CUDA implementation of the Smith-Waterman algorithm

Fabrizio Maria Aymone*

Politecnico di Milano
fabriziomaria.aymone@mail.polimi.it

December 31, 2022

Abstract

The Smith–Waterman algorithm is a method widely used in bioinformatics for identifying regions of similarity between two sequences of nucleic acids or proteins, i.e. sequence alignment. The algorithm is computationally intensive and time demanding. Nevertheless, several parts of the latter can be parallelized drastically improving overall performance. In this report, I will describe my implementation in C of the Smith-Waterman algorithm for gpu parallel programming using CUDA. Lastly, I will compare the results obtained via gpu acceleration with the cpu version of the algorithm.

I. Introduction

Sequence alignment is a technique used in bioinformatics to find similar regions in DNA, RNA or protein sequences as they may represent functional, structural or evolutionary relationship between sequences. The most interesting challenges in genomics involve the computation of the alignment of very long sequences, resulting in the need for computational power and efficient algorithms. The approaches to sequence alignment can be divided in global and local alignments. On the one hand global alignment enforces an end-to-end alignment of the two sequences, while on the other hand local alignment focuses on identifying alignment between local sub-sequences of the original sequences. A famous global alignment algorithm is the Needleman-Wunsch, which assigns a score to every possible alignment in order to, then, find all highest score alignments. Along the lines of the latter, the Smith-Waterman local alignment algorithm was developed.

*Second-year Electronic Engineering Student at Politecnico di Milano

II. The algorithm

The Smith-Waterman algorithm performs sequence alignment of two strings using insertions, deletions, matches or mismatches, where insertion and deletions introduce a gap in the original sequences. In the following section I will describe specifically the implementation of the Smith-Waterman algorithm which has been provided for the current project. It may differ slightly from other implementations. It is possible to divide the approach in four main phases:

- Penalties setting
- Scoring and Direction matrices initialization
- Scoring and Direction matrices calculation
- Traceback

The first step consists in setting the match, mismatch, deletion and insertion values based on the case we are treating. The match and mismatch values can have different values according to which sequence characters are considered. This gives origin to a substitution matrix. In the second part, the scoring and direction matrix are initialized to zero along the first

row and first column. These matrices have as x-dimension the length of the query sequence plus one and as y-dimension the length of the reference sequence plus one. Thirdly, the algorithm fills out the Scoring matrix according to the following rule:

$$H_{ij} = \max \begin{cases} H_{i-1,j-1} + s(a_i, b_j), \\ H_{i-1,j} - del, \\ H_{i,j-1} - ins, \\ 0 \end{cases} \quad (1)$$

where i and j range from 1 to the length of the respective sequence. H_{ij} is the element in the scoring matrix at row i and column j, considering that the first column has corresponds to $j = 0$ and analogously for the rows. $s(a_i, b_j)$ is a function that for every combination of character sequences a_i coming from the query sequence and b_j coming from the reference sequence outputs the corresponding match/mismatch value. For the convention being used the first element of the reference sequence has index equal to one and the same goes for the query sequence. *del* and *ins* are the penalties for insertion or deletion. The Direction matrix is simultaneously filled. The last phase of the algorithm consists in performing the traceback to find the optimal alignment. This methods begins from the position of the element with highest score in the previously filled Scoring matrix and following the directions indicated in the, also previously filled, Direction matrix it computes the best alignment. It is worth noticing that if the scoring matrix has more elements with maximum score, all the alignments obtained with traceback starting at the coordinates of the maxima are all optimal, as they have the same score.

III. Parallelism

The first step into writing a parallelized version of the algorithm suitable for a NVIDIA gpu is understanding the dependencies between the different operations. Beginning from the initialization, it is possible to completely parallelize this operation by launching a kernel where each thread is mapped to a corresponding element of the matrix and only the threads mapped to the first row or first column update their element to zero (even if all the matrix is updated to 0, it does not matter for the further computations). The second step is the filling of the Scoring matrix (which occurs at the same time of the Direction matrix). The calculation of the general element at row i and column j depends only on the three elements located upwards, left-upwards and leftwards. This means that to calculate the elements sitting at a generic anti-diagonal of the Scoring matrix, we only need to know the two previous (i.e. left-upwards) anti-diagonals. This concept is at the heart of my implementation as it enables to exploit the fast-access shared memory. Lastly, the traceback is sequential, as each new element of the optimal alignment depends on the previous one.

IV. The CUDA implementation

To begin with, as for the project I needed to calculate the alignment of N different query-reference couples, I assigned each of these couples to a different block, as they are completely independent. For matrix initialization, I preferred to use the *cudaMemset*() function, instead of the kernel launch as it shows similar results in terms of latency.
For the filling of the matrices, I exploited the parallelism related to the anti-diagonals by putting in the kernel a for loop that iterates for every anti-diagonal. At each iteration, it computes the elements in the current anti-diagonal and stores the maximum element performing an *atomicMax*(). At the end, it synchronizes the threads before entering the next cycle. Considering that in our case the matrix has dimensions 513x513 (i.e. the longest diagonal has 513 elements) and that the maximum number of threads in a kernel is 1024, it is possible to perform all the filling operations inside a single kernel.

This parallelism offers an great advantage in terms of memory access. The difference in latency between shared and global memory vary drastically, i.e. $\sim$ 32 vs $\sim$ 800 clocks. To take the most out of our program we should use as much as possible shared memory. The main drawback of using shared memory is its low memory size of 48 KB, that does not let us store the two matrices. By using the parallelism related to the anti-diagonals, we do not need anymore to store any Scoring Matrix, as we compute its elements and find the maximum with its coordinates "on the go". We still, however, need to store the Direction matrix in global memory as we need it later to perform backtracing. Therefore, we will use shared memory to store three arrays of length equal to the maximal possible length of the anti-diagonal, namely 513, corresponding to previous-previous, previous and current anti-diagonals. Lastly, only a single thread is selected to perform the backtracing starting from the coordinates of the maximum elements and giving us the final alignment.
In conclusion, we compare the maxima and the alignments of the different query-reference couples obtained with the cpu implementation with the ones obtained with the gpu implementation. As in a Scoring matrix there could be more than one element with highest score, the cpu and gpu algorithms may select different coordinates for the maximum and, consequently, producing different alignments. Even if theoretically the alignments obtained by the implementations are both correct by having highest score, I decided to write two different versions of the gpu algorithm so that one could match the results obtained in the cpu. I called the version that does not care about aligning with the cpu *pure* and the one that uses the same convention as the cpu *conventional*. The convention used to select the maximum picks the element with the highest score located in the lowest anti-diagonal and in the rightest tile along this anit-diagonal

V. Results

To benchmark the gpu implementation against the cpu one, we ran the algorithm on Google Colab on two different GPUs; the A100-SXM4-40GB and the Tesla T4. The results in terms of time of execution of the algorithm are reported in the following table. Hence, we can derive the relative speedup between cpu and gpu for the four cases. For the Tesla T4, the speedup in latency of the gpu implementation compared to the cpu is $\sim$ 25.8x for the pure version and $\sim$ 22.6x for the conventional one. For the A100-SXM4-40GB, the speedup of the gpu compared to the cpu is $\sim$ 266.4x for the pure version and $\sim$ 236x for the conventional one. The pure performs slightly better than conventional for both gpus. However, in the technically inferior gpu, i.e. Tesla T4, this difference is amplified to $\sim$ 10ms, compared to $\sim$ 1ms in the A100-SXM4-40GB. This is due to the fact that in the conventional version of the gpu implementation the kernel performs an additional atomic operation to keep the same convention as the cpu, enforcing sequentiality between threads. Probably, the A100-SXM4-40GB is capable of handling atomic operations and threads syncing, the main bottleneck of my implementation, far better than the Tesla; hence, the huge improvement in speedup.

Table 1: *Execution time*

Tesla T4		
Device	Version	Time (s)
CPU	-	2.2840359211
GPU	pure	0.0886352062
GPU	conventional	0.1009891033
A100-SXM4-40GB		
Device	Version	Time
CPU	-	2.7850348949
GPU	pure	0.0104529858
GPU	conventional	0.0118029118

VI. Discussion

i. Scalability

One of the major issues with my implementation lies in scalability. Firstly, I assumed that the longest anti-diagonal of the matrices (which corresponds to the length of the shortest sequence in the query-reference couple, plus one) has a length inferior or equal to 1024, the maximum number of threads in a block. If we consider query-reference couples where the shortest string has length larger than 1023 my implementation fails, as it currently allocates the computation of the matrix to each single block. In this latter case, we should subdivide the matrix calculation between multiple blocks and, than, store the arrays for the anti-diagonals computations in global memory losing the advantage of the fast-access shared memory. This modification would give rise to other problems such as thread syncing (CUDA allows syncing only between threads in the same block).

ii. Alternative to Direction matrix

The gpu implementation performs backtracing by storing a full Direction matrix in the global memory and filling it "on the go" simultaneously with the Scoring matrix (more precisely, with the three anti-diagonal arrays). At the end, one single threads calculates the optimal alignment starting from the coordinates of the maximum and following, via the backtrace method, the directions in the Direction Matrix. This algorithm does not exploit fast-access shared memory, as such matrix exceeds shared memory limits. An alternative to the current program would not use any Direction matrix and would perform backtracing by recalculating the N previous anti-diagonals to the anti-diagonal where the maximum lies, do local backtracing up until the first of the N anti-diagonals, i.e. the left-upmost one, store the coordinates where this local backtracing operation ended, and iterate again the cycle. N should be the largest number possible with the constraint for the anti-diagonal arrays to be stored in shared memory. Even if this method exploits the most of shared memory as the kernel accesses the global memory only to read the query-reference couple and to write the results, it introduces far more computations which grow with the length of the shortest sequence in the query-reference couple and diminish with N. Hence, it introduces another issue related to scalability as N is constrained by the size of shared memory. Further studies on this matter should be done.

VII. Github repository

At the following link it is possible to find the github repository containing the implementation described in the report with an easy-to-follow guide to reproduce the results. https://github.com/fabrizioaymone/cuda-smith-waterman