FPGA-Accelerated Smith-Waterman Algorithm

1st Christodoulos Zerdalis
Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece
chzerdalis@uth.gr

2nd Tsiantos Dimitrios
Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece
dtsiantos@uth.gr

***Abstract*—The Smith-Waterman algorithm is a dominant method used for local sequence alignment in bioinformatics, providing a consistent and accurate way to identify similar regions between genome sequences. However, its computational complexity makes it slow and energy-intensive on general-purpose processors. This creates the need for a faster and more efficient way to run the algorithm. To address this issue, we present an implementation of the Smith-Waterman algorithm on a Field-Programmable Gate Array (FPGA) taking advantage of the customizability, efficiency and parallelism of FPGAs.**

***Index Terms*—Local sequence alignment, Smith-Waterman, FPGA**

I. Introduction

Sequence alignment in bioinformatics is the process of comparing two or more DNA, RNA or protein sequences providing useful functional, structural and evolutionary information. The two types of sequence alignment are global and local. In global alignment the sequences - which are usually similar in size - are compared to each other from start to end, maximizing character matches. In contrast, local alignment compares sequences that may differ significantly in size, searching for the region with the best match [1]. The most widely used method for global alignment is the Needleman-Wunsch algorithm [2] while for local alignment, it is the Smith-Waterman (SW) algorithm introduced by Temple F. Smith and Michael S. Waterman in 1981 [3]. Both are dynamic programming algorithms with significant data dependencies. In this work, we implement and optimize the Smith-Waterman algorithm.

II. The Algorithm

Given two sequences, a query Q and a database D the goal of the SW algorithm is to find regions of similarity between the sequences. The lengths of the query and the database are N and M, respectively. The dynamic programming matrix S of length $M \times N$ and elements $S_{i,j}$ where $i = 1, 2, \ldots, M$ and $j = 1, 2, \ldots, N$ keeps track of the similarity score:

$$S_{i,j} = \max \begin{cases} S_{i-1,j-1} + S(d_i, q_j) & \\ S_{i-1,j} - 1 & 1 \leq i \leq M \\ S_{i,j-1} - 1 & 1 \leq j \leq N \\ 0 & \end{cases}$$

The term $s(d_i, q_j)$ is the alignment score of the elements d_i and q_j defined as:

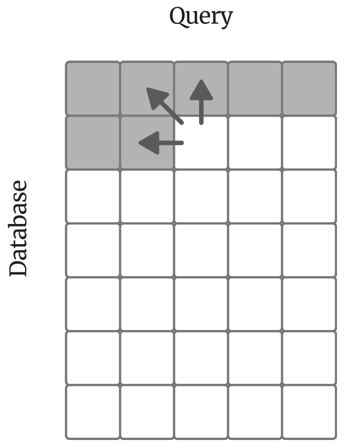

Fig. 1. Dependencies of the cell $S_{i,j}$

$$s(d_i, q_j) = \begin{cases} +2, d_i = q_j \\ -1, d_i \neq q_j \end{cases}$$

As shown in Fig. 1, this algorithm has many data dependencies during its execution, with a time complexity of $\mathcal{O}(M \times N)$, hence the need for both algorithmic and hardware optimizations. Before optimizing for the FPGA, it is important to first maximize the performance of the general-purpose processor and use that as a reference for later speedup.

III. Software

A. Parallelization

There have been some major improvements to the Smith–Waterman algorithm as summarized in [4]. The first and most important optimization, introduced by A. Wozniak [5] in 1997, is changing the access pattern of the dynamic programming matrix and using anti-diagonals (Fig. 2). This change allows for parallelization of the computation since the dependencies of each element across the anti-diagonal k are in $k-1$ and $k-2$. This modification enables multiple elements of each anti-diagonal to be computed simultaneously through vectorization and OpenMP.

B. Implementation

We have created three different versions of this algorithm for an x86 architecture CPU to act as a baseline.

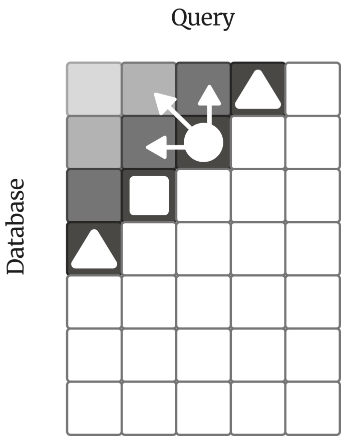

Fig. 2. Data dependencies of the anti-diagonal layout. Each shape represents a different thread.

- The basic "Naive" implementation illustrates how the algorithm works but has some major flaws such as a modulo operation, an unnecessary "if" statement that checks for the first column or row and a straightforward but inefficient way to determine element $S_{i,j}$.
- The second "Optimized" version is improved by using nested for loops and an extra row and column for padding, thus eliminating an "if" statement minimizing the time needed to determine the max value.
- The last and best version is the one that parallelizes the algorithm using OpenMP.

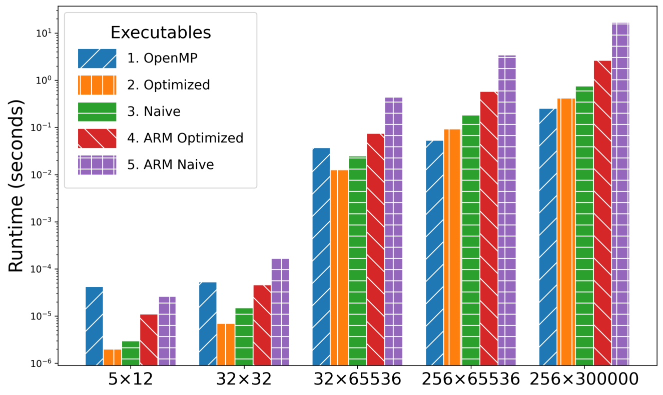

Fig. 3. Execution times for each version of the algorithm for various N and M values (on the X-axis). The runtime (Y-axis) is in logarithmic scale. The tests 1, 2 and 3 ran on an x86 12th Gen Intel® Core™ i5-1235U × 12 CPU and the tests 4 and 5 on an Dual-core Arm Cortex-A9 MPCore CPU. The OpenMP version was limited to 8 threads.

TABLE I
CPU RUNTIMES IN MILLISECONDS PER SOFTWARE IMPLEMENTATION.

$N \times M$	OpenMP	Opt.	Naive	Arm Opt.	Arm Naive
5×12	0.042	0.002	0.003	0.011	0.026
32×32	0.053	0.007	0.015	0.046	0.167
32×65536	37.179	12.634	24.960	75.002	438.036
256×65536	53.418	92.491	182.408	578.555	3439.462
256×300000	254.382	416.565	754.058	2649.575	16842.597

C. Roofline model

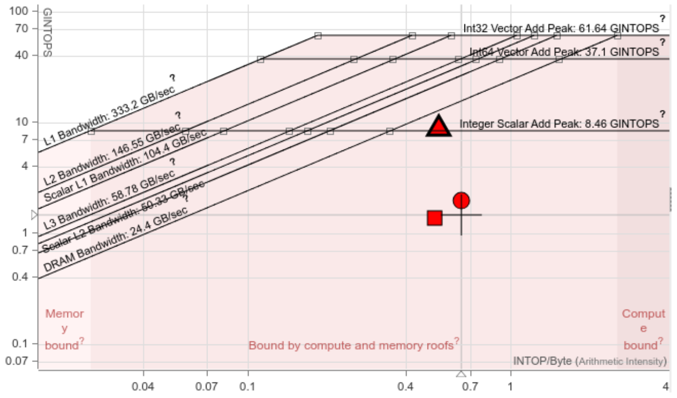

Fig. 4. Roofline model of software implementations on x86. The OpenMP version is the triangle, the optimized version is the circle and the naive is the square.

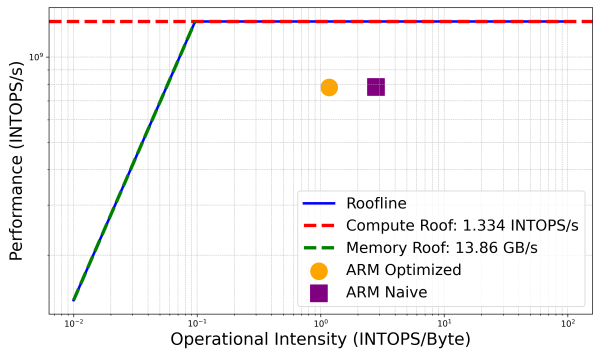

Fig. 5. Roofline model of software implementations on Arm.

IV. FPGA OPTIMIZATIONS

A. The x86 algorithm

Before making any hardware optimizations, we ran the "Optimized" algorithm from the Subsection III-B as a baseline. This was done to calculate the speedup of our improvements. In all tests regarding FPGA optimizations, we chose the values $N = 32$ and $M = 65536$.

B. Reducing Space Complexity

To backtrack the alignment path and locate regions of similarity, the only required output is the direction matrix P. The dynamic programming matrix, with space complexity of

TABLE II
FPGA BUILD REPORT OF THE X86 OPTIMIZED ALGORITHM

Latency on FPGA (ms)	4562.5097
Loop Name	OUTER_LOOP
Min Latency (cycles)	297088813
Max Latency (cycles)	435367289
Latency Range	4529 – 6637
Iteration Latency	–
Initiation Achieved	–
Initiation Target	–
Trip Count	65597
Pipelined	no

$\mathcal{O}(M \times N)$, is not required as an output. Instead of using a similarity matrix, we can only use three anti-diagonals of size $N+1$ (including the padding) to scan the matrix and determine the direction, since diagonal k depends only on diagonals $k-1$ and $k-2$. This drastically reduces the space needed for the algorithm, allowing us to load those three diagonals as buffers in faster memory on the FPGA fabric, such as BRAMs (Block RAMs). With this change, there is now a need to check if the anti-diagonal buffers are within the borders of the matrix. This is slow and can be avoided if we use padding on top and at the bottom of the matrix. Even though the additional calculations are N^2, the performance improvement by removing the check for the bounds is greater.

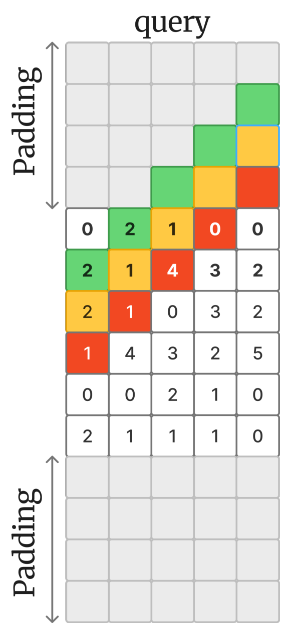

Fig. 6. Using padding and three anti-diagonals instead of the similarity matrix.

C. Unrolling & Pipelining

To fully exploit the parallelism of the algorithm's new structure, we run in parallel the inner loop that calculates the similarity along the anti-diagonal, ensuring that there are no dependencies between each $S_{i,j}$ cell. Additionally, in hardware, loops can be pipelined—meaning that the execution of one iteration doesn't need to complete before the next begins. The goal, and the biggest challenge, is to achieve the lowest possible Iteration Interval (II), ideally 1, which allows for maximum throughput. Subsequent optimizations aim to further reduce the II by maximizing memory bandwidth and removing dependencies between iterations of the outer loop.

D. Memory Access

The algorithm requires high memory bandwidth to efficiently compute the results. To meet this demand, all arrays used within the function or kernel are stored in BRAMs provided by the FPGA fabric. BRAMs offer significantly faster access—typically just 1 clock cycle per read or write—compared to the 30 or more cycles needed to fetch data from external RAM. To ensure that arrays are mapped to BRAM, an HLS directive was used to explicitly bind all local arrays to BRAM. However, BRAMs come with a limitation, they have a limited number of ports, with the best configuration (e.g. bram_t2p) offering only two read/write ports. To further improve memory bandwidth, the access patterns of all arrays were analyzed. The goal was to break down large arrays into smaller, independent BRAM blocks, allowing parallel access. For this purpose, a directive to partition the matrices across BRAMs was used. This combination of storage binding to BRAM and array partitioning enables more efficient pipelining and parallel execution, reducing memory access bottlenecks and improving overall performance.

TABLE III
ADDED BRAM BUFFERS, PIPELINING, UNROLLING AND REMOVED SIMILARITY MATRIX

Latency on FPGA (ms)	21.641
Loop Name	OUTER_LOOP
Min Latency (cycles)	2,098,215
Max Latency (cycles)	2,098,215
Iteration Latency	104
Initiation Achieved	32
Initiation Target	1
Trip Count	65,567
Pipelined	yes

E. Skewed Direction Matrix

After the computation of each diagonal, the kernel needs to transfer the direction data of cells to the main memory. This results in N separate writes to RAM, which severely bottlenecks pipelining in the hardware. To resolve this issue, the direction array which stores the direction of each cell was restructured to be written row-wise into main memory instead of in a diagonal fashion. This flattening of the array helps the compiler recognize that all the data can be written continuously in a single burst. As a result, the HLS tool is able to perform one large write operation instead of N smaller ones, significantly improving memory throughput and allowing for more efficient pipelining.

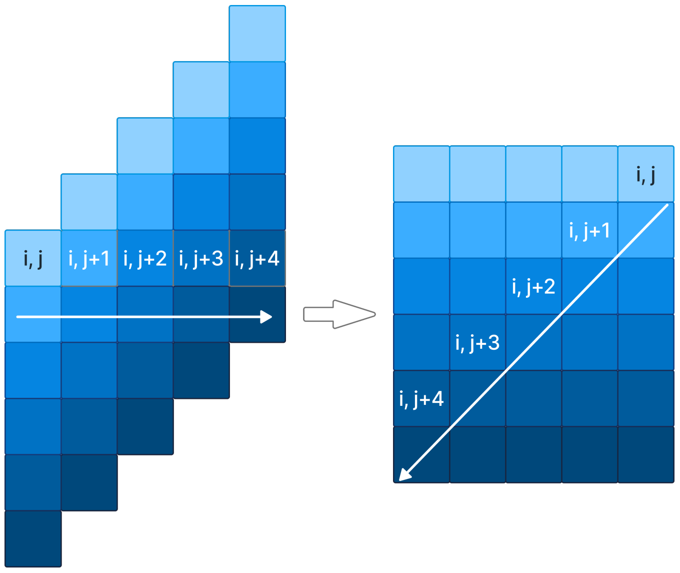

Fig. 7. Layout of skewed direction matrix.

TABLE IV
SKEWED DIRECTION MATRIX

Latency on FPGA (ms)	12.8369
Loop Name	OUTER_LOOP
Min Latency (cycles)	1,049,075
Max Latency (cycles)	1,049,075
Iteration Latency	20
Initiation Achieved	16
Initiation Target	1
Trip Count	65,567
Pipelined	yes

F. Finding the Maximum Value

At this stage of the algorithm, N modules, each computing their respective maximum values, attempt to access a shared variable that holds the global maximum. This shared access becomes a bottleneck, limiting the parallelization of diagonal computations, since multiple modules cannot simultaneously read and write to the same variable. To resolve this, an array of size N was introduced, allowing each module to access its own dedicated variable. This array can be partitioned between the BRAMs with a factor of $N/2$ since we are using two ports. This enables true parallelization, as each column's maximum value is stored independently in a separate element of the array. Once the outer loop finishes, the index with the highest value is selected from this array with minimal latency, ensuring both performance and scalability.

G. Optimizing Branching Logic

After applying the optimizations mentioned, as the Iteration Interval is decreasing, a timing violation is caused by the branching logic of the algorithm where the max value is determined. The x86 and Arm implementations calculate the max value using three consecutive “if” statements that result in poor hardware implementation. To improve this, a single “if-else-if” chain was used, allowing the HLS to implement it using multiplexers, reducing the number of cycles this operation requires.

H. Datatypes

Efficient use of data types is crucial for minimizing the area of the design and optimizing critical paths, which can become bottlenecks for pipelining in HLS. To determine the smallest possible variable sizes suitable for our algorithm, we analyzed the maximum values of the diagonal arrays with respect to the query length. Since values in these arrays only increase when there is a match between the database and the query, the maximum possible value is $2\times N$. For example, when $N = 32$, a signed character is sufficient. For $N = 64$ or greater, a short is required to safely represent all possible values. Furthermore, for the query, database and direction matrix, we use char types to save space. For the maximum index, we use an integer, since its maximum value can reach $N \times M = 32 \times 65536$, for the smallest test, which far exceeds the range of a short.

TABLE V
IMPROVED MAX FUNCTION, BRANCHES AND DATATYPES

Latency on FPGA (ms)	4.96728
Loop Name	OUTER_LOOP
Min Latency (cycles)	262,268
Max Latency (cycles)	262,268
Iteration Latency	4
Initiation Achieved	4
Initiation Target	1
Trip Count	65,567
Pipelined	yes

I. Database Buffer Shift

A major concern for the algorithm's scalability is the storage of the database array. Although small inputs allow the array to fit entirely in BRAM, this results in high utilization. For larger databases, BRAM capacity becomes insufficient. To address this, the characters of the database are shifted into a buffer of size M to a reduced size of $N + 1$. This means that for each new diagonal, the oldest character in the buffer is discarded (as it is no longer needed), and a new character is fetched from main memory. This optimization significantly reduces BRAM usage and decreases the iteration interval from 4 to 3, improving overall performance. Although fetching a new character from memory in every outer loop iteration introduces slow reads, this does not negatively affect performance because the reads are pipelined. The only trade-off is an increase in pipeline depth, which increases from 4 to 144.

TABLE VI
APPLIED DATABASE BUFFER SHIFT

Latency on FPGA (ms)	3.78776
Loop Name	OUTER_LOOP
Min Latency (cycles)	196,841
Max Latency (cycles)	196,841
Iteration Latency	144
Initiation Achieved	3
Initiation Target	1
Trip Count	65,567
Pipelined	yes

J. Diagonals Rotation

To achieve an Iteration Interval of 1, one final optimization was necessary. Initially, the diagonal arrays were rotated after the inner loops has finished execution. This caused an inter-iteration dependency between the outer loops. In the optimized version, the rotation occurs in the beginning of the next iteration of the outer loop. This change, combined with fully partitioning both the array with the maximum values (Section IV-F) and the third diagonal buffer, successfully achieved an Iteration Interval of one.

TABLE VII
APPLIED OPTIMAL ARRAY ROTATION

Latency on FPGA(ms)	3.77532
Loop Name	OUTER_LOOP
Min Latency (cycles)	65,740
Max Latency (cycles)	65,740
Iteration Latency	175
Initiation Achieved	1
Initiation Target	1
Trip Count	65,567
Pipelined	yes

V. Results

Comparing the results, we notice that the FPGA is 3.3× faster than the x86.

TABLE VIII
RUNTIME (IN MILLISECONDS) FOR EACH IMPLEMENTATION

Implementation	**Runtime (ms)**
1. x86 Optimized	12.634
2. FPGA Baseline	4562.510
3. FPGA Memory, Pipeline & Unroll	21.641
4. FPGA Skewed Direction Matrix	12.837
5. FPGA Max, Branches & Data Types	4.967
6. FPGA Database Buffer Shift	3.787
7. FPGA Optimal Array Rotation	3.775

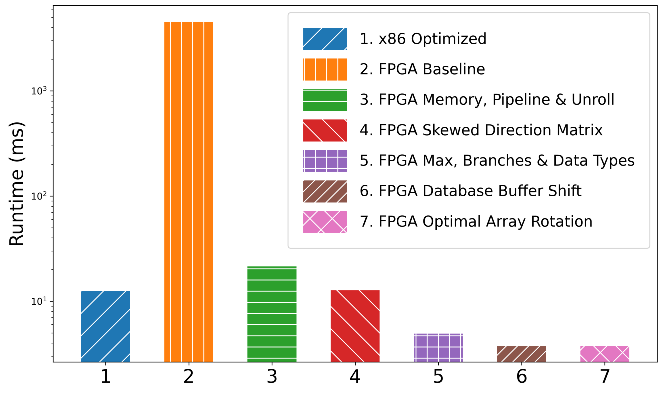

Fig. 8. Data of Table VIII - runtime comparison between the optimized x86 version and the gradual optimizations added on the FPGA. The scale of the Y-axis is logarithmic.

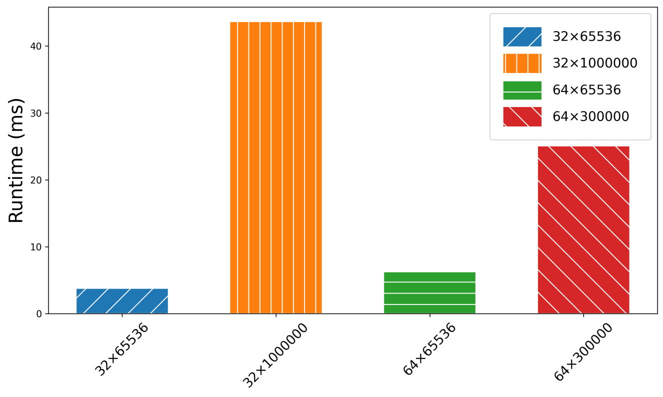

Fig. 9. Runtimes of the best FPGA version for various N and M values.

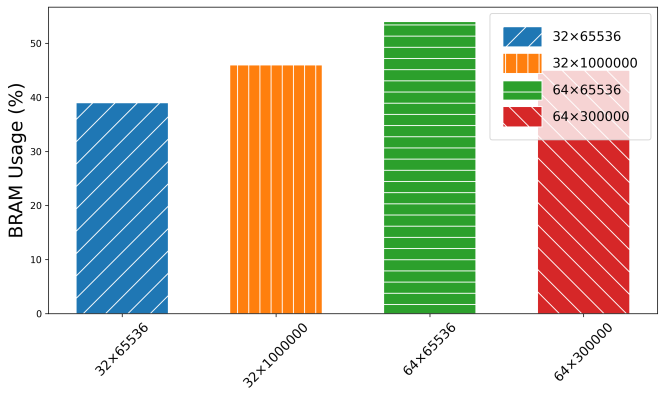

Fig. 10. Block RAM usage percentage

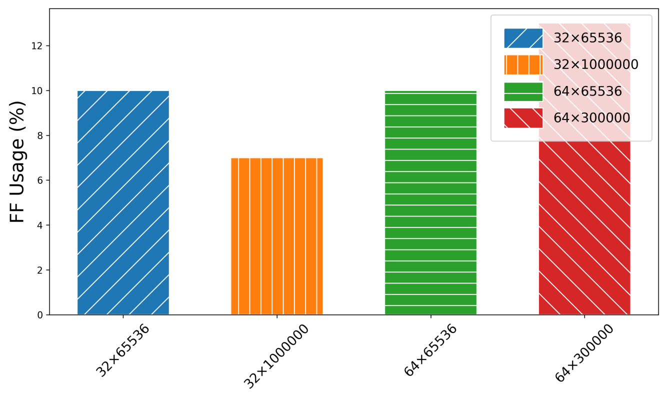

Fig. 11. Flip Flop usage percentage

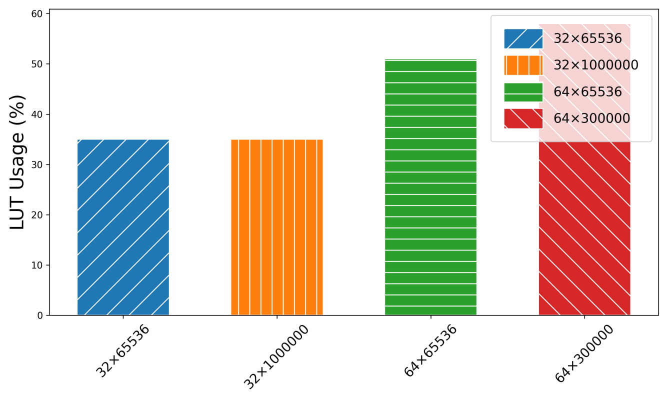

Fig. 12. Look Up Tables usage percentage

References

[1] David W. Mount. *Bioinformatics: Sequence and Genome Analysis*. 2nd ed. Cold Spring Harbor, NY: Cold Spring Harbor Laboratory Press, 2004, pp. 53–58.

[2] SAUL B. NEEDLEMAN and CHRISTIAN D. WUNSCH. "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins". In: *Molecular Biology*. Ed. by S. Brenner. London: Academic Press, 1989, pp. 453–463. ISBN: 978-0-12-131200-8. DOI: https://doi.org/10.1016/B978-0-12-131200-8.50031-9. URL: https://www.sciencedirect.com/science/article/pii/B9780121312008500319.

[3] T. F. Smith and M. S. Waterman. "Identification of common molecular subsequences". In: *Journal of Molecular Biology* 147.1 (1981), pp. 195–197. DOI: 10.1016/0022-2836(81)90087-5.

[4] Zeyu Xia et al. "A Review of Parallel Implementations for the Smith–Waterman Algorithm". In: *Interdisciplinary Sciences: Computational Life Sciences* 14 (Sept. 2021). DOI: 10.1007/s12539-021-00473-0.

[5] A. Wozniak. "Using video-oriented instructions to speed up sequence comparison". In: *Bioinformatics* 13.2 (Apr. 1997), pp. 145–150. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/13.2.145. eprint: https://academic.oup.com/bioinformatics/article-pdf/13/2/145/689453/13-2-145.pdf. URL: https://doi.org/10.1093/bioinformatics/13.2.145.