GPU acceleration of Smith Waterman and PairHMM algorithms

Antonio Marusic

June 30, 2024

Abstract

This report wants to investigate the behavior of two well-known algorithms in the field of sequence alignment, the Smith-Waterman and Pair Hidden Markov Models, when accelerated on GPU. Experiments on GPU acceleration have been performed on both programs using the state-of-the-art technique. Parameter tuning has been performed on different datasets.

1 Introduction

The Smith-Waterman and Pair Hidden Markov Model (PHMM or PairHMM) algorithms are well-known methods for performing local sequence alignment. Those are widely used in bioinformatics to identify regions of similarity between biological sequences, such as proteins or DNA. However, computational complexity can be a significant bottleneck for both algorithms when working with large datasets. This report investigates the potential of GPU acceleration to enhance the performance of the Smith-Waterman algorithm without considering the backtrace part and regular PairHMM algorithms.

This paper first explores the domain of sequence alignment and then overviews the CUDA programming model2. A part of the state of the art is then covered3. Later, both C implementations of the algorithms are explained4, and then the idea of the GPU acceleration is commented on4. The report concludes with an analysis of the results5.

2 Background

In this section I introduce the main concepts in local alignment and GPU programming, with a specific focus on CUDA's programming model.

2.1 Local alignment

Local alignment is a technique developed in the context of sequence alignment in the DNA domain. The method aims to identify regions of similarities between two sequences. These regions may represent evolutionary relationships. Conversely, global alignment considers the entire length of both sequences. Both local and global alignment methods work similarly; the first is particularly useful when comparing very different sequences that may contain only small regions of similarity.

The original Smith-Waterman algorithm seeks the local alignment that yields the highest score based on a predefined scoring scheme. This scoring scheme typically assigns positive values for matches between corresponding nucleotides (e.g., A aligning with A), negative values for mismatches (e.g., A aligning with G), and penalties for introducing gaps (insertions or deletions) in one sequence to better align with the other. Gotoh extended in the 1980s the functionalities of the original algorithm by introducing the idea of affine gap penalty. While the original algorithm always assigns the same penalty to gaps, the modified one penalizes the initiation of a gap differently and the extension of an existing gap; in particular, extending a gap isn't as penalized as the beginning one. This approach better models the way mutations in biology happen; in fact, it is more probable that a single longer gap is inserted in a sequence rather than multiple smaller gaps. Moreover, it has been proven that the first amino acid or nucleotide inserted or deleted is more significant, from a biological point of view, than the subsequent ones.

Since extending gaps is less penalized, the algorithm discourages introducing unnecessary gaps in the alignment.

ParirHMM algorithm, on the other hand, uses Hidden Markov Models to give a similarity score based on probabilities. By incorporating states that model matches, mismatches, insertions, and deletions, pairHMM can effectively align two sequences while accounting for evolutionary events that may have caused divergence.

One could use variable scores for substitutions to further extend the algorithms in the case of protein alignment. The most used approach is the substitution matrix, a matrix of size $alphabet * alphabet$ that associates with each pair of letters a score built based on the probability that that particular letter is replaced with another during the evolutionary process. So, a positive value in the Substitutional Matrix means that the two letters are similar or identical and frequently exchanged without notable loss of biological function. Hernikoff, in 1992, introduced the Blosum Matrix, one of the most used Substitutional Matrix.

Both algorithm versions have a quadratic time complexity but can be parallelized on parallel architectures such as the GPU, thus significantly improving performances.

2.2 CUDA programming model

CUDA is the programming model used to parallelize this work. A programming model bridges an application and the underlying architecture that focuses on sharing information between the parts and coordinating activities. Moreover, it gives a logical view of specific computing architectures hiding too particular details, allowing the organization of threads and access memory.

In CUDA, a thread is the smallest unit of execution, is lightweight, and can be created in large numbers to take advantage of the parallel processing capabilities of a GPU. Each thread has its unique thread ID, which allows access to its registers and local memory. This ID lets the thread know its place within the block and grid. Threads are grouped into thread blocks, which can cooperate by sharing data through shared memory and synchronizing their execution. Blocks can be one-, two-, or three-dimensional. This flexibility allows you to match the organization of threads to the structure of the data being processed. The GPU architecture determines the maximum number of threads per block. For example, many modern GPUs allow up to 1024 threads per block. A grid is a collection of thread blocks that execute the same kernel, a function written in CUDA C/C++. Grids can also be one-, two-, or three-dimensional, providing an additional structure for organizing threads. CUDA allows the programmer to allocate shared and global memory space, launch multiple kernels, and decide the number of blocks and threads to assign to each. The kernels are asyncronally invoked by the CPU (host in CUDA terminology) so that the CPU can perform other actions during kernel execution. However, the CPU can also wait synchronously for the GPU (device in CUDA terminology) to terminate. Threads can synchronize with one another if they belong to the same block, and CUDA offers synchronize statements.

2.2.1 architectural overview

Streaming Multiprocessors are the fundamental processing units within a GPU. Each SM is assigned a run time and a subset of the scheduled blocks, and for each block, the threads inside it are grouped into warps, groups of 32 threads. An SM schedules and executes threads in warps. A warp is a group of 32 threads that execute instructions in a SIMT (Single instruction multiple thread) fashion, which means that within a warp, all threads perform the same instruction simultaneously but on different data. The SM contains a warp scheduler that selects which warp to execute next. The scheduler's decision is based on several factors:

- Resource Availability: The scheduler checks if the necessary resources (registers, shared memory, execution units) are available for the warp.
- Instruction Readiness: The scheduler ensures that the next instruction for the warp is ready to be executed. If the instruction is waiting for data from memory, the warp may be stalled, and the scheduler will choose another warp that is ready to proceed.

The memory model in CUDA includes different areas that can be directly programmed (this is a different approach from the CPU model that hides all this complexity). The areas are

- Shared memory
- Local memory
- Constant memory
- Texture memory
- Global memory
- Registers

Physically, the GPU has different types of memory; only three of those are programmable: the device memory, the shared memory, and the registers.

- device memory: highest latency and lowest bandwidth of the three but has the highest capacity. It is divided into areas that host local, constant, texture, and global memory. This memory is accessible to all threads through the kernel execution.
- L2 cache: a nonprogrammable area that hosts the L2 cache.
- L1 cache and shared memory: is a memory area local to each streaming multiprocessor and is used like L1 caches are used in CPUs but conversely to them has a programmable area called shared memory. It offers high bandwidth and low latency due to the proximity of the processing cores. Shared memory is partitioned across the blocks scheduled in the multiprocessor. Higher shared memory occupancy means fewer blocks can be scheduled on that streaming multiprocessor, reducing parallelism.
- registers: lowest latency of the three but are in a minimal number.

Registers are private to each thread and can be used to store local thread variables. Registers are a limited resource and, thus, are partitioned across the active warps in the streaming multiprocessor. If the kernel uses few registers, the scheduler can assign more thread blocks to the same streaming multiprocessor, increasing occupancy. And thus performance. If the number of registers is exceeded, the overflowing memory is allocated to local memory (thus storing data in a lower-performance memory area).

2.2.2 Fundamental concepts for performances

- Occupancy: This refers to the ratio of active warps to the maximum number of warps an SM can support. High occupancy can help hide memory latency by ensuring there are always warps ready to execute while others are waiting for memory operations to complete.
- Divergence: If threads within a warp take different execution paths (e.g., due to an if statement), the warp must serialize the different paths, which can reduce efficiency. Minimizing divergence is key to maintaining high performance.

3 State of the art

The CUDASW++3.0 paper [6] explores acceleration of the Smith-Waterman algorithm through CPU and GPU SIMD (Single Instruction, Multiple Data) instructions, particularly CPU computation, are carried out using vector instructions, while GPU uses the CUDA programming model specifically on Kepler architecture.

The algorithm they present distributes the workload over potentially multiple CPUs and GPUs based on an empirical formula they developed based on empirical constants and factors like core frequency, the number of cores in the CPU case, and streaming multiprocessors in the GPU.

Then, it computes the alignments using the GPU to fill the matrixes and the CPU for the other tasks. It also tries first to compute the alignments with lower precision, and if they exceed the accuracy, they increase the precision.

The ADEPT (A Domain-Independent strategy for GPU-accelerated sequence alignment) paper [2] proposes a novel technique to accelerate sequence alignment that can be domain-independent. The paper has a different approach concerning others that optimize their implementations for specific sequences or use cases. Results demonstrated that ADEPT outperforms or closely matches existing methods, achieving a peak performance of 360 GCUPS for proteins and 497 GCUPS for DNA on a single GPU node. GCPUS (Giga Cell Updates Per Second) is a metric used to measure the performances of dynamic programming algorithms that rely on a dynamic programming matric, which measures the number of cell updates per second. ADEPT showed a node-to-node (meaning on the same machine) speedup of 10x for DNA alignments and 7x for protein alignments compared to CPU implementations. ADEPT supports multi-GPU systems, making it scalable across multiple GPUs. ADEPT was integrated into existing bioinformatics software pipelines, specifically MetaHipMer (a metagenome assembler) and PASTIS (a protein similarity graph construction pipeline).

An older paper from 2008 [7] describes a CUDA implementation of the Smith-Waterman algorithm. The implementation achieved over 3.5 GCUPS (Giga Cell Updates Per Second) on a workstation equipped with two GeForce 8800 GTX GPUs. This approach was tested exhaustively against SEARCH, BLAST, and other GPU implementations, showing superior performance.

Regarding pairHMM explicitly, the paper [3] tries to solve the problem of existing hardware accelerated versions of the algorithm that lack flexibility and are constrained by fixed input sizes, limiting their use in clinical practice. This methodology pre-computes required resources based on sequence length, dynamically allocating GPU shared memory for shorter sequences and global memory for longer ones, thus maximizing GPU efficiency and supporting varying alignment lengths. The new implementation outperforms the best current FPGA hardware solutions by up to 1.6× and achieves an 8154× speedup over the software baseline. The algorithm is parallelized using a wavefront-based computational pattern, similar to Smith-Waterman and Needleman-Wunsch algorithms. This pattern allows cells to be updated in parallel, mapping each thread to an element of the anti-diagonal.

In the development of my specific implementation I relied on the above papers, in particular I got the formulation of the PairHMM algorithm from the [3] paper, the idea for the GPU acceleration from the [2] paper, and regarding Smith Waterman algorithm I relied on Freiburg website online implementation.

4 Implementation

In this section, I first explore the C implementation of the algorithms and then discuss the parallel version with the GPU that uses the same idea for both the algorithms under analysis.

4.1 Smith Waterman C version

The algorithm assigns a score to the pairwise alignment of two sequences. The algorithm has three matrixes (P, Q, D) containing partial alignment scores. In the classical implementation, only one matrix is used, allowing the model of just indels (insertions and deletions) and gaps to be modeled, assigning the same score to each gap. In the version I have implemented, the three matrixes allow assigning a different score to open a gap. The algorithm then employs dynamic programming, a powerful technique for efficiently solving problems by breaking them into smaller, overlapping subproblems. It takes in input the two strings to align and the scoring weights. This one uses recursion to fill the matrixes.

Input

- String s_1
- String s_2
- Gap initiation penalty α (negative number)
- Gap extension penalty β (negative number)
- Score for a match (positive number)
- Score for a mismatch (negative number)

Scoring Matrices

Three scoring matrices are used: D, P, and Q.

Matrix Definitions

- $D_{i,j}$: Represents the cost for the alignment of the prefixes up to that point (e.g., $(s_{1_0}, \ldots, s_{1_i})$, $(s_{2_0}, \ldots, s_{2_j})$)
- $P_{i,j}$: Represents the cost for the alignment of the prefixes up to that point that end with a gap in s_2

- $Q_{i,j}$: Represents the cost for the alignment of the prefixes up to that point that end with a gap in s_1

Initialization

- $P_{0,j} = -\infty$, with j ranging from 0 to the length of s_2
- $Q_{i,0} = -\infty$, with i ranging from 0 to the length of s_1
- $D_{0,j} = 0$, with j ranging from 0 to the length of s_2
- $D_{i,0} = 0$, with i ranging from 0 to the length of s_1

Iterative Filling of Matrices The gap penalty is computed with the following formula, where k is the size of the gap:

$$g(k) = \alpha + \beta k$$

Matrices:

$$\begin{aligned} P_{i,j} &= \max(D_{i-1,j} + g(1), \quad P_{i-1,j} + \beta) \\ Q_{i,j} &= \max(D_{i,j-1} + g(1), \quad Q_{i,j-1} + \beta) \\ D_{i,j} &= \max(D_{i-1,j-1} + \text{match score}, \quad P_{i,j}, \quad Q_{i,j}) \\ D_{i,j} &= \max(D_{i-1,j-1} + \text{mismatch score}, \quad P_{i,j}, \quad Q_{i,j}) \end{aligned} \tag{1}$$

The last two lines of the equation are used respectively in the case of matching nucleotides and in the case of mismatching nucleotides.

4.2 PairHMM C implementation

Also, this algorithm uses three matrixes and fills them with a recursive formula. Here, 4.2 is reported as the formula and the initialization constants. The input of this algorithm is organized in batches. Each batch contains a group of reads and probabilities encoded in ASCII and then a group of haplotypes (some sequences taken from a reference genome).

$$\begin{aligned} M[i][j] &= p(R[i-1], H[j-1], Q_r[i-1]) \cdot (\text{mm}(Q_i[i-1], Q_d[i-1]) \cdot M[i-1][j-1] \\ &\quad + (1 - Q_g[i-1]) \cdot (X[i-1][j-1] + Y[i-1][j-1])) \\ X[i][j] &= M[i-1][j] \cdot Q_i[i-1] + X[i-1][j] \cdot Q_g[i-1] \\ Y[i][j] &= M[i][j-1] \cdot Q_d[i-1] + Y[i][j-1] \cdot Q_g[i-1] \end{aligned} \tag{2}$$

The constants used in the formulae above are explained here.

$$\begin{aligned} &\text{OFFSET} = 33 \\ &\quad Q_r[i] = 10^{-(r[i]-\text{OFFSET})\cdot 0.1} \\ &\quad Q_i[i] = 10^{-(\text{ins}[i]-\text{OFFSET})\cdot 0.1} \\ &\quad Q_d[i] = 10^{-(d[i]-\text{OFFSET})\cdot 0.1} \\ &\quad Q_g[i] = 10^{-(g[i]-\text{OFFSET})\cdot 0.1} \\ &\quad \text{mm} = 1 - (Q_i[i] + Q_d[i]) \\ &\quad p = \begin{cases} 1 - Q_r[i-1] & \text{if } R[i-1] = H[j-1] \text{ or } R[i-1] = N \text{ or } H[j-1] = N \\ Q_r[i-1] & \text{otherwise} \end{cases} \end{aligned} \tag{3}$$

r, ins, d, and g are arrays of ASCII symbols generated by the sequencing machinery to encode probabilities converted in the Phred scale using the formulae above. For reference, see the following paper [4].

Unfortunately, the tests of my C implementation on 10s input found that the score was wrong on some of the alignments. I could not spot the bug, but I decided to include this algorithm still because the accelerated version of it is coherent with my C version.

4.3 GPU version

The idea behind both algorithms' GPU versions has been taken from the [2] paper. We can parallelize in two dimensions: intra-alignment (parallelize the computation of a single alignment to decrease response time) and inter-alignment (overlap different alignments to increase throughput).

Since the version of the Smith-Waterman algorithm does not have the backtrace part, we can use the parallelization trick for the Pair Hidden Markov Model alignment algorithm. We don't need to remember the whole matrix; having only the three dependent antidiagonals is enough. To further speed up the algorithm, we can save this antidiagonal in the shared memory due to the lower access times concerning the global memory. It can be seen from the recursion in the paragraphs 4.1 and 4.2 that the recursion needs data that reside at most in the previous two antidiagonals, for example, in the case of the Smith-Waterman algorithm, to compute a cell of the D matrix are necessary only the cells form two antidiagonals before, conversely, to compute the cells of the P and Q matrixes are necessary the cells from the previous antidiagonal. This optimization allows us to reserve space in the shared memory of each block to contain the antidiagonals so that we can considerably reduce the access time to this data structure.

The algorithm collects the alignments from an input file and stores them in the device's global memo. In the case of Smith-Waterman, the kernel is launched once with a grid size that amounts to half of the sequences (since the sequences are aligned in pairs in the input file, there are N sequences, and in the output, we will have N/2 scores.). Differently, in the case of the PHMM algorithm, the kernel is launched once for a batch of sequences. Each block will perform an alignment. Each block has a multiple size of the warp size (32).

The idea has been implemented thanks to the study performed on this book [5].

5 Results

The code has been compiled and run on the ETH CLUSTER [8] servers with the AMD INSTINCT MI210 [1]. The goal of this section is to analyse the performances of the algorithms with inputs of alignments of different length, to find the optimal block size for each input. This analysis has been performed only on Smith Waterman since the PairHMM version that I implemented isn't correct with respect to the original algorithm, thus making meaningless an analysis on performances because I can't make comparisons with the state of the art.

Here are the reported results for Smith Waterman. The analysis was also performed on PHMM with 10s dataset, and similar results were found, however as previously said I won't report them because I think they lack utility. The code has been given in input 50,000 random sequences that result in 25,000 alignments. This number of alignments completely fits in global memory. By running the code on different datasets and with

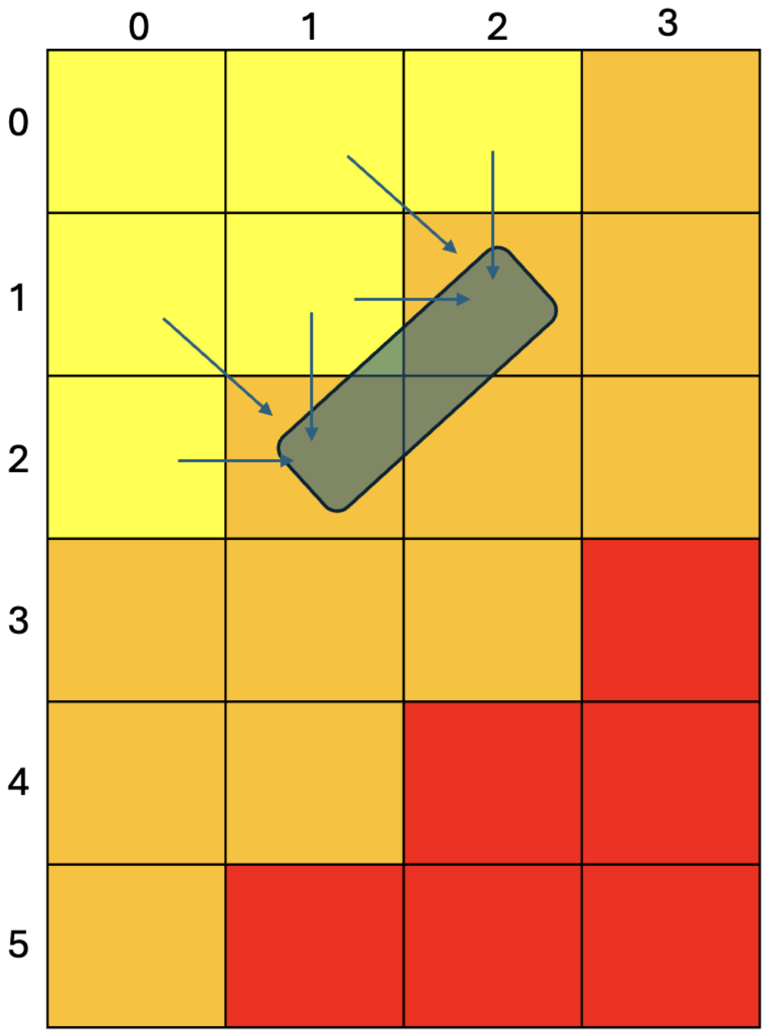

Figure 1: Rapresentation of the dependencies of each line, of the different phases of the algorithm, of the thread sliding window

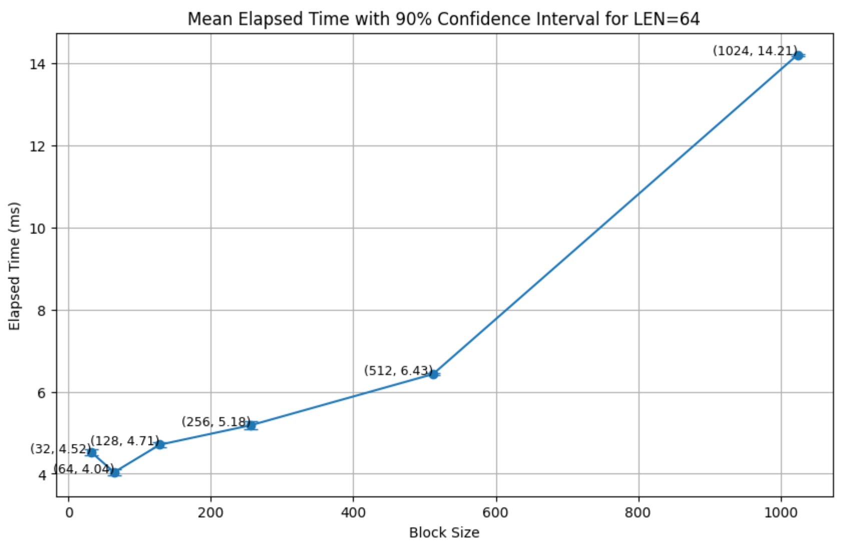

Figure 2: Input len 64.

varying sizes of block, it is evident from the graphs in the images (for example, see Figure 4) how there is always an optimal number of threads that grants the best performances.

Launching a more significant number of threads benefits performances because it increases parallelism; however, creating and managing many threads incurs a non-negligible overhead cost. The GPU dedicates resources to thread scheduling and synchronization, which can become substantial and not justifiable if no actual computation can be done. Moreover, memory bandwidth between global memory and registers is a precious resource. When an excessive number of threads attempt to access memory concurrently, ”contention” arises, resulting in delays as threads queue for their turn.

In the specific case studied, it is optimal to use a block size equal to the size of the biggest antidiagonal for inputs of length 64 and 256, however it is optimal with half of the size of the biggest antidiagonal for the other cases. This happens because the antidiagonal size perfectly maximizes the utilization of the GPU’s processing cores.

Warp size is essential when deciding the number of threads for each block. Launching a block size that is a multiple of the warp size guarantees that no threads are left idle. This happens because of the working logic of CUDA. When launching a kernel, the grid blocks are distributed between the available streaming multiprocessors. Subsequently, the threads in these blocks are grouped in warps of size 32; the hardware always allocates a discrete number of warps for a thread block. A warp is never split between different thread blocks; thus, if the thread block size is not an even multiple of warp size, some threads in the last warp are left inactive. And if the number of threads in the blocks is not a multiple of 32, there will be a warp with 32 threads, but some threads will be left idle. The threads in the warp execute in a SIMT (Single instruction multiple thread) fashion, meaning that they perform the same instruction simultaneously, each operating on its private data.

6 Conclusions

This project successfully accelerated two state-of-the-art algorithms using the massively parallel architecture of the GPU. By leveraging GPU technology, I explored the behavior

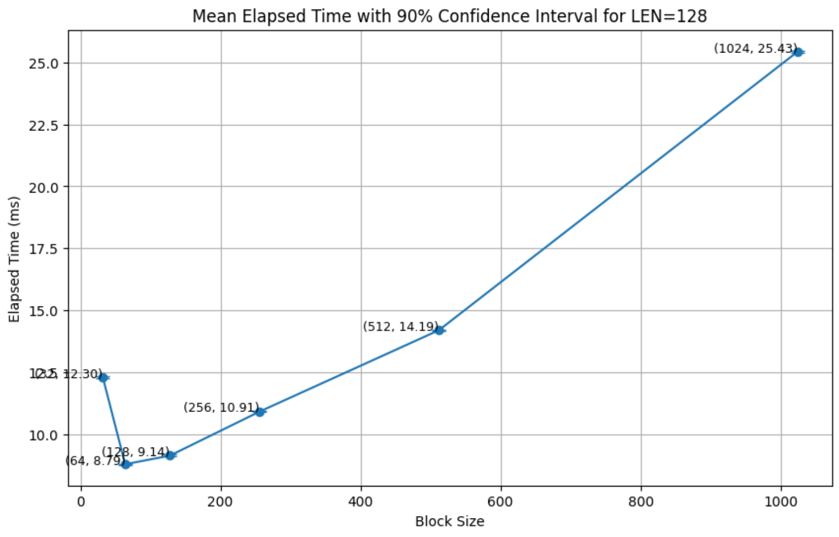

Figure 3: Input len 128.

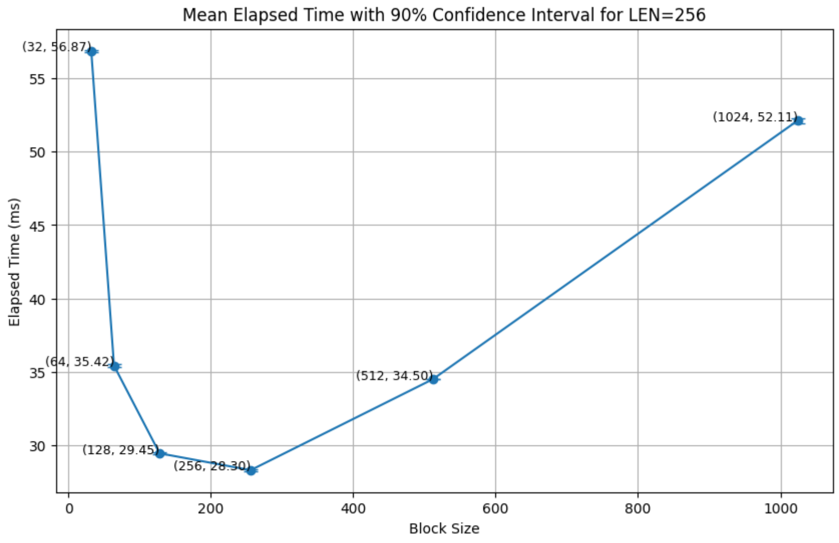

Figure 4: Input len 256.

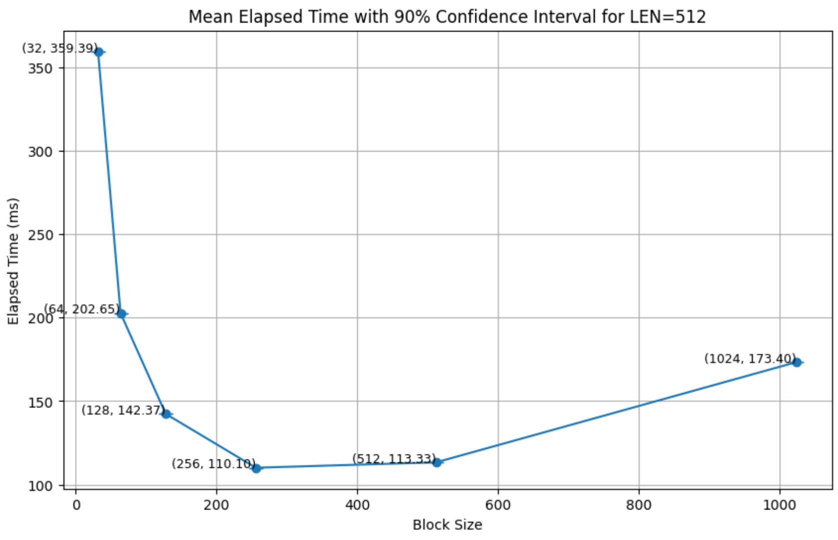

Figure 5: Input len 512.

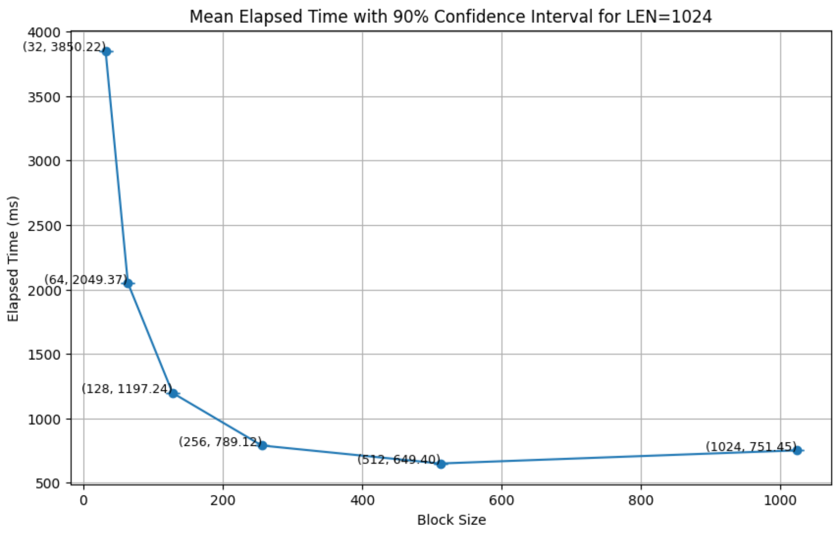

Figure 6: Input len 2024.

of Smith Waterman under various parameters and inputs. As a result, the performance of both algorithms significantly improved compared to the baseline implementation in C, demonstrating the substantial potential of GPU technology for computational acceleration.

References

[1] *AMD Instinct mi210 website.* URL: `https://www.amd.com/en/products/accelerators/instinct/mi200/mi210.htm`.

[2] Muaaz G. Awan et al. *ADEPT: a domain independent sequence alignment strategy for gpu architectures.* 2020. URL: `https://doi.org/10.1186/s12859-020-03720-1`.

[3] Beatrice Branchini, Alberto Zeni, and Marco D Santambrogio. "A Methodology for Accelerating Variant Calling on GPU". In: (2021).

[4] M Carneiro et al. *Enabling high throughput haplotype analysis through hardware acceleration.* 2017.

[5] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming.* Wrox : Programmer to Programmer. Wiley, 2014. ISBN: 9781118739310. URL: `https://books.google.it/books?id=Jgx_BAAAQBAJ`.

[6] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. *CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions.* 2013. URL: `https://doi.org/10.1186/1471-2105-14-117`.

[7] Svetlin A. Manavski and Giorgio Valle. *CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment.* 2008. URL: `https://doi.org/10.1186/1471-2105-9-S2-S10`.

[8] Javier Moya et al. *fpgasystems/hacc: ETHZ-HACC 2022.1.* 2023. URL: `https://doi.org/10.5281/zenodo.8344513`.