# Smith-Waterman Algorithm

Nicolò Dellepiane

# Table of Contents

# What is the Smith-Waterman algorithm?

The Smith-Waterman algorithms is used to determine the *grade of similarity of two nucleic-acidic sequences*. More specifically, it finds the regions of the two strings that are the most similar, within some set parameters.

# Sequential implementation

The algorithm confronts each character of the query string with each character of the reference string.
It then computes the score for each possible action it can make, based on the scores computed for the actions taken on the immediatly adjacent values

Each action is associated with a prize/penalty.
The action with the higher score will be taken and the said score will be stored. If a score happens to be negative, a 0 is stored instead

# Auxiliary data structures

The maximum possible score for each possible combination is stored in the matrix `score_matrix`.
To each action (match/mismatch, insert, delete) is associated a direction (respectively, upLeft, left, up). These directions are stored at the same index of their corresponding value, in the apposite matrix `direction_matrix`

# Example Image

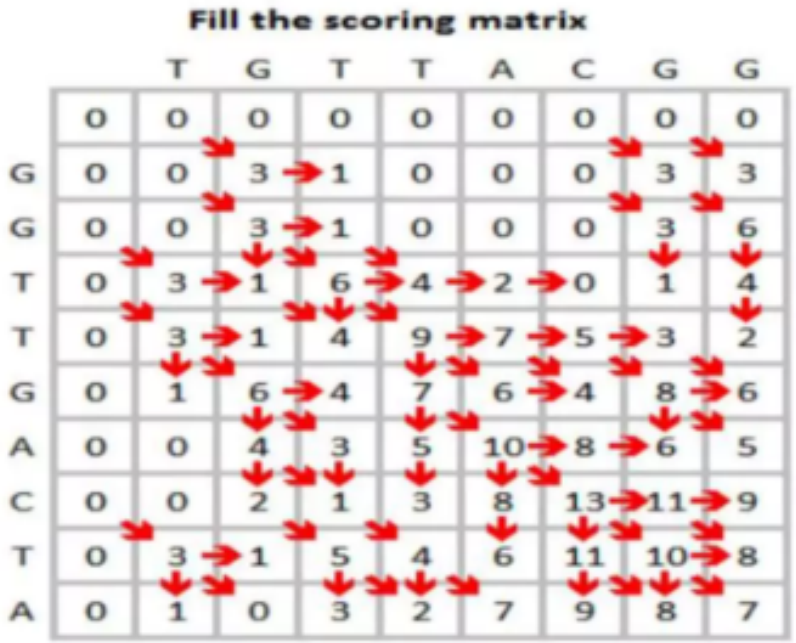


Figure: Example of score matrix

Each score is the maximum possible value obtained adding the prize/penalty associated with a move, and the value of the element from the direction associated with that move

# Parallel implementation

Parallelism can be achieved at two different levels:

1. **Grid-level parallelism:** since 1000 query/reference couples are given from the start, a block can compute a pair of sequences independently
2. **Block-level parallelism:** within a block, the value $M_{i,j}$ only depends on values $M_{i-1,j}, M_{i-1,j-1}, M_{i,j-1}$. *This means that all values on the same "diagonal" can be computed at the same time*, and diagonals need to be commputed sequentially

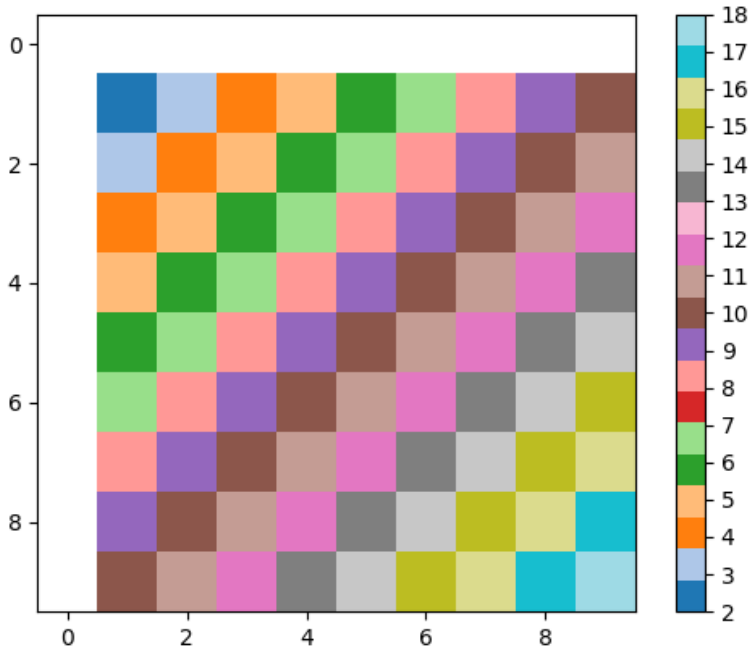


Figure: Computation pattern

Scores and directions are still stored in two matrices (one pair per block), and are passed from device to host in order to find the maximum score and backtrace from there the best fitting subsequence

# Optimizations

Running a code profiler reveals the bottleneck of this implementation are the big `cudaMemCpy` and `cudaMalloc` API calls used to allocate the matrices for each block (more than 1GB of data)

## Reducing memory operations

Memory operations can be significantly reduced by computing the maximum score on device, and by using shared memory to store the scores needed to compute the new ones

# Conclusion

The final cuda code executes in 0.5/0.6 seconds on an RTX 3060, while the unoptimizd sequential code runs in around 3 seconds on a ryzen 5 7600.
Even with compiler optimizations enabled (-o3 flag set), the parallel algorithm runs slightly faster than the sequential one (0.4 against 0.7 seconds on average).
Thus, i consider my implementation of the algorithm to be successfull