1. Algorithm of Program

Our program implements a brute-force search for viral DNA signatures within patient DNA samples using two CUDA kernels.

Preprocessing Kernel

- Convert Phred+33 encoded qualities into integer Phred scores and compute an integrity hash for each sample. With one thread per sample, each thread iterates over its sample's quality array in parallel (on the GPU), subtracts 33 from each character (except for 'N', which has a Phred score of 0)

Signature Matching Kernel

- The total number of (sample, signature) pairs are calculated as the number of samples multiplied with the number of signatures.

- For each (sample, signature) pair, find the best contiguous match (if it exists) and compute an average Phred score as the match confidence. One thread processes per (sample, signature) pair. Each thread retrieves the sample's and signature's information and starts the naive matching algorithm on the GPU and writes out the resulting confidence.

The program processes two main tasks on the GPU. For both tasks, it uses a commonly optimal configuration of 256 threads per block. 256 threads per block are typical for CUDA kernels to achieve good occupancy on a broad range of NVIDIA GPUs.

The grid sizes (number of blocks) are computed based on how many threads are needed for the total amount of work. This ensures full coverage while keeping the workload evenly distributed across the GPU.

We will be choosing a different configuration for an optimised parallelised strategy in our optimizations. Additionally, the program does not use shared memory but we will be implementing that in our optimization.

2. How Input Factors Parameters Affect Runtime

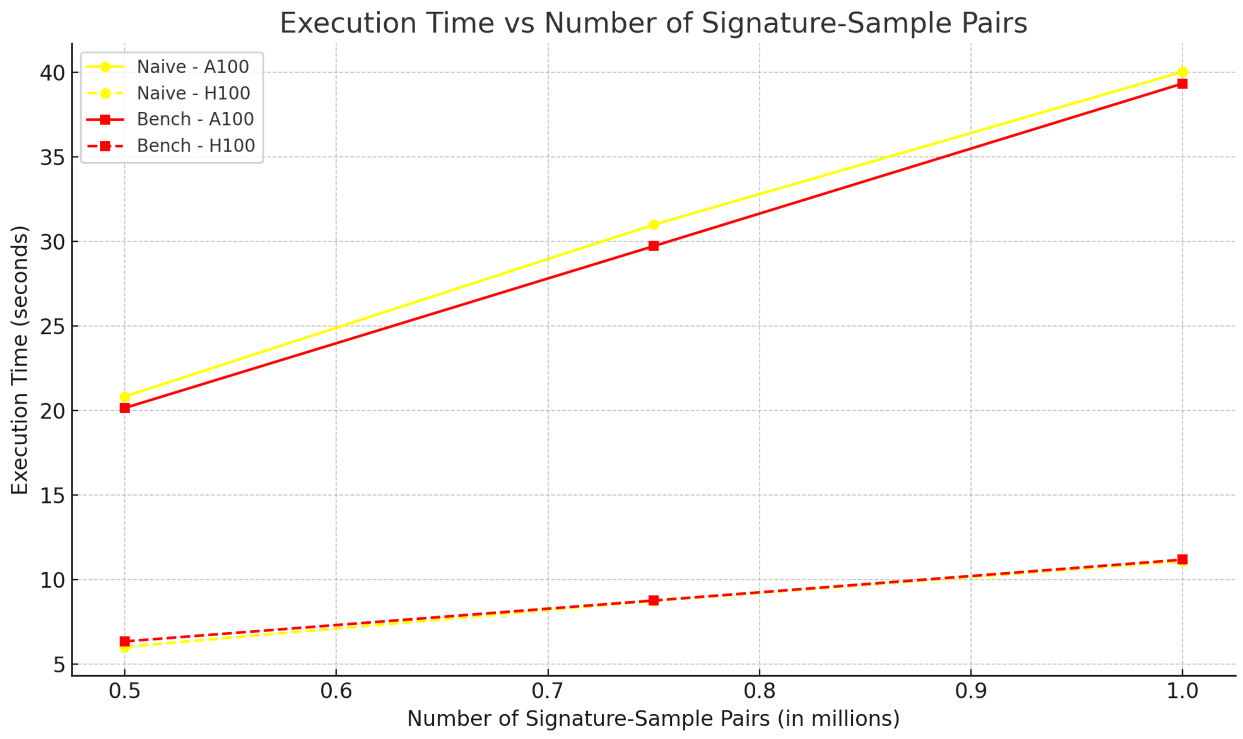
We can adjust numerous variables to explore the program's performance, but we've decided to focus on three key aspects:

1. The total number of (sample, signature) pairs.
2. The percentage of nucleotide 'N' within each signature.
3. The length of the signature.

We consider a single (sample, signature) pair as the basic unit of comparison. Increasing either the number of signatures or the number of samples effectively increases the total count of these pairs.

We keep the signature set constant at 500 Signatures to minimize variability while varying the number of samples.

This setup yields 500,000 to 1,000,000 total (sample, signature) pairs in the graph.

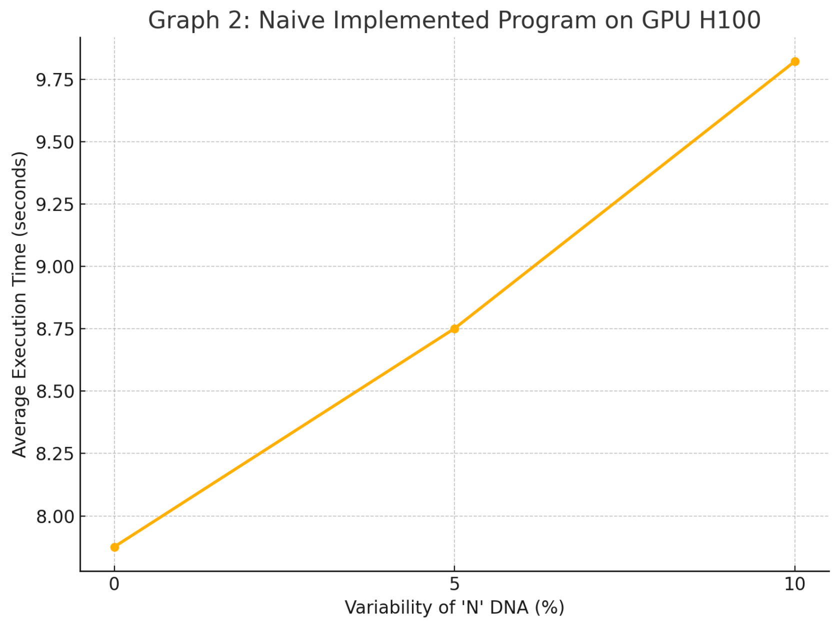


While our naive program is straightforward, this approach is computationally intensive and scales poorly with larger datasets. It performs poorer than the benchmark. The increasing trend in execution time with a higher number of signature-sample pairs is expected, as the computational workload grows linearly with the number of comparisons. Each additional pair introduces more matching operations.

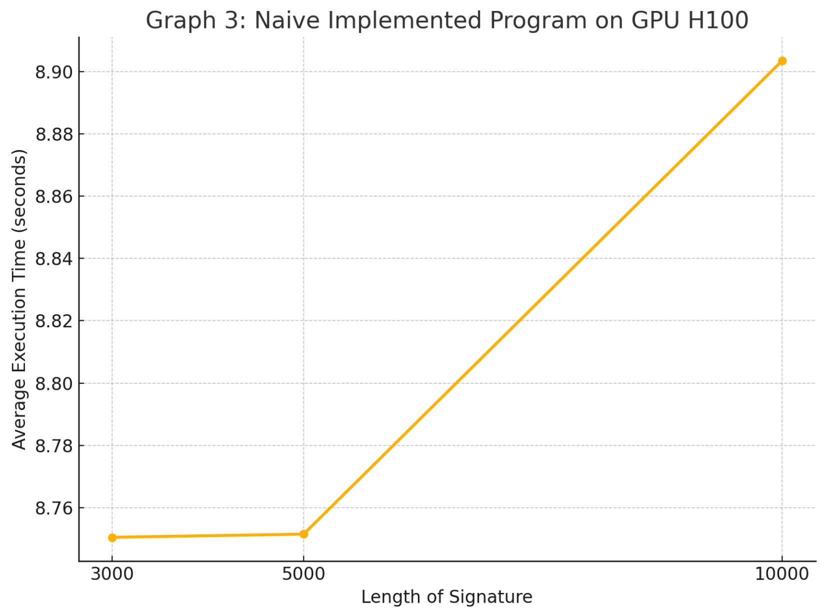
For this graph, we use:

- 500 signatures (each 4,000–5,000 nucleotides long)
- A maximum ‘N’ nucleotide percentage of 0.05
- Three different sample set sizes (1,000, 1,500, and 2,000 as this is within range of input size), split evenly between infected and non-infected DNA

For brevity, the below graphs are programs run on 1 GPU showing a clear trend in the expected outcome of increasing Wildcard Variability and Signature Length.



The increasing execution time with higher 'N' DNA variability is due to the naive algorithm allowing `'N'` to act as a wildcard during comparisons. While this increases the chance of matches, it also forces the program to perform additional checks for ambiguity at each base, making the match logic more complex and resulting in longer execution times.



For increasing signature length, the execution time grows because the naive algorithm performs a linear scan over the sample for each signature, comparing character-by-character. Longer signatures mean more characters need to be compared at every position, increasing the computational workload per match. This overhead becomes especially prominent as the signature length approaches large values (e.g. 10,000 bases), which leads to a sharp rise in runtime.

These observations are all expected.

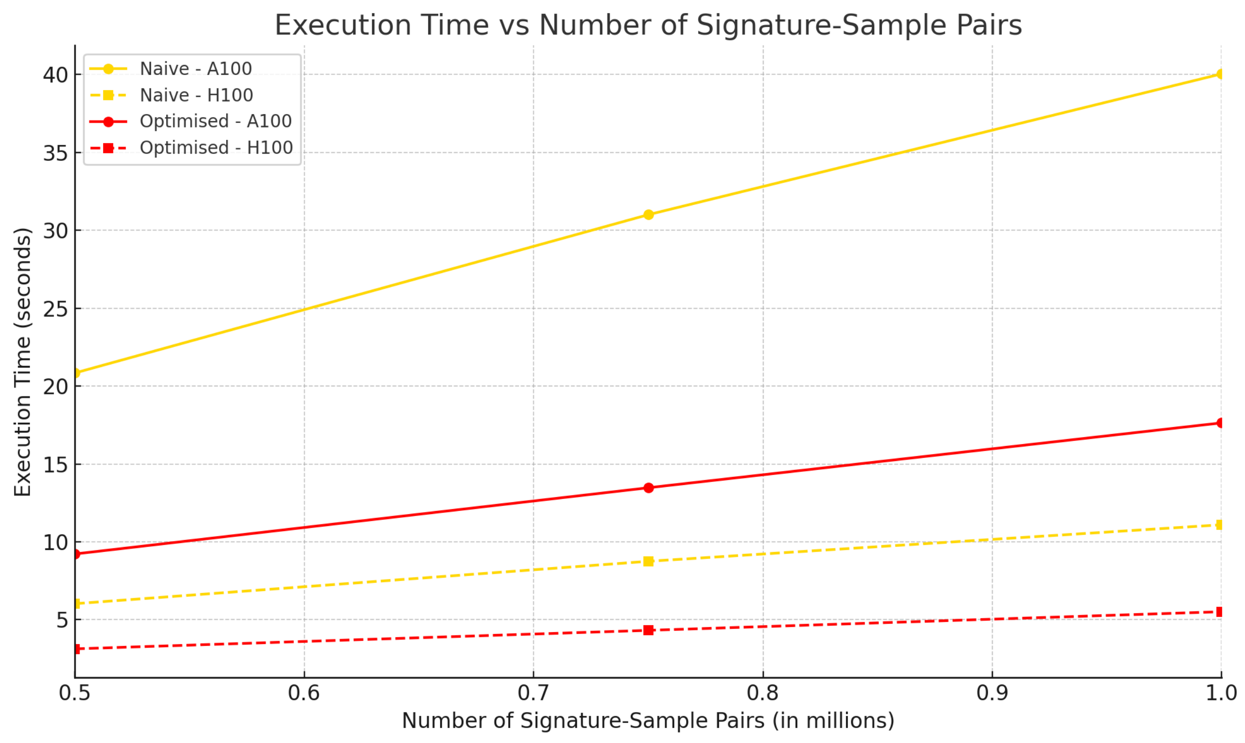
3. First Optimization: Single-Block per (Sample, Signature) Pair + Shared Memory Reduction

Originally, the naive matching kernel launched one thread per (sample, signature) pair and naively processed all positions within that single thread. The entire signature matching loop (substring comparisons, confidence scoring, etc.) was done serially inside a single thread which underutilized GPU resources.

We replaced it with a single-block per (sample, signature) pair strategy which details:

1. Each block corresponds to a single (sample, signature) pair.
2. Within the block, multiple threads stride over potential match-start positions, thus parallelizing the search.
3. We collect the maximum confidence score in shared memory. A final reduction stage merges all partial results within the block, which leverages on fast block-level operations.

Results



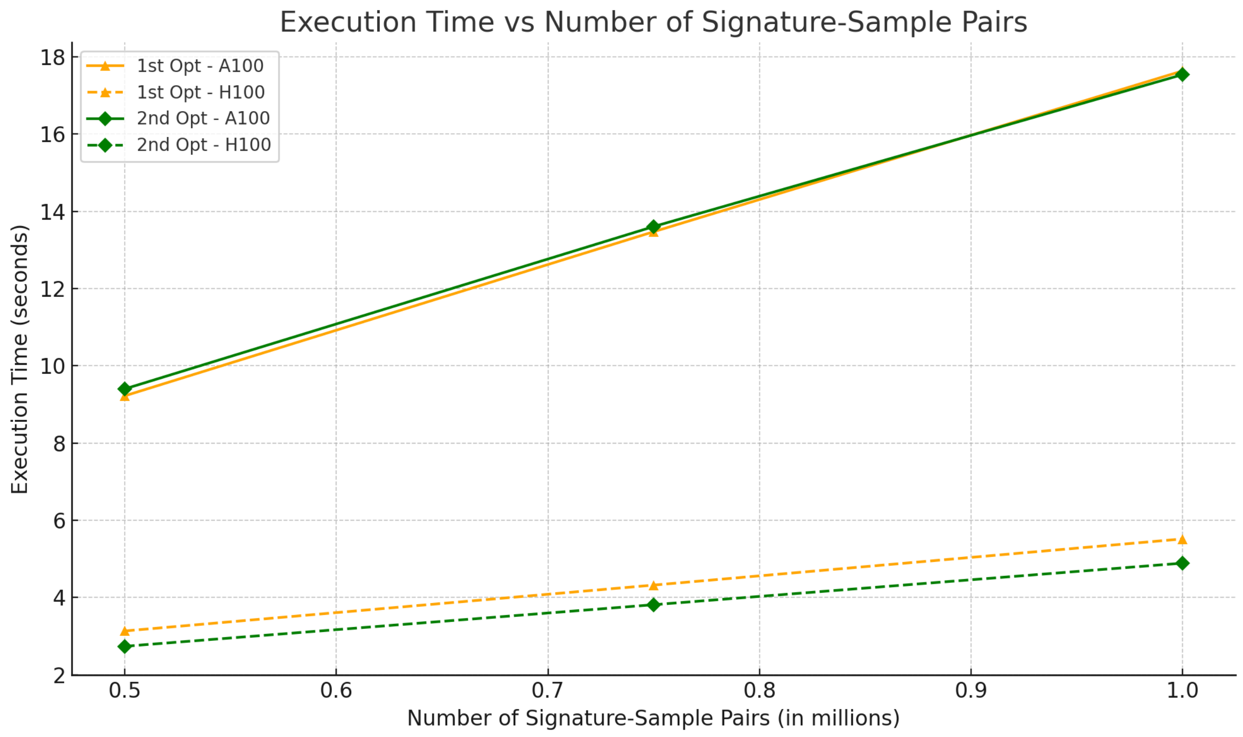
The results show that the optimised program improved by almost a magnitude of 2.

3. Second Optimization: Host-to-Device Memory Copy

Next we addressed a bottleneck caused by host-to-device memory transfers. The original implementation allocated and copied memory individually for each sequence and signature, resulting in a large number of small `cudaMalloc` and `cudaMemcpy` operations that introduced significant overhead.

The optimized version reduces this overhead by aggregating all sequence, quality, and score data into a few large contiguous buffers, allocating memory only once per data type and copying in bulk. Each data structure then references its specific region within these buffers. This significantly reduces the number of GPU memory operations, improves memory access patterns, simplifies memory management, and enhances overall execution efficiency.

Results



The results show that the 2nd optimization helped the program perform better on the H100 GPU, consistently achieving lower execution times across all input sizes. However, on the A100, both optimisations perform similarly, with the 1st optimisation slightly outperforming the 2nd, indicating that the enhancements in the 2nd version do not translate as effectively.

Conclusion

The 1st and 2nd optimisation programs significantly reduce execution time compared to the naive baseline, especially on the H100 GPU. Overall, both optimisations scale well, with the H100 delivering over 2× speedup compared to A100 across all input sizes.

Appendix

Reproducing samples and signature

For samples and signature used for input variability

Variability of N

Signatures

```
./gen_sig 500 4000 5000 0 > fasta_files/nlow_sig.fasta
./gen_sig 500 4000 5000 0.05 > fasta_files/basic_sig.fasta
./gen_sig 500 4000 5000 0.1 > fasta_files/nhigh_sig.fasta
```

Samples

```
./gen_sample fasta_files/nlow_sig.fasta 750 750 2 3 1000000 1500000 10 30 0 >
fastq_files/nlow_med.fastq

./gen_sample fasta_files/basic_sig.fasta 750 750 2 3 1000000 1500000 10 30 0.05 >
fastq_files/basic_med.fastq

./gen_sample fasta_files/nhigh_sig.fasta 750 750 2 3 1000000 1500000 10 30 0.1 >
fastq_files/nhigh_med.fastq
```

Variability of signature len

Signatures

```
./gen_sig 500 3000 3000 0.05 > fasta_files/lenlow_sig.fasta
./gen_sig 500 4000 5000 0.05 > fasta_files/basic_sig.fasta
./gen_sig 500 10000 10000 0.05 > fasta_files/lenhigh_sig.fasta
```

Samples

```
./gen_sample fasta_files/lenlow_sig.fasta 750 750 3 5 1000000 1500000 10 30 0.05 >
fastq_files/lenlow_med.fastq
```

```
./gen_sample fasta_files/basic_sig.fasta 750 750 2 3 1000000 1500000 10 30 0.05 >
fastq_files/basic_med.fastq
```

```
./gen_sample fasta_files/lenhigh_sig.fasta 750 750 1 1 1000000 1500000 10 30 0.05 >
fastq_files/lenhigh_med.fastq
```

For samples and signature used for optimization graphs:

Signatures

```
./gen_sig 500 4000 5000 0.05 > fasta_files/basic_sig.fasta
```

Samples

```
./gen_sample fasta_files/basic_sig.fasta 500 500 2 3 1000000 1500000 10 30 0.05 >
fastq_files/basic_smol.fastq
```

```
./gen_sample fasta_files/basic_sig.fasta 750 750 2 3 1000000 1500000 10 30 0.05 >
fastq_files/basic_med.fastq
```

```
./gen_sample fasta_files/basic_sig.fasta 1000 1000 2 3 1000000 1500000 10 30 0.05 >
fastq_files/basic_big.fastq
```

Assignment 2 Raw data:
https://docs.google.com/spreadsheets/d/1Ky_sVwPtRAaLDECkCtFLGRRgXtsE69DHyFhmI0SiRpQ/edit?usp=sharing