Passion in Action: Parallelized Smith-Waterman Implementation for GPU101 Course

Lorenzo Vergata
MSc in Engineering Physics
Politecnico di Milano
Milan, Italy
lorenzo.vergata@mail.polimi.it

***Abstract*—Being in the big data era, parallel programming is needed. This work aims to be my first step in the world of CUDA programming, trying to see if the parallel alignment of sequence of characters exceeds the serial counterpart, calculating the most similar sequence. This is done through the Smith-Waterman (SW) algorithm, by calcolating the scoring and direction matrices, then by finding the maximum score, followed by a traceback of the (most similar) sequence.**
After an introduction of the SW algorithm, it is described in an informal way the reason why this work exist. The challenge remains the same, but the starting code was modified a little: it is dedicated a short section on these changes. Then, after some words about GPUs architecture, the way of proceeding is provided.

***Index Terms*—gpu, pia, smith-waterman, cuda, parallel programming**

I. Introduction

This work is a fruit of the Passion in Action (PiA) GPU101 course at Politecnico di Milano. The students needed to implement the parallelized version of some well-known algorithm through GPU. More specifically, this code is dedicated to the Smith-Waterman (SW) algorithm, used, e.g., in bioinformatics to identify regions of similarity between two sequences. According to [1], the algorithm is as the following:

Algorithm 1 Smith-Waterman Algorithm for Local Sequence Alignment

Input: Sequences $A = a_1 a_2 \dots a_n$ and $B = b_1 b_2 \dots b_m$, similarity score function $s(a, b)$, and gap penalty W_k for gap length k.
Output: Optimal local alignment of A and B.
Step 1: Initialization
Construct a scoring matrix H of size $(n+1) \times (m+1)$.
Set $H_{k0} = 0$ for $0 \leq k \leq n$ and $H_{0l} = 0$ for $0 \leq l \leq m$.
Step 2: Fill the scoring matrix
for $i = 1$ to n **do**
  for $j = 1$ to m **do**
    Calculate H_{ij} using:

$$H_{ij} = \max \begin{cases} H_{i-1,j-1} + s(a_i, b_j), \\ \max_{k \geq 1}\{H_{i-k,j} - W_k\}, \\ \max_{l \geq 1}\{H_{i,j-l} - W_l\}, \\ 0 \end{cases}$$

  end for
end for
Step 3: Traceback
Find the highest score in H.
Start traceback from the cell with the highest score and continue until a cell with a score of 0 is reached.
Generate the local alignment based on the traceback path.

The extra first row and first column make it possible to align one sequence to another at any position, and setting them to 0 makes the terminal gap free from penalty. The traceback is given by the direction matrix, that has in the position *i,j* a value that represent the direction pointing to the right value to reconstruct the most similar sequence. In our case we used numerical values in this matrix, as:

- 1 or 2: for pointing to left-up;
- 3: for the left;
- 4: for up direction;
- 0: no direction.

In our case, we had $N = 1000$ sequences of length $S\_LEN = 512$ consisting in 5 nucleotides randomly chosen. At the end we have then two matrices of dimension $S\_LEN + 1 \times$

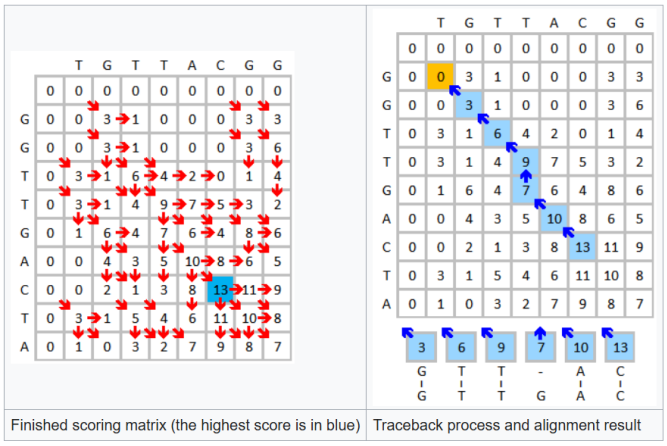

Fig. 1: The resulting alignment is *GTT-AC*

$S\_LEN + 1$.
Of course, each step can be parallelized in harder or easier manners. Due to difficulties found in the parallelized search of the maximum, because we stored solely the maximum value per each block, only the traceback and the construction of score and direction matrices were done.

II. The Reason Why of This Work

Of course, the time dedicate in this PiA was to try to achieve the following:

- Start to see and use some C/C++ code in order not to be scared of the course [2], because due to bachelor in material science and nanotechnology, I had to start programming first by myself and then thanks to the course [3];
- See how parallelization is done, exploiting the ways to (re-)model an algorithm;
- Due to syntax programming difficulties, use AI as Chat-GPT, Gemini and Copilot to be able to write something working;
- Understand more if I like programming or not;
- Use the **Collatz-Weyl Generators** [4] to create the random sequences.

III. Little Changes of the Original Code

The given C/C++ code was modified in order to generate the sequences through Collatz-Weyl Generators. Moreover, using as IDE CLion, the suggestions were taken into account. Nevertheless, because of the O3 optimization level, at the level of the machine almost nothing changes. Moreover, the function used to get time was reformulated in order to use a high resolution clock. Obviously, this does not affect the performance nor the effective precision due to the fact that *cout* stops at the sixth significant digit by default.
The seed was fixed in order to make the result be reproducible, in terms of created sequences.

IV. Some Words about GPUs and CUDA

Thanks to their architecture, GPUs can offer w.r.t. CPUs [5]:

- Massive parallelism;
- Higher throughput;
- Reduced computation time;
- Scalability and flexibility;
- CUDA and ecosystem support.

CUDA organizes parallel computations in a hierarchical structure to manage and scale parallelism efficiently. The smallest unit of parallel execution in CUDA is a **thread**. Threads are grouped into **thread blocks**. All threads in a block can communicate and synchronize on tasks. Threads in a block execute concurrently on a SM. When a CUDA kernel is launched, threads within each thread block are organized into groups of 32 threads called **warps**. A warp always contains 32 threads, regardless of the GPU architecture. All threads in a warp execute the same instruction at the same time (SIMT paradigm), but each thread operates on its own data element. A collection of blocks forms a **grid**. Blocks in the grid are independent of each other [5]. Each block is assigned

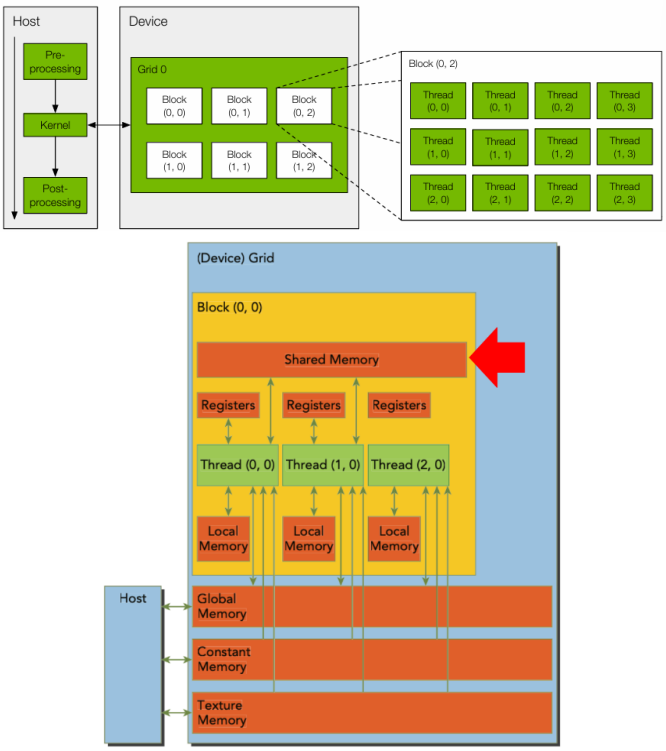

Fig. 2: Overview of the architectures we have. *Host* is CPU-side, while *device* is GPU-side. The accesses in the different memories play a key role to reach high performances, e.g., exploiting **coalesced access**. [5].

to a single streaming multiprocessor (SM), i.e., computational units. SMs have CUDA cores, tensor cores, L1 cache, and registers. The latters are the fastest form of memory available to each SM. L2 cache is used as an intermediate storage layer

between device memory and the SMs [5].
The nearer the memory, the faster the process, and the same is with a better exploitation of the multiple of 2 (or 32) in the utilization of the resources.

V. Modus Operandi

We can see in the algorithm in 9 that an element in position *i,j* in the scoring matrix does not depend on the element in *i-1,j+1*, i.e., the right-up score, so that we can make full use of the fact that if the right-up element of the score matrix exists, then we have all the elements to compute the final result in the cell *i,j*. This method is a sort of parabola in terms of number of not idle threads, with only a single operative thread in the first and last computation of the score matrix, while a maximum of S_LEN when computing the last element of the first row.
Because of certain idle threads, the block size is of course N, i.e., the total number of strings, having S_LEN threads for each block.
Before starting the computation, the kernel uses shared memory to store the block-level maximum score and its corresponding indices. The first row and column of the score and direction matrices are initialized to zero using coalesced memory access, i.e., each thread initializes multiple matrix elements in the same row or column by incrementing by the dimension of the block (along x).
The main computation follows a diagonal traversal of the scoring matrix, and after the function *dir_score*, the maximum score and its indices are updated atomically by threads. Finally, the block-level maximum score and indices are written to global memory.
After this, backtracking is performed to reconstruct the alignment path in such a way that starting from the maximum score position (i, j), threads independently traverse the direction matrix.
The output of the kernel provides the highest alignment score for each pair and the position where the alignment ends. For the global maximum, it was relied on a for cycle on the host, having an array of only N element, saving then the index of the string that has at the end the most similar sequence.
For a fair comparison, the time on the GPU version is not entirely on the GPU, but it takes account also on the memory transfers and on this final iteration for the global maximum across all the highest value in each block.

VI. Experimental Results and Discussion

The code was executed on a NVIDIA GeForce RTX 3050 Laptop, paired with an AMD Ryzen 5600H processor, with the plug for power inserted. The maximum number of threads per block is 1024, enough for our task.
The following steps were parallelized:

- Score and direction matrices: each block of threads is responsible for computing the score matrix (sc_mat) and direction matrix (dir_mat) for a specific sequence pair. Inside a block, threads work to compute the scores for the matrix cells using the diagonal, and the computations for the next diagonal wait until the previous one is complete;
- Traceback: After the highest score (max_score_shared) and its corresponding indices (max_i_shared, max_j_shared) are found for a block, each thread contributes to the traceback process, building a reversed string ($simple\_rev\_cigar$), which records the sequence of operations required to align the query and reference sequences.

Unfortunately, the for cycle in the host and the copy processes from device to host and vice versa slowed the entire run, ending up in a speed up of ≈ 2.5. Nevertheless, the kernel itself compared to the serial process, sees a speed up of ≈ 1000 starting from the second run of the code.

VII. Conclusion

For sure, the parallel programming is harder than the serialized one, and it must when high performances are needed. Also, the parallelization itself is not enough: a wise utilization of the resources must be present, because one can end up in a simple speed up that is not worth for the time invested. It is a "hardcore" programming, that even 3 AI (ChatGPT, Copilot, Gemini) cannot handle well if one has no idea of what strategy is to be implemented. In fact, they suggested a *wave-fronts parallelization* where it was not possible to see any result because the shared memory capacity of 96 kB was surpassed.
To make the code run faster it is needed a parallel search for the maximum, because using AI it was not possible for me to find out a true solution, as the result does not change in terms of time.

References

[1] Wikipedia contributors, "Smith–Waterman algorithm — Wikipedia, The Free Encyclopedia," 2025. [Online]. Available: https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm

[2] Politecnico di Milano, "Data Mining Course — Politecnico di Milano," 2025. [Online]. Available: https://www11.ceda.polimi.it/schedaincarico/schedaincarico/controller/scheda_pubblica/SchedaPublic.do?&evn_default=evento&c_classe=837799&lang=IT&__pj0=0&__pj1=87eb10dc7712c59a131823d49d969417

[3] ——, "Machine Learning Course — Politecnico di Milano," 2025. [Online]. Available: https://www11.ceda.polimi.it/schedaincarico/schedaincarico/controller/scheda_pubblica/SchedaPublic.do?&evn_default=evento&c_classe=841196&lang=IT&__pj0=0&__pj1=fda9e077a72ebd235b575855f4a9bca2

[4] T. R. Działa, "Collatz-weyl generators: High quality and high throughput parameterized pseudorandom number generators," 2024. [Online]. Available: https://arxiv.org/abs/2312.17043

[5] Beatrice Branchini, "GPU101 - Lecture 1: Introduction and CUDA Programming Model," 2024-2025.