The Smith-Waterman algorithm in CUDA

Matteo Matassoni

GPU101 Course

Politecnico di Milano

Abstract

This repository contains an implementation of the Smith-Waterman algorithm.
Starting from a sequential version running on a CPU, I accelerated it using a parallel approach using the potentiality of a GPU.

The algorithm

The Smith–Waterman algorithm performs local sequence alignment; that is, for determining similar regions between two strings of nucleic acid sequences or protein sequences. Instead of looking at the entire sequence, the Smith–Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure.[1]

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

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

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

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

[1] https://en.wikipedia.org/wiki/Sequence_alignment

The functionality of the Algorithm

My solution computes the solution of the problem in two different ways: the first uses CPU (with the provided algorithm) and the second one uses GPU. In this way is very simple to comprehend which version is faster and if the results are correct.

The output of the program is the time taken by the two solutions and if they are equal.

GPU version

To create the GPU version of the algorithm I followed the structure of the CPU version which can be identified as:

1. Randomly generate sequences
2. Initialize the scoring matrix and direction matrix to 0
3. Compute the alignment
    - Compare the sequences of characters
    - Compute the cell knowing the comparison result

The first step is obviously executed only one time (by the CPU) and the query matrix and the reference matrix are initialized at the start of the program and used also by the CPU version.
The first thing that I've done is allocate the memory for the matrices on the GPU and copy the data from the host.
I used a standard approach that speeds up the transfer of data between CPU and GPU (that is very slow) this is for example the copy of the query matrix:

```
    char **d_query;

    char **d_queryh = (char **)malloc(N * sizeof(char *));

    CHECK(cudaMalloc((void***)&d_query,  N * sizeof(char *)));

    for(int i=0; i<N; i++) {

        CHECK(cudaMalloc((void**) &(d_queryh[i]), S_LEN*sizeof(char)));

        CHECK(cudaMemcpy (d_queryh[i], query[i], S_LEN*sizeof(char),
cudaMemcpyHostToDevice));

    }

    CHECK(cudaMemcpy (d_query, d_queryh, N*sizeof(char *), cudaMemcpyHostToDevice));
```

After this initial step, the program calls the kernel function `gpuSW<<<N, S_LEN>>>` using N blocks and S_LEN threads.

To calculate the maximum value in the CPU version, the program stores the value found up to that moment in a variable and at each subsequent iteration, it compares the value found with the one stored, in this way it can easily find the global maximum value.
In a parallel approach, this can't be done because each thread calculates its max simultaneously. Only after knowing all the max is possible to find the global max for the block.
For this reason, the GPU has to save all the max and so I created two arrays:

- `__shared__ int value_max[S_LEN]` to store the max value
- `__shared__ int pos_max[S_LEN]` to store the position of the max value

After initializing the scoring matrix and direction matrix to 0 the program starts to compute the alignment, which is pretty similar to the CPU version.
The most important thing is that the operation must be executed in order so I use the `__syncthreads()` so every operation block is executed in the right order.

Result

I have run this program a lot of time on Colab with these resources:

- 12 GB of RAM Memory
- 12 GB of GPU Memory
- 0.82 GHz of GPU MemoryClock
- GPU Model: Tesla T4

And the average time is:

- CPU time: 7,0534623691 (2,3764234 whits the O3 optimization)
- GPU time: 0,0000281333 (0,0000313 whits the O3 optimization)

So the program on GPU is 250000 times faster (75000 with O3 optimization), of course, this can be improved but I think this is a very good result.

References

- Original Code
- Wikipedia: Smith-Waterman algorithm