0.1" "1000 10 100 200 50000 100000 10 30 0.1" "1000 10 100 200 10000 20000 10 30 0.1" )"

(3 signatures \* 2 samples) 6 tests total.

noop: the cuda code have no optimization

op1: the cuda code using first optimization

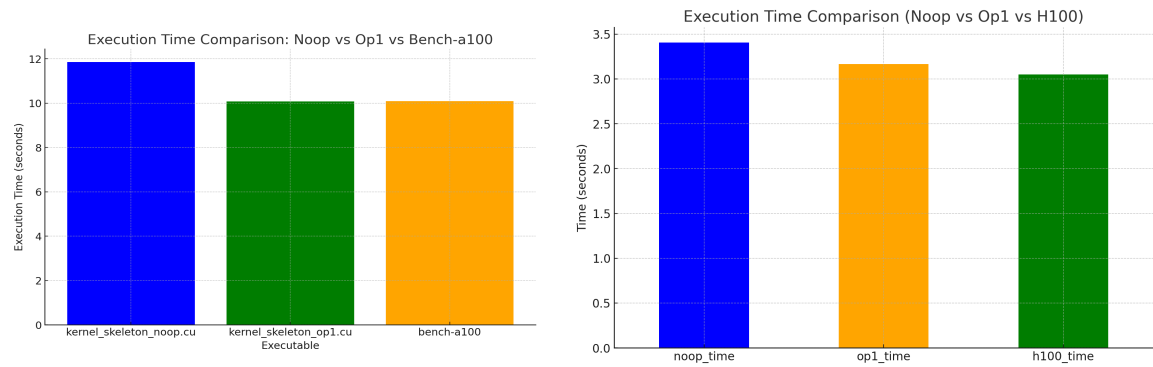
op1op2: the cuda code using first and second optimization

#### First optimization:

The first optimization involves precalculating the prefix sum array for each sample’s qualification scores using the Parallel Prefix Sum (Scan) algorithm in CUDA(Appendix 1). The core idea is to convert the character array of qualification scores into an integer array, and then apply the scan algorithm to obtain a prefix sum for each sample.

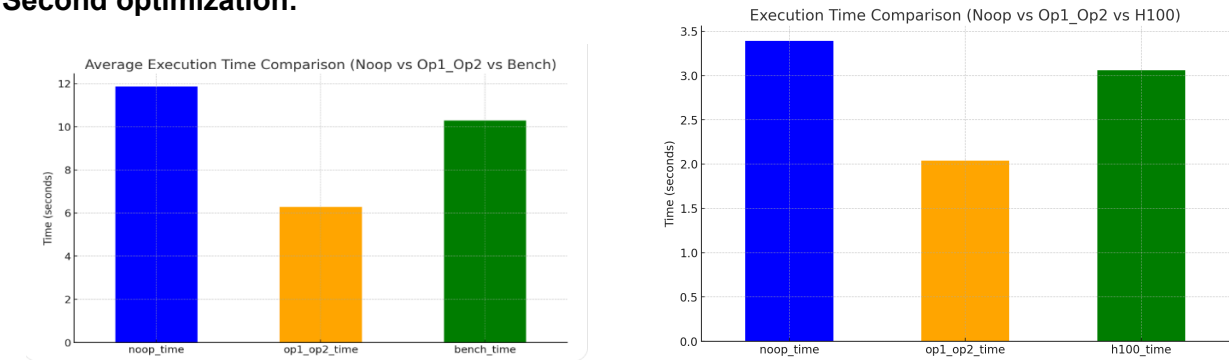
In the previous implementation, the qualification sum was computed on-the-fly using a sliding window approach, which led to unnecessary recalculations and overhead. By precalculating the prefix sum for each sample, we eliminate this overhead, reducing the complexity of subsequent calculations.

Instead of recalculating the sum every time we match a substring, we use the precomputed



prefix sum array (d\_prefSum) to directly access the sum for a given substring range by simply using the start and end indices. Additionally, this optimization allows us to compute the integrity value more efficiently: instead of recalculating it separately, we simply take the last element of the prefix sum array and apply the modulus operation (% 97). Table is from Appendix 3

**Second optimization:**



The second optimization focuses on flattening 2D arrays (e.g., char\*\* for samples and signatures) into 1D arrays (e.g., char\*). This change is aimed at improving memory access patterns and enhancing cache locality. In the original implementation, 2D arrays of pointers were used. These arrays were allocated in separate memory locations, leading to disjoint memory access patterns and inefficient use of the cache. By flattening the arrays into a single 1D array, we store the data in a continuous block of memory, which improves spatial locality. This means that when we access elements of these arrays, the likelihood of cache hits increases, leading to faster memory access and improved performance. Table is from Appendix 3

## Appendix

We use Chatgpt 4-o to assist answering questions about CUDA, shell and help us to write the report.

### Appendix 1: Parallel Prefix Sum (Scan) with CUDA

Document:


<https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda>

Code: [https://github.com/dhoefflinger/CUDA/blob/master/MP4/MP4.2/scan\\_largearray.cu](https://github.com/dhoefflinger/CUDA/blob/master/MP4/MP4.2/scan_largearray.cu)

Code: <https://github.com/mattdean1/cuda/blob/master/parallel-scan/scan.cu>

### Appendix 2:

Raw data for part 2:  Raw data

Script to run for part 2:  script

All raw data is produced by running 3 times the scripts and calculating the average value. All are run on H100-96

### Appendix 3:

Run using bench, `#SBATCH --gres=gpu:a100-40:1`

Run with the test having SAMPLE\_CONFIGS "1000 10 100 200 100000 200000 10 30 0.1"

Job	noop_time	op1_time	bench_time
1	11.8541	10.0838	10.0941
2	11.9275	10.0753	10.0678
3	11.9165	10.089	10.2882
4	11.8262	10.1699	10.1199
5	11.8187	10.1058	10.1699

table from job-gpu\_562456.slurmlog

Run with the test having SAMPLE\_CONFIGS "1000 10 100 200 100000 200000 10 30 0.1"

Job	noop_time	op1_op2_time	bench_time
1	11.8583	6.27911	10.2852
2	11.8358	6.27911	10.0807
3	11.8817	6.28545	10.1767
4	11.8551	6.28811	10.1199
5	11.7961	6.26265	10.1352

from job-gpu\_562463.slurmlog

Run using bench, h100 `#SBATCH --gpus=h100-96`

The average times are as follows:

- `noop_time`: 3.3888 seconds
- `op1_time`: 3.1599 seconds
- `h100_time`: 3.0561 seconds

Job	noop_time	op1_time	h100_time
1	3.39001	3.16644	3.05173
2	3.41524	3.14918	3.04954
3	3.37244	3.17189	3.06196
4	3.40483	3.13981	3.06987
5	3.36142	3.17196	3.04739

The average times are as follows:

- `noop_time`: 3.3888 seconds
- `op1_op2_time`: 2.0278 seconds
- `h100_time`: 3.0591 seconds

Job	noop_time	op1_op2_time	h100_time
1	3.39001	2.03812	3.05928
2	3.41524	2.01147	3.03081
3	3.37244	2.03197	3.05741
4	3.40483	2.01804	3.06021
5	3.36142	2.03962	3.08778