GPU101 PiA Course Project

Agnese Taluzzi

A.A. 2022/2023

Chapter 1

Introduction

1.1 Objective

The objective of this project is to develop a GPU-accelerated implementation of the Smith-Waterman algorithm, an optimal method for locally aligning pairs of sequences. The implementation focuses on processing 1000 sequence pairs, where the sequences have a length of 512, and conducting the necessary back-tracing operations.

1.2 Smith-Waterman

The Smith-Waterman algorithm is designed for local sequence alignment, where it compares segments of varying lengths and optimizes the similarity measure. During the initial phase, the algorithm consists of computing a scoring matrix with dimensions $(sequence\_length + 1) * (sequence\_length + 1)$, taking into account the constraints outlined in Figure 1.1. The weights associated with these constraints are set before the computation.

In the following step, the program initiates the traceback process by identifying the highest score within the scoring matrix. It then proceeds recursively, following the sources of each score until reaching a matrix cell with a score of 0. By computing the traceback in this way, the program generates the optimal local alignment for the given sequences.

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

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

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

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

Figure 1.1: Scoring method of the Smith-Waterman algorithm

Chapter 2

Implementation

In this chapter, I present the three main phases I followed for the implementation of the accelerated program. The sections follow a chronological order in order to point out the design process and the choices I made to reach the final product.

2.1 One Thread per Block

In the first step, I distributed the execution across N (1000) distinct blocks, with each block handling one pair of sequences. The parallelization is achieved by computing the N comparisons concurrently. Initially, I assigned only one thread to each block. In this way, the algorithm employed on the device is identical to the one used on the host.

2.2 Parallelization of Score Matrix Computation

The subsequent step 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 sequence_length (which is equal to the size of the anti-diagonal of maximum length), exploiting the CUDA SIMT execution model. Furtherly, I noticed that for computing the maximum score I could keep a local maximum for each thread of the block, and then take advantage of parallel reduction. The backtracking phase in this implementation remains partially parallelized, with only one thread per block responsible for its execution.

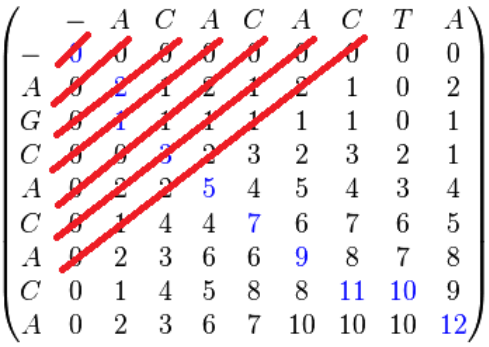

Figure 2.1: Anti-diagonals

2.3 Shared Memory

To optimize the speedup. I tried to instantiate the scoring matrix (in my case, only the last two computed diagonals) and the direction matrix in shared memory. However, there is a limitation on the available shared memory per thread block. Even with GPUs of computing capability 9.0, the maximum shared memory per block is 227 KB. For older GPUs, this limit is even lower, and for the Tesla T4 GPU (which was utilized in this implementation), the limit is 64 KB [1]. This means that storing a matrix of dimensions 512x512 integers is not feasible within the shared memory. To overcome this limitation, the direction matrix is instantiated in the global memory of the device. This is a feasible solution but results in increased memory access latency compared to using shared memory.

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/#compute-capabilities

Chapter 3

Experimental Results

3.1 Performance Analysis

To validate the accuracy of the GPU-accelerated execution, the program verifies the consistency of its results by comparing them against the ones obtained from the host implementation. Additionally, to evaluate the performance of the implementation, I calculated the mean of the outcomes of 100 executions. This average does not consider the API calls to allocate and copy memory. The time taken by these operations is reported in Section 3.2. The results are presented in Table 3.1.

CPU time	2.6887068200 s
GPU time	0.0561734819 s

Table 3.1: Comparison of execution times

Based on the given CPU and GPU time, the average speedup is approximately 47.86.

3.2 Profiling

In Table 3.2, I present the results obtained from profiling the program using Nvprof.

Name	Time(%)	Time	Calls	Avg	Min	Max	Description
GPU activities	99.69%	56.424ms	1	56.424ms	56.424ms	56.424ms	sw_gpu
	0.16%	89.726us	2	44.863us	44.127us	45.599us	memcpy HtoD
	0.15%	83.038us	2	41.519us	2.5920us	80.446us	memcpy DtoH
API calls	76.37%	196.20ms	5	39.240ms	3.8310us	195.83ms	cudaMalloc
	21.96%	56.430ms	1	56.430ms	56.430ms	56.430ms	cudaDeviceSynchronize
	0.66%	1.6858ms	5	337.15us	4.9000us	1.0808ms	cudaFree
	0.48%	1.2419ms	4	310.48us	52.967us	887.26us	cudaMemcpy
	...	...	...	...	...	...	...

Table 3.2: GPU Activity and API Calls