CS3210 Assignment 2 Report

Team: `a2-e1297741-e1297745` **Authors:** Ding Zhe, Sun Weiyang

1. Algorithm Description and Parallelization Strategy

This section details the algorithm, parallelization strategy, and memory management techniques used to implement the virus signature scanner in CUDA.

1.1. Algorithm

The core of our solution is a **brute-force, sliding-window algorithm**, which is highly parallelizable and well-suited for the GPU architecture. The overall process is broken down into three distinct computational stages, each implemented as a separate CUDA kernel:

1. **Phred Score Conversion (`get_phred` kernel):** The input quality scores, which are in Phred+33 ASCII format, are converted into their numerical `unsigned char` Phred score equivalents by subtracting 33.
2. **Integrity Hash Calculation (`get_hash` kernel):** For each patient sample, an integrity hash is computed by summing all of its Phred scores and taking the result modulo 97. This is done for data verification purposes.
3. **Signature Matching (`solve_matcher` kernel):** This kernel performs the primary matching logic. For every possible sample-signature pair, the signature is treated as a sliding window across the sample. At each position, it is checked if the signature is a contiguous substring of the sample, accounting for 'N' as a wildcard character. For every valid match, a match confidence score is calculated as the average of the Phred scores in the matched sample segment.

1.2. Parallelization Strategy

The workload is parallelized across the three kernels, which are launched sequentially on a single CUDA stream to enable asynchronous execution.

- **`get_phred` Kernel:** This kernel exhibits simple data parallelism. The entire set of quality scores for a batch is treated as a single large array. The kernel is launched as a 1D grid, and a **grid-stride loop** is employed. This allows a fixed number of threads to process an array of any size, ensuring scalability. Each thread is independent and converts multiple quality characters to Phred scores.
- **`get_hash` Kernel:** This kernel is parallelized by assigning **one CUDA block to compute the hash for one entire sample**. Within each block, threads work cooperatively to calculate the sum of Phred scores using a highly efficient **parallel reduction** algorithm that leverages shared memory.

- **`solve_matcher` Kernel:** This is the most computationally intensive part. Parallelism is achieved at two levels:
    - **Block-Level (Task Parallelism):** The problem is mapped to a 2D grid of blocks, where each block is responsible for matching **one unique sample-signature pair**.
    - **Thread-Level (Data Parallelism):** Within each block, the threads work together to perform the sliding window search. The search space (the sample sequence) is divided among the threads, with each thread checking a different subset of starting positions. A final parallel reduction is performed to find the single match with the highest confidence score for that block's sample-signature pair.

1.3. Grid and Block Dimensions

The choice of grid and block dimensions is critical for performance. Our choices are as follows:

- **Block Size:** For all kernels, a block size of **256 threads** (`blk_size = 256`) was chosen. This is a multiple of the warp size (32), which is crucial for execution efficiency on the GPU. 256 provides a good balance between exposing enough parallelism to hide memory latency and not using so many resources (e.g., registers) per block that it limits the total number of concurrent blocks (occupancy).
- **Grid Size:**
    - `get_phred`: The grid size is calculated as `(total_samples_len + blk_size - 1) / blk_size` to ensure enough threads are launched to cover all quality scores.
    - `get_hash`: The grid size is simply the number of samples in the batch, as each block handles one sample.
    - `solve_matcher`: A 2D grid of `(number_of_samples, number_of_signatures)` is launched, mapping directly to the problem space.

1.4. Memory Handling

Efficient memory management is key to performance. Our strategy focuses on minimizing data transfers and using the fastest memory spaces available.

- **Asynchronous Transfers:** All data transfers between the host and device (`cudaMemcpyAsync`) and all kernel launches are executed on a single CUDA stream. This allows the GPU to overlap computation with data transfers, hiding the latency of moving data across the PCIe bus.
- **Pinned Host Memory:** Host-side memory for data buffers is allocated using `cudaHostAlloc`. This pins the memory, allowing the GPU's DMA engine to transfer it directly without CPU intervention, resulting in significantly higher transfer bandwidth.
- **Data Batching:** To handle massive sample files without exceeding the GPU's VRAM capacity, samples are processed in batches of 1024. Data for each batch is transferred, processed, and the results are returned before moving to the next batch.

- **Shared Memory Usage:** Shared memory, an extremely fast on-chip memory, is used critically in two kernels:
    1. In `get_hash`, it stores intermediate sums during the parallel reduction, avoiding slow writes to global memory.
    2. In `solve_matcher`, it serves two purposes: first, as a **user-managed cache for the signature sequence**, which all threads in the block read from repeatedly; second, to store intermediate maximum scores during the final parallel reduction for the match confidence.

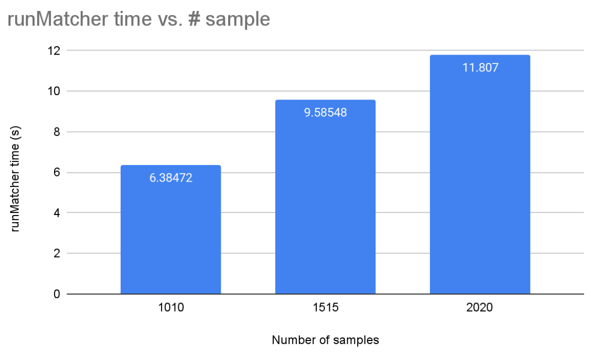
---

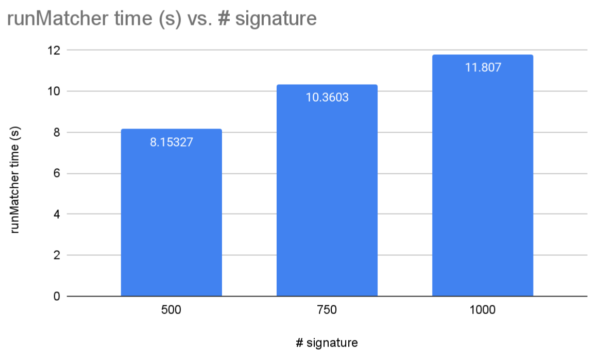
2. Performance Analysis

2.1. Effect of Input Factors on Runtime

The overall runtime is most heavily influenced by the parameters that define the search space of the `solve_matcher` kernel.

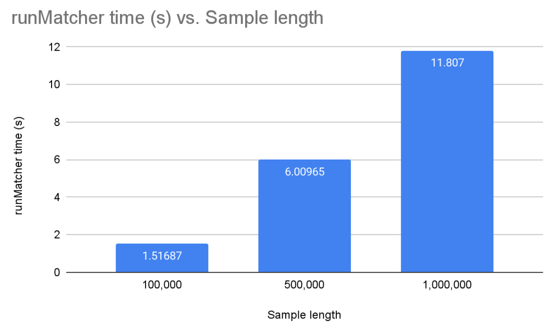
- **Number of Samples and Signatures:** The total work is directly proportional to the product of the number of samples and signatures, as each pair must be checked. Therefore, the relationship between runtime and these counts is **linear**. Doubling the number of samples while keeping other factors constant will roughly double the execution time, as the GPU simply has twice as many independent tasks to perform.
- Dataset: Tests used **1010, 1515 and 2020 samples** of length 1000k, and **500, 750 and 1000 signatures** of length 6k generated with the provided helper scripts on **NVIDIA H100-96** GPU.

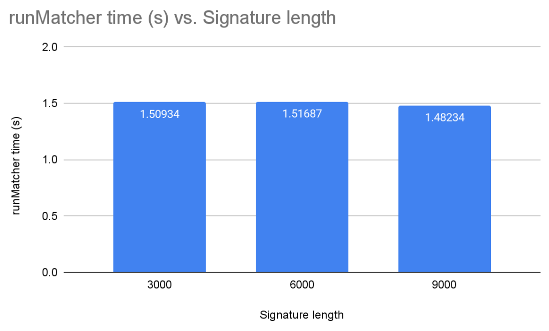




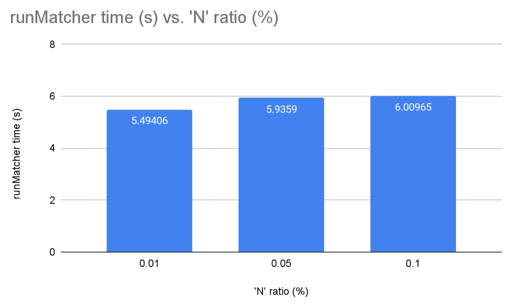
- **Sequence Lengths:** The runtime of the `solve_matcher` kernel for a single sample-signature pair is approximately proportional to `O(Sample Length)`. As the sample length increases, the search space for the sliding window grows **linearly**. However, as the signature length increases, the runtime **remains roughly the same**. This is because most of the time, for the inner loop, it would be a mismatch. So there would usually be a `break` from this loop after several comparisons. As a result, the work done at each window position (the inner comparison loop) doesn't change in amortized analysis. This combined effect makes the sequence length of samples, but not signatures, a dominant factor in performance.

- Dataset: Tests used 2020 **samples of length 100k, 500k and 1000k**, and 1000 **signatures of length 3k, 6k and 9k** generated with the provided helper scripts on **NVIDIA H100-96** GPU





- **Percentage of 'N' Characters:** The effect of 'N' characters is nuanced. Since 'N' matches any character, it reduces the probability of an early mismatch in the inner comparison loop of `solve_matcher`. A `break` from this loop is a performance optimization, as it allows the thread to skip the expensive confidence score calculation. Therefore, a higher percentage of 'N's leads to more full comparisons and score calculations, slightly **increasing the average workload per thread** and thus marginally increasing the overall runtime.
- Dataset: Tests used 2020 samples of length 500k **with N ratio 0.01, 0.05 and 0.1**, and 1000 signatures of length 6k **with N ratio 0.01, 0.05 and 0.1,** generated with the provided helper scripts on **NVIDIA H100-96** GPU



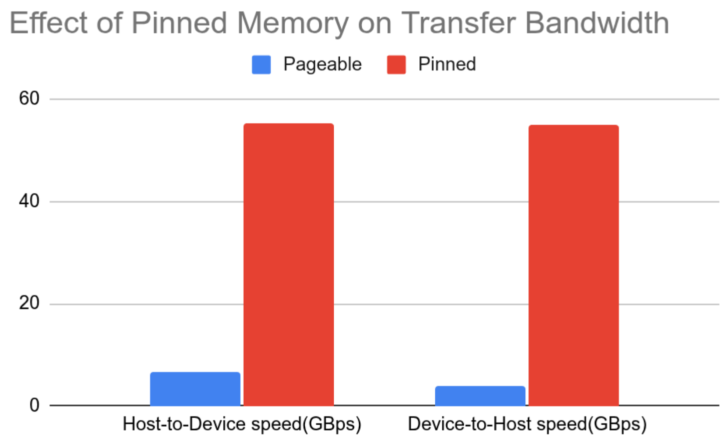
3. Performance Optimizations

3.1 Optimisation A — Using Pinned Memory

We initially tried asynchronous allocation (`cudaMallocAsync`) to reduce host–device transfer latency, but the overall runtime improvement was negligible.

Further inspection revealed the main bottleneck came from using **pageable host memory** (`std::vector`), which forces CUDA to copy data into a temporary pinned buffer before each DMA transfer. We therefore replaced all host arrays with **page-locked (pinned) memory** allocated using `cudaHostAlloc`, enabling direct GPU access and truly asynchronous data transfers.

Dataset: Tests used approximately **2000 samples** (100–200 KB each) and **1000 signatures** (3–10 KB each) generated with the provided helper scripts on **NVIDIA H100-96** GPU.



The transfer bandwidth increased by nearly **8×**, confirming that **pinned memory** was the true enabler of high-throughput asynchronous transfer.

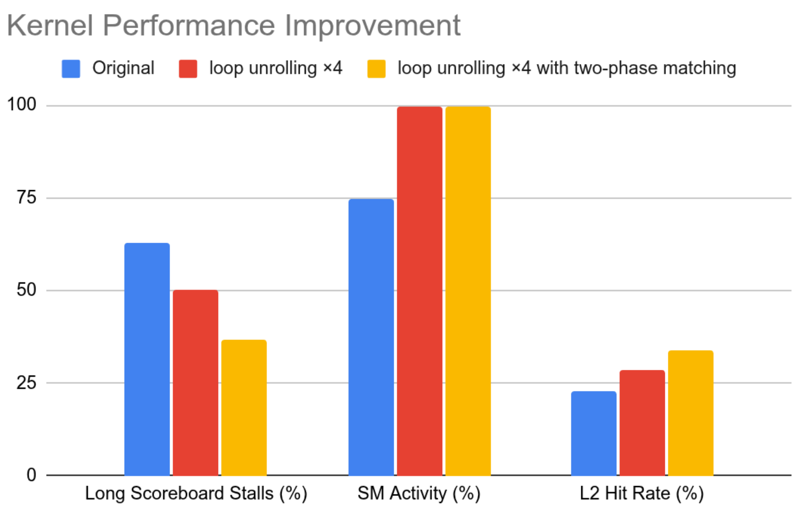
3.2 Optimisation B — Loop Unrolling and Two-Phase Matching

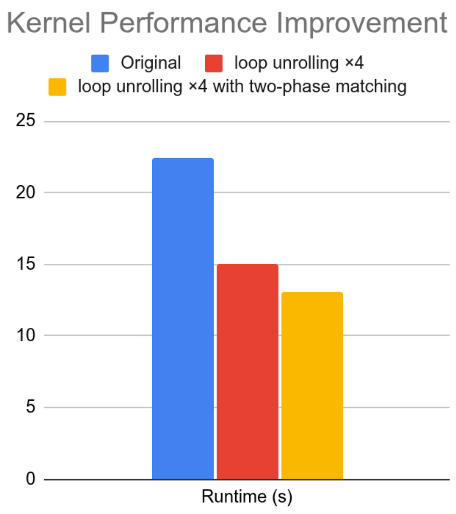
The original kernel exhibited severe long scoreboard stalls (**62.9%**), showing that warps frequently waited for dependent global-memory loads. We first applied manual **loop unrolling × 4**, which reduced instruction dependency chains and allowed better **instruction-level parallelism**. This lowered stalls to **50.1%** and improved runtime from 22.48s to 15.05s.

We then extended the approach into a **two-phase matching** design, separating the character-comparison and Phred-score accumulation loops. This reduced warp divergence and improved cache locality. As a result, long scoreboard stalls further dropped to **36.6%**, while SM activity rose from **74.7%** to **99.6%**. The kernel runtime decreased to **13.04s**.

Memory metrics also confirmed the effect: **L2 hit rate** increased from 22.7 % to 33.9 %, and **DRAM traffic** (`dram__bytes.sum`) decreased from 245.1 GB to 101.8 GB, indicating fewer redundant global loads and higher cache reuse.

Dataset: Tests used approximately **2000 samples** (100–**2000** KB each) and **1000 signatures** (3–10 KB each) generated with the provided helper scripts on **NVIDIA H100-96** GPU.





Appendix

A.1. How to Reproduce Results

The results can be reproduced on the SoC Compute Cluster using the provided `Makefile`.

1. **Environment:** The code was compiled and run on the `xgph` (NVIDIA A100) and `xgpi` (NVIDIA H100) nodes.
2. **Compilation:** Navigate to the root directory and run `make`. This will produce the `matcher` executable.
3. **Execution:** Run the program using a `srun` command with the specified resource constraints, for example: `srun --ntasks 1 --cpus-per-task 1 --cpu_bind core --mem 20G --gpus h100-96 --constraint xgpi ./matcher samples.fastq signatures.fasta`
4. **Test Data:** The input files used for the graphs in Section 3 were generated using the provided `gen_sample` and `gen_sig` utilities.

A.3. Supporting Measurements

All raw data collected and used is available in CS3210_Assignment2_Raw_Data. The script to generate data used by section 3 is in

A.3. Acknowledgement

We used ChatGPT to help generate the outline of this report.