Virus Signature Scanning with CUDA

Implementation Overview

Algorithm

Our program implements a parallel matching algorithm for each sample-signature pair. The matching process is performed by sliding a window of the signature's length across the sample sequence. The program compares the signature's character for every position with the corresponding characters in the sample. If a mismatch is detected in any position, the current window is considered invalid and the window is moved to the next position of the sample. The kernel stops scanning once it finds a match between a sample and a signature as only the first match is considered.

Parallelisation Strategy

- We can parallelize this algorithm as each sample-signature combination is independent of the others. Thus, each thread in the CUDA kernel would handle one such combination pair.
  The work is divided by mapping each thread to a unique sample-signature combination. If there are *N* sample and *M* signatures, then there would be *N*M* comparisons, each handled by a separate thread.

  The total number of threads will thus be the total number of sample-signature pairs. We would be using a single kernel. Each block would be run using 512 threads, and the grid size (number of blocks) is determined by dividing the total number of comparisons by the number of threads per block, ensuring every thread gets a unique pair of samples and signatures to compare.

Choice and justifications for the dimensions

Modern NVIDIA GPUs typically support up to 1024 threads per block. We need to strike a good balance of using the optimal number of threads per block. If we use too few threads, the GPU will have low occupancy and sit idle. Conversely, running too many threads might lead to resource limitations and contention.

Therefore, we have tried running different amounts of threads per block to compare performance. After testing varying threads per block number, with large multiples of 32 as a

warp consists of 32 threads, we found that 256 threads per block optimizes occupancy without hitting resource limits for this program (table 1).

Memory Handling

We allocated memory for sequences, offsets (which store the start positions of each sequence), quality scores, and results (whether a match was found and the corresponding score) in the GPU.

Global memory is used for storing both input data and results. Shared memory is not utilized in this specific kernel since there are no frequent data-sharing requirements between threads, and the access pattern is mainly global.

Impact of Input Factors on Runtime & Explanations

Sequence Lengths:

- As the length of the sample or signature sequence increases, the runtime also increases. This is because each thread must slide the signature over a longer sample, checking for matches at every possible position. The number of computations per thread scales linearly with the sequence length, so the runtime scales accordingly.

- The longer the sequences, the more positions there are to check in the sample sequence. This increases the number of character comparisons per thread. Runtime scales with the amount of work each thread performs. When threads are given longer sequences to compare, they require more time to finish their task. The number of computations per thread scales linearly with sequence length, so if the sample length doubles, the runtime for that thread approximately doubles as well.

Number of Samples and Signatures:

- Runtime increases as the number of samples and signatures increases. Even though many threads are executed in parallel, the GPU has a limited number of threads that can run concurrently. If the number of threads required exceeds the number of available cores on the GPU, threads must be executed in multiple "batches" or warps. This causes some threads to wait until others finish, leading to a bottleneck and increasing the total runtime.

- There will also be the overhead associated with launching and managing the memory access and synchronization of threads. Therefore, even though each comparison is independent, increasing the number of samples or signatures increases the workload linearly.

Percentage of N Characters:

- A higher percentage of 'N' characters can decrease runtime. When either the sample or the signature has an 'N' character at a given position, the algorithm skips the comparison for that character and moves on to the next one. Fewer actual comparisons are needed, allowing threads to finish their work faster.

- By having ‘N’ characters, the most time-consuming part of comparing and determining if individual characters match is skipped, effectively reducing the number of meaning comparisons that each thread must perform. For example, if half of a signature consists of ‘N’s, only half as many character comparisons are made.

Percentage of Matching Sequences:

- An increase in the percentage of matching sequences will reduce the runtime. If a large proportion of the sample-signature pairs match, the program can exit early from the comparison loop for those threads, reducing the amount of work each thread performs. On the other hand, if there are few matches, the threads must fully scan the sample for every signature, increasing runtime.

- According to our algorithm, a thread terminates and avoids further comparisons when a match is found. When there are many matching sequences, there is a much higher likelihood of finding a match early and exiting, leading to a reduction in overall runtime.

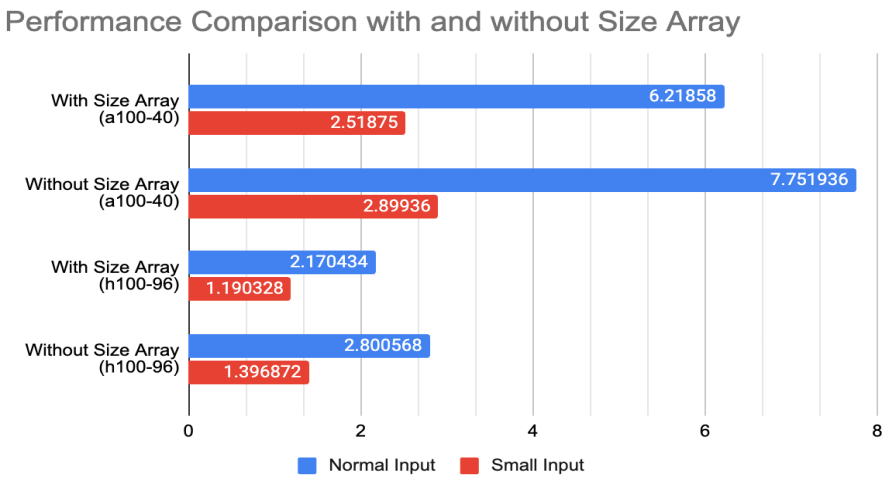
Performance optimisations

1. Passing in an additional size array into GPU

Initial solution:
Initially, we passed in a flattened string of sequences and an array of offsets that showed the starting position of each sequence. This meant that each thread would have to use the offset array to dynamically calculate the position and length of the sample and signature sequences it was operating on.

Optimized Solution:
We introduced an additional array containing each sequence's sizes in memory. Thus, we were able to perform computationally expensive arithmetic operations within the GPU. The threads can directly access the pre-calculated values from the arrays inside the kernel, significantly improving performance.

Performance Gains:

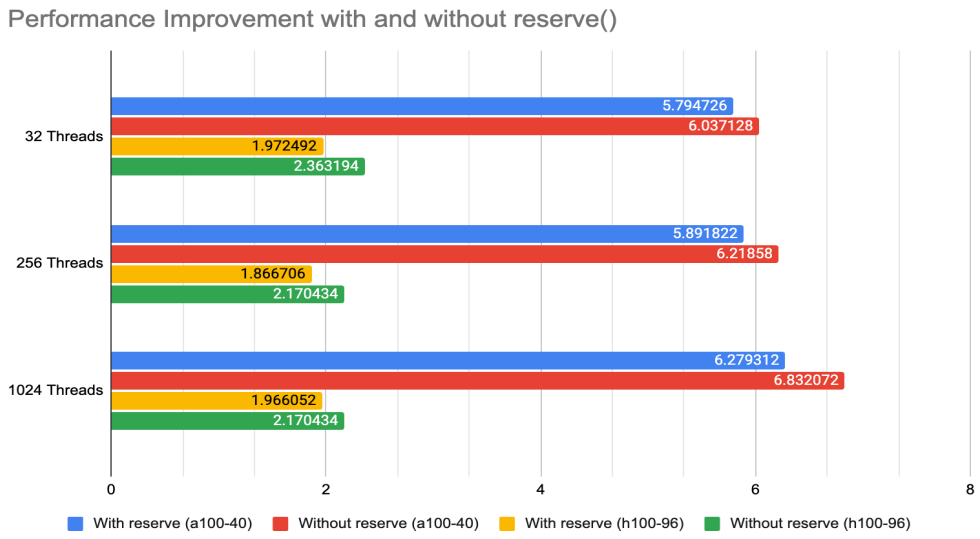


2. Reserving String Length

Initial Problem:
The strings used to store flattened virus signatures and sample sequences were dynamically resized as data was appended to them, causing frequent memory reallocations, which involved copying existing data to a new location. This process is both computationally expensive and inefficient.

Optimized Solution:
We pre-calculated the total size of both sequence signatures and sample sequences. By using the *reserve()* function to allocate memory upfront, we prevented multiple reallocations, significantly improving performance.

Performance Gains:



Appendix

Reproduction of results (input, execution time measurement)

Creation of *sig.fasta* and *samp.fastq* files for input:

- *./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*

Commands to run the executable on SoC Computer Cluster*:*

- *srun --ntasks 1 --cpus-per-task 1 --cpu-bind=core --mem 20G --gpus a100-40 --constraint xgph ./matcher samp.fastq sig.fasta*
- *srun --ntasks 1 --cpus-per-task 1 --cpu-bind=core --mem 20G --gpus h100-96 --constraint xgpi ./matcher samp.fastq sig.fasta*

Commands for Profiling with nsys Profile:

- *srun --ntasks 1 --cpus-per-task 1 --cpu-bind=core --mem 20G --gpus a100-40 --constraint xgph nsys profile ./matcher samp.fastq sig.fasta*
- *srun --ntasks 1 --cpus-per-task 1 --cpu-bind=core --mem 20G --gpus h100-96 --constraint xgph nsys profile ./matcher samp.fastq sig.fasta*

Nodes used for testing and performance measurement

- A single a100-40 MIG GPU (xgph node)
- A single h100-96 GPU (xgpi node)

Relevant performance measurement

Nsys Report:
From NVIDIA Nsight System UI, using report generated by nsys Profile:

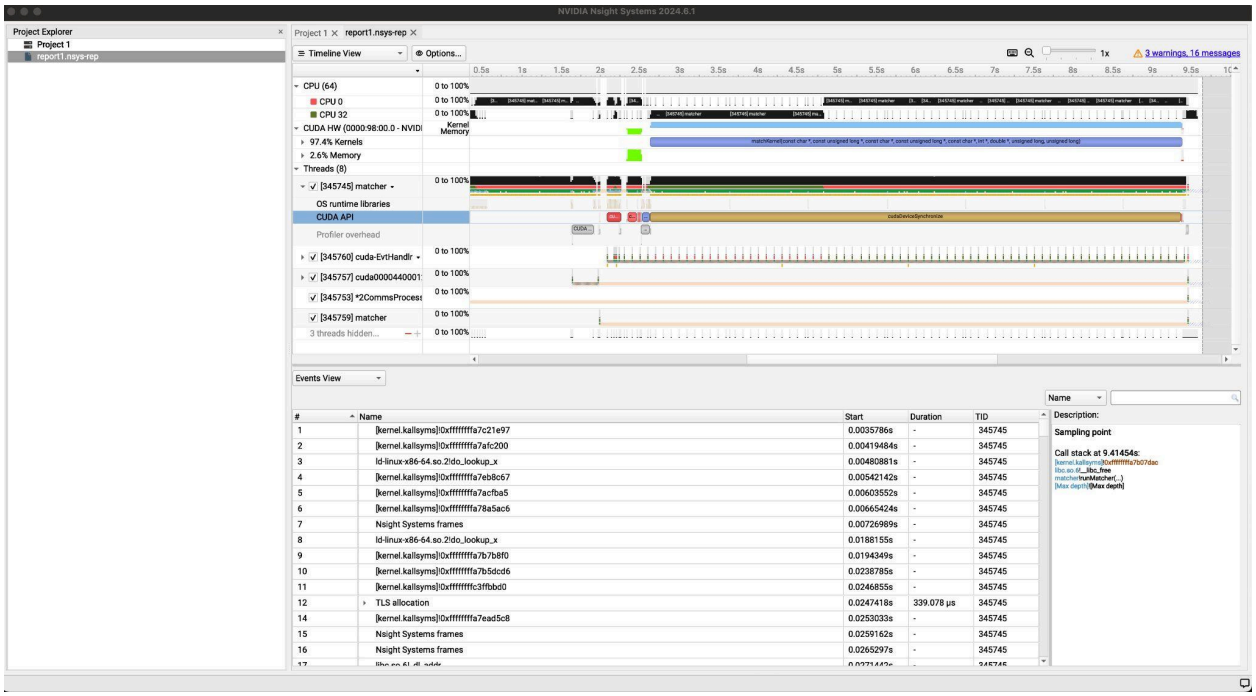


Table 1: Profiling for the number of threads per block:

All data in the table is an average of 5 runs, with the raw data in the provided excel sheet.

Number of threads per block	a100-40 MIG GPU	h100-96 GPU
64	5.671558	1.899518
128	5.624732	1.78209
256	5.605666	1.758896
512	5.80848	1.812912

Excel sheet with raw data:

Excel Sheet with Raw Data