GPU101 PiA Course Project

Prof. Alberto Zeni

Agnese Taluzzi

A.A. 2022/2023

Contents

Chapter 1

Introduction

1.1 Objective

The project aims to provide a GPU-accelerated implementation of a given application that implements the Smith-Waterman algorithm, an optimal algorithm for the local alignment of a pair of sequences. The program applies the algorithm to N (1000) pairs of sequences of length S_LEN (512) and performs the respective backtracing.

1.2 Smith-Waterman

The Smith–Waterman algorithm performs local sequence alignment; that is, it compares segments of all possible lengths and optimizes the similarity measure. It consists of computing a scoring matrix of size $(S\_LEN + 1) * (S\_LEN + 1)$ with the constraints represented in Figure 1.1 (weights are set before the computation).

$$H(i,0) = 0,\ 0 \leq i \leq m$$

$$H(0,j) = 0,\ 0 \leq j \leq n$$

$$\text{Se } a_i = b_j\ w(a_i, b_j) = w(\text{Match})\ \text{ o se } a_i != b_j\ w(a_i, b_j) = w(\text{Substitution})$$

$$H(i,j) = \max \left\{ \begin{array}{cc} 0 & \\ H(i-1, j-1) + w(a_i, b_j) & \text{Match/Substitution} \\ H(i-1, j) + w(a_i, -) & \text{Deletion} \\ H(i, j-1) + w(-, b_j) & \text{Insertion} \end{array} \right\},\ 1 \leq i \leq m, 1 \leq j \leq n$$

Figure 1.1: Scoring method of the Smith-Waterman algorithm

Then, starting at the highest score in the scoring matrix H and ending at a matrix cell that has a score of 0, it computes the traceback based on the source of each score recursively to generate the best local alignment.

Chapter 2

Implementation

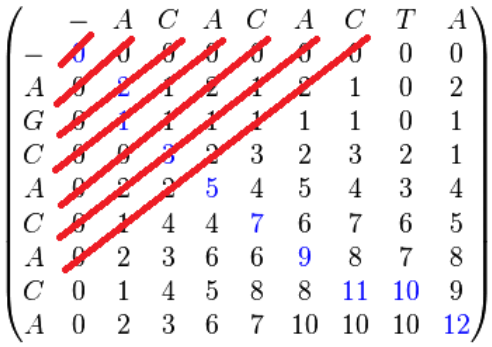
In this chapter, I present the three different programs I implemented. The sections follow a chronological order in order to point out the design process and the choices I made to reach the final (and correct) product.

2.1 One Thread per Block

In this first implementation, the execution is distributed among N different blocks (one per each pair of strings), and one thread per block. The only form of parallelization is in the computation of the N comparisons. The algorithm is the same as the host one, simply executed on the device. The drawback is that even with GPUs of computing capability 9.0, the maximum amount of shared memory per thread block is 227 KB (for older GPUs is 48 KB, for the Tesla T4, which is the one I used, is 64 KB). This means that we cannot store a matrix of dimension 512x512 integers (¿1000 KB), in shared memory. The only (naive) way is instanciating the score matrix and the direction matrix in the global memory of the device.

2.2 Direction Matrix in Shared Memory

The next phase was parallelizing the computation of the score matrix. I noticed that each element of the anti-diagonals could be computed at the same time, and that for each anti-diagonal I could save only the last two ones that had been computed, instead of the whole scoring matrix. As a result, I changed the number of threads per block from one to S_LEN (one for each element of the anti-diagonal of maximum length), exploiting the CUDA SIMT execution model. To optimize the speedup I also kept the direction matrix in shared memory, but as I said before this is unfeasible with strings of the given length.



	–	A	C	A	C	A	C	T	A
–	0	0	0	0	0	0	0	0	0
A	0	2	1	2	1	2	1	0	2
G	0	1	1	1	1	1	1	0	1
C	0	0	3	2	3	2	3	2	1
A	0	2	2	5	4	5	4	3	4
C	0	1	4	4	7	6	7	6	5
A	0	2	3	6	6	9	8	7	8
C	0	1	4	5	8	8	11	10	9
A	0	2	3	6	7	10	10	10	12

Figure 2.1: Anti-diagonals

2.3 Final Implementation

The final implementation differs from the previous one simply by saving the direction matrix in the global memory of the device, making the execution feasible for strings with S_LEN = 512. The backtracking phase remains the only one not parallelized, with one thread per block executing it.

Chapter 3

Experimental Results

Other than checking the correctness of the results of the GPU-accelerated implementation by comparing them with the results obtained with the host one, I measured the performance of my implementation.

3.1 With and Without Shared Memory Comparison

For S_LEN = 128, saving the direction matrix on shared memory is possible and yields better results:

```
SW Time CPU: 0.1523220539
SW Time GPU: 0.0027489662
```

The GPU implementation is about 55 times faster.
Without shared memory:

```
SW Time CPU: 0.1611959934
SW Time GPU: 0.0059340000
```

The GPU implementation is about 27 times faster, but it takes approximately twice the time obtained with shared memory.

3.2 Final Product Performance

With the original parameter S_LEN set to 512, the performance is as follows:

```
SW Time CPU: 2.4860649109
SW Time GPU: 0.3083939552

SW Time CPU: 3.1018850803
SW Time GPU: 0.3011970520
```

The speedup is beetween 8 and 10.
By comparing these results with the above ones, I can conclude that the major bottleneck is the backtracking phase. As a matter of fact, increasing S_LEN leads to a meaningful degradation of the performance of the GPU. This is caused by the fact that backtracing is executed in a sequential way for each block.