FPGA IMPLEMENTATION AND OPTIMIZATION OF LSAL ALGORITHM

Christos Karagiannis, Athanasios Gerampinis

Abstract—The Smith-Waterman algorithm is a fundamental technique in bioinformatics for performing local sequence alignment, enabling the identification of highly similar regions within DNA, RNA, or protein sequences. Despite its accuracy, the algorithm's computational complexity presents a significant challenge for real-time and large-scale analysis. In this work, we present a hardware accelerator tailored for the Smith-Waterman algorithm, designed to exploit parallelism and data locality to achieve high-throughput and low-latency alignment.

Index Terms—Smith Waterman, FPGA optimization

1 Introduction

Biological sequence alignment is a core task in computational biology, where researchers compare DNA, RNA, or protein sequences to identify regions of similarity. Such similarities may indicate functional, structural, or evolutionary relationships. Alignment algorithms are generally classified into global and local alignment methods. Global alignment algorithms, such as Needleman-Wunsch, compare sequences from end to end, assuming complete similarity. In contrast, local alignment, exemplified by the Smith-Waterman algorithm, identifies the most similar subregions within sequences, allowing it to detect conserved motifs or domains within otherwise divergent genetic material.

2 Literature Review

The Smith-Waterman algorithm has been extensively studied due to its critical role in bioinformatics applications such as database search, genome assembly, and variant detection. Early implementations focused on CPU-based solutions, including SIMD-accelerated versions using SSE and AVX instruction sets [1] to exploit data-level parallelism. These methods, while effective for small datasets, struggle with scalability and power consumption. GPU-based implementations have also emerged that leverage thousands of cores to accelerate alignment tasks; however, memory bandwidth limitations and irregular data dependencies limit performance. More recently, FPGA-based accelerators [2] have gained attention for their ability to achieve high throughput with low power consumption.

3 The Algorithm

The algorithm that we are studying, Smith Waterman [1], is applying Local Sequence Alignment, a technique in bioinformatics used to identify regions of similarity between two biological sequences (DNA, RNA, or protein). Unlike global alignment (where we use the Needleman-Wunsch algorithm), which aligns sequences from start to end, local alignment focuses on finding the most similar subsequences within the two sequences. This is useful when the sequences are partially similar or contain conserved regions within otherwise different sequences.

The Smith-Waterman algorithm is an algorithm that uses dynamic programming. It finds the optimal local alignment between two sequences by building a scoring matrix and then performing a traceback to identify the best scoring subsequence alignment.

How It Works

Let $\mathbf{d} = d_1 d_2 \dots d_m$ be the first sequence of length M (database) and $\mathbf{q} = q_1 q_2 \dots q_n$ be the second sequence of length N (query).

Create a similarity matrix S

S_{ij}: the similarity score at cell (i, j) in the similarity matrix $(0 \leq i \leq M, 0 \leq j \leq N)$

- The rows represent characters of the database sequence $\mathbf{d}$.
- The columns represent characters of the query sequence $\mathbf{q}$.

Initialize the first row and column to zero (Base Case):

$$S_{0j} = 0, \forall j \quad S_{i0} = 0, \forall i$$

Scoring

The rest of the similarity matrix elements are defined as:

$$S_{ij} = \max \begin{cases} 0 \\ S_{i-1,j-1} + s(q_i, d_j) \\ \max_{k \geq 1}\{S_{i-k,j} - W_k\} \\ \max_{l \geq 1}\{S_{i,j-l} - W_l\} \end{cases}$$

where:

- $1 \leq i \leq M, 1 \leq j \leq N$
- $S_{i-1,j-1} + s(q_i, d_j)$ is the score of aligning q_i and d_j:

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

- $\max_{k \geq 1}\{S_{i-k,j} - W_k\}$ is the score if d_i is at the end of a gap of length k (Deletion from **d**),
- $\max_{l \geq 1}\{S_{i,j-l} - W_l\}$ is the score if q_j is at the end of a gap of length j (Insertion to **q**),
- 0 means there is no similarity up to d_i and q_j.

We use a linear gap penalty that has the same scores for opening and extending a gap: $W_k = kW_1$ where W_1 is the cost of a single gap (most of the times it is equal to 1).

Thus the formula can be simplified to:

$$S_{ij} = \max \begin{cases} 0 & \\ S_{i-1,j-1} + s(q_i, d_j) & \text{(Match/Mismatch)} \\ S_{i-1,j} - 1 & \text{(Deletion)} \\ S_{i,j-1} - 1 & \text{(Insertion)} \end{cases}$$

Traceback

In order to calcualate the final aligned subsequences in **d** and **q** we perform the traceback step. First we start from the position with the highest value in the similarity matrix. Then we trace back diagonally (match/mismatch) or up/left (gap), until a zero is reached. This gives the highest-scoring local alignment.

In order to determine when to move diagonally, up or down, we need to know if the current similarity score came from a match/mismatch or from a gap, either from **q** or from **d**. Thus a direction matrix is used, which records how each cell in the similarity matrix was computed.

Each element D_{ij} is determined as follows. If S_{ij} came from:

- $S_{i-1,j-1} + s(q_i, d_j)$, we store 'D' (Diagonal),
- $S_{i-1,j} - 1$, we store 'U' (Up),
- $S_{i,j-1} - 1$, we store 'L' (Left),
- otherwise we store '-' (Stop)

Can it be optimized?

It is evident that the most computationaly expensive part of this algorithm is the computation of the similarity and direction matrices. Not only because of the time complexity of $O(M \times N)$, but also due to strong data dependencies that prevent the algorithm from being naively paralellizable. These dependencies introduce a wavefront pattern. That means we cannot compute a cell until the three neighbors it depends on have been computed.

Nevertheless, parallelization is possible, but only along anti-diagonals, where all elements in a given diagonal can be computed in parallel and only when the previous diagonal is completely computed:

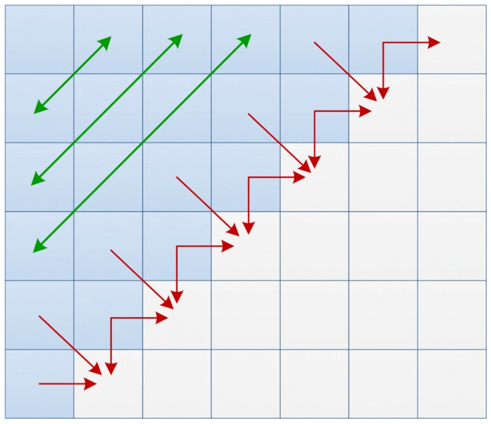


Figure 1: Data dependences in the similarity matrix (in red) and cells that can be computed simultaneously (in green). [3]

Of course the algorithm cannot be fully parallelized, but now the number of steps required to execute it are equal to the number of anti-diagonals $(M + N - 1)$, assuming that we have $P \geq \min(M, N)$ available computational units. So now, via fine-grained parallelism we can drop the time complexity down to $O(M + N)$. Such implementations can be massively parallelized with APIs like OpenMP, in the GPU with CUDA or even in FPGAs, with a low-latency and deeply pipelined architecture for diagonal patterns and predictable memory access patterns. This tells us that the need for hardware acceleration in this scenario is strong.

SOFTWARE IMPLEMENTATION

Before moving into heavy optimizations with the Zedboard we have of course to develop and test the code for executing the local sequence alignment algorithm. We are also required to make different implementations and aproaches, mainly for the computational part (the traceback step is fairly standard) and benchmark those implementations in different platforms: Our x86 personal computer and Zedboard's ARM CPU.

We have 3 variations of the function that computes the similarity and direction matrices:

- **lsal_compute_matrices_u** – The unoptimized version that walks through it with a single for loop. Each iteration recomputes the current row and column indices from the index, then evaluates the diagonal (`D`), up (`U`), and left (`L`) candidate scores. The winning score is written to the similarity array; the corresponding move is stored in the direction matrix. Because the loop touches memory in strict row-major order, data dependencies are naturally satisfied, but the repeated division/modulo and the branch-full max logic trade a bit of arithmetic overhead for the convenience of a single tight loop. The function also tracks the global maximum score and its location.
- **lsal_compute_matrices_o** – Here the computation is laid out in the usual two-nested-loop style. The outer loop iterates over database rows, the inner over query columns. The logic is the same but this layout avoids the division/modulo arithmetic of the unified loop and computes the similarity and direction values for the current cell at the same time, reducing the number of branches. Though it offers no built-in parallelism beyond what the compiler might automatically vectorize along the inner loop.
- **lsal_compute_matrices_p** – This variant restructures the iteration to follow successive anti-diagonals (rounds) of the similairity matrix—exactly the pattern required for LSAL diagonal parallelism. The outer loop round runs from 0 to M + N 2; the inner loop enumerates cells on that anti-diagonal by choosing the row and computing the column. Because every cell on a given diagonal depends only on cells from the 2 previous diagonals, the body of the inner loop can, in principle, be executed in parallel. The result is a compute pattern well-suited to hardware acceleration.

We've also implemented code for the traceback step, for printing the matrices (if they're not too big) and timing the execution of each version for matrices with different sizes.

x86

Below are the charts with the execution time for the different versions of the algorithm, running on the x86 CPU of one of our laptops:

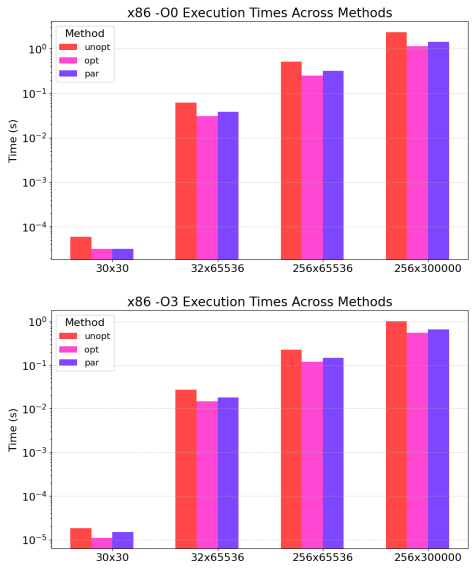


Figure 2: Execution time without (top) and with (bottom) compiler optimizations for different input sizes.

We can see that the fastest implementation is consistently the 2nd while the 3rd remains always close to it.

Roofline Analysis

An essential part of our research is evaluating how our software performs on the target hardware. To achieve this, we rely on the Roofline model—a visual framework that provides insights into the balance between computational intensity and memory usage in our implementation. By combining theoretical modeling with commercial profiling tools, the Roofline diagram helps us identify performance bottlenecks and determine whether they stem from limited computational throughput or memory bandwidth. This understanding is crucial for guiding our optimization strategies and improving overall efficiency.

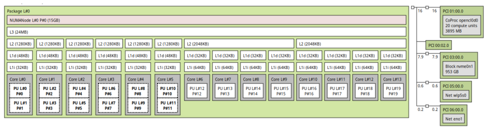


Figure 3: Intel's I7 12-700H Alder Lake architecture.

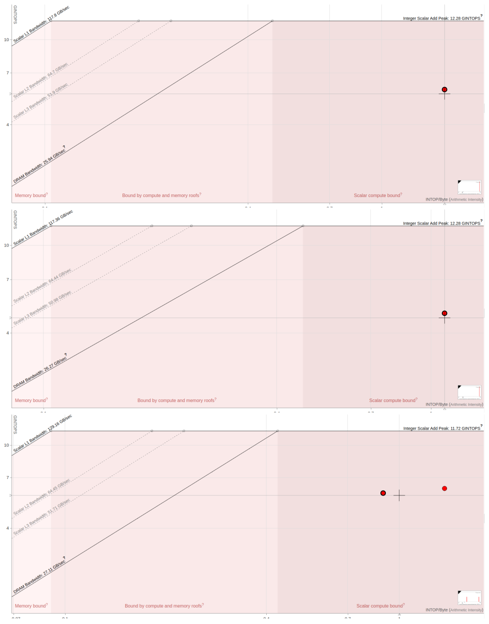


Figure 5: Roofline plots for unoptimized (top), optimized (middle), and parallel (bottom) code using Intel's Advisor software [4].

As demonstrated in our analysis for the selected input sizes, memory bandwidth does not appear to be a limiting factor. Despite the processor's strong computational capabilities, performance is constrained by the presence of significant data dependencies in the algorithm. These dependencies hinder parallelism and prevent the implementation from reaching higher throughput, highlighting a key area for future optimization.

ARM

Below are the charts for the same experiment, running on the Zedboard's Dual-Core ARM Cortex-A9 processor:

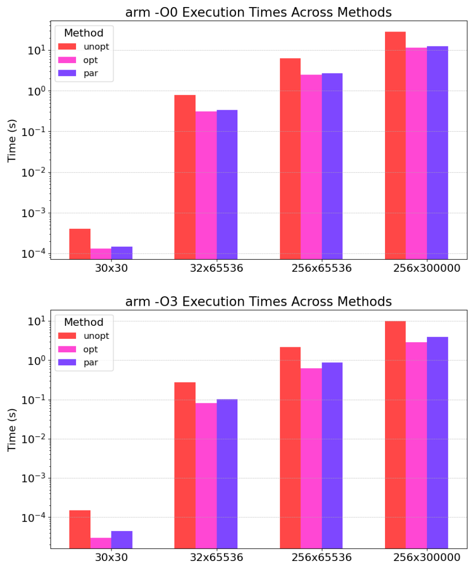


Figure 6: Execution time without (top) and with (bottom) compiler optimizations for different input sizes.

We can see that the results are proportionally similar to the ones shown for x86 but around 10 times slower. For the record, below are some charts that compare the results between the 2 platforms:

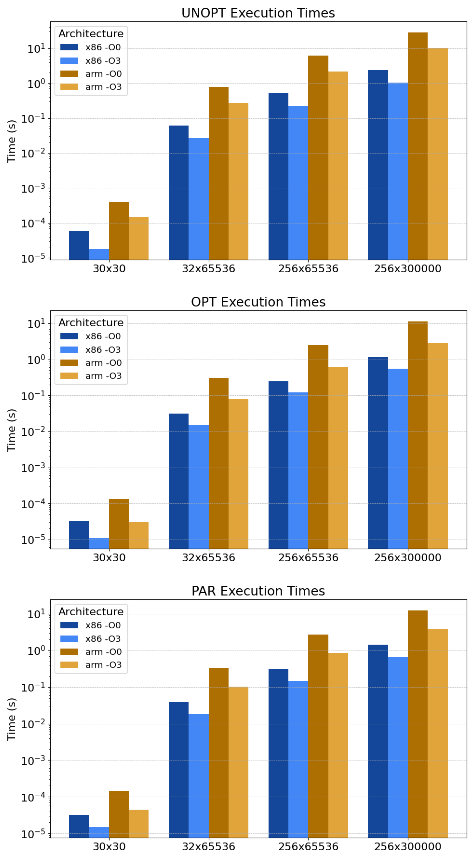


Figure 7: Execution times of x86/arm cpus for unoptimized (top), optimized (middle) and parallel code (bottom).

Roofline Analysis

Since no dedicated tools are available for constructing a Roofline model specifically for the ARM Cortex processor used in our setup, we manually extracted the necessary performance metrics from the manufacturer's documentation. According to the specifications, the CPU features two cores with dual-issue [5], out-of-order execution capabilities for scalar arithmetic operations [6]. Running at 667 MHz, the processor delivers a theoretical peak performance of 2.668 GFLOPS across both cores. However, since our implementation is designed to utilize only a single core, we consider a peak performance of 1.334 GFLOPS in our analysis.

Furthermore, the ZedBoard's Technical Reference Manual reports a DDR memory bandwidth of up to 4,264 MB/s [7]. These two key parameters—compute throughput and memory bandwidth—form the foundation for constructing a theoretical, yet insightful, Roofline model. While this model is relatively simple, it is extremely useful in highlighting the performance bottlenecks of our implementation and guiding targeted optimization strategies, particularly on a low-power embedded processor.

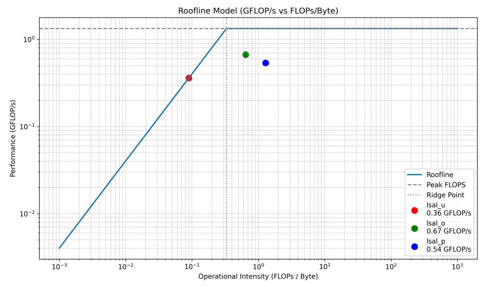


Figure 13: Roofline Model for Cortex A9 processor [8].

By analyzing the Arithmetic Intensity of each code version and plotting their positions on the Roofline model, we observed that memory performance plays a critical role in achieving high efficiency. These findings underscore the importance of memory-aware algorithm design in optimizing performance on resource-constrained platforms.

4 HARDWARE IMPLEMENTATION

Baseline implementation results

By using Vitis HLS [9], we start our attempts at optimizing the LSAL algorithm by simply copy-pasting our optimized version of the software implementation, to see some initial estimations about performance and resource usage.

We generally work with sizes of $N = 32$ and $M = 65536$. We hit C synthesis and receive the initial report:

	Top Module	Main Loop
Latency (cycles)	755236935	755236935
Latency (ms)	7.552E9	7.552E9
Iteration Latency	-	15524
Interval	755236935	-
Trip Count	-	65536
Pipelined	no	no
BRAM (%)	10	-
FF (%)	95	-
LUT (%)	572	-

Our major goal for the hardware model was to beat the execution times for x86 and then try to make it even faster. As we can see, the initial model is far from satisfactory. Let's elaborate on several techniques we used to achieve our goal.

Our optimizations

Adding diagonal parallelism & Pipelining

The baseline LSAL implementation processes matrix cells row-by-row, inherently limiting data dependencies to row/column neighbors. In our hardware design, we can introduce our approach from the 3rd software implementation and exploit diagonal parallelism by computing anti-diagonals, at each iteration, in a pipelined fashion. This eliminates long dependency chains and exposes a natural opportunity for loop pipelining, which you explicitly enable with `#pragma HLS PIPELINE`. Pipelining allows new iterations of the loop to begin every clock cycle, even before previous ones finish, dramatically increasing throughput.

In loop pipelining, the Iteration Interval (II) is the number of clock cycles between the start of consecutive loop iterations in a pipelined loop. An II=n means a new iteration starts every n cycles, so, naturally performance increases as N degrades. In pipelined designs, we aim for maximum throughput. Thus, an II=1 is desirable, something that we are not yet able to obtain, due to computational and memory handicaps.

Adding array partitioned buffers

In order to avoid direct reads/writes from the dynamic memory, we partitioned critical arrays like `q_buf`, `d_buf`, `dir_buf`, and the similarity buffers across all dimensions using `#pragma HLS ARRAY_PARTITION`. This transforms these arrays from monolithic RAMs into fully parallel registers, allowing constant-time access to every element in the same cycle. It enables full parallelism in the inner loop and ensures that operations like comparisons and updates across the query/database characters can happen simultaneously. Without this, even a simple loop with $N = 32$ would serialize memory access and stall the pipeline.

For small arrays like `q_buf` (size of N) we are able to perform a complete array partition which corresponds to decomposing the array into individual registers. For a buffer like `d_buf` with a size of $M >> N$ we are forced to use other types of array partitioning like cyclic, which creates smaller arrays by interleaving elements from the original array in a round-robin fashion. In this case we are partitioning with a factor of 16 which means that we are breaking the database in chunks of size $M/16$ across different BRAMs.

Adding AXI memory interfaces

We added AXI4 slave interfaces (m_axi) for the query, database, similarity and direction matrices, and an AXI-lite interface (s_axilite) for the max index pointer. This enables the hardware function to directly read and write data from external DDR memory in an SoC or FPGA environment. It allows the accelerator to operate independently of the CPU, fetching data via DMA and writing results back to memory, making the design suitable for integration in various standard memory architectures. Furthermore, we've bundled the query, database etc. ports separately to allow concurrent & high-bandwidth memory access.

Removing similarity matrix / Adding similarity buffers

Since we don't realy need the similarity matrix to create the final alignment via traceback, we simply remove it from the argument list. Also rather than handling the similarity scores with a complete buffer (of size $M \times N$ we replaced it with sliding 1D buffers: `buf_prev_1`, `buf_prev_2`, and `buf_curr`, which represent only the necessary parts of the previous and current rows. This massively reduces memory usage from $M \times N$ to $3 \times N$ and leverages temporal reuse. It also significantly improves locality and supports pipelined execution by further droping the II, as we're only ever reading/writing to a small number of registers.

Reshaping the direction matrix

In the original version, the direction matrix was a 2D array matching the similarity matrix size. However, that means that whenever a new anti-diagonal is computed inside the direction buffer (`dir_buf`), those elements have to be written in the correct cells of the corresponding anti-diagonal in the direction matrix. Those kind of memory accesses are problematic. To solve this, we reshape the direction matrix so that each anti-diagonal is now mapped to a different row:

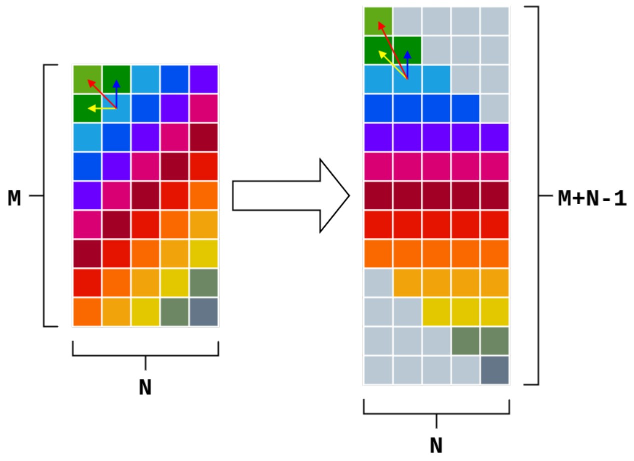


Figure 8: Transforming the direction matrix for optimized memory access pattern.

Notice that by reshaping the array, the new size is $N \times (M + N - 1)$, which slightly increases the size, however, it is accepted in general that $M >> N$ so we don't have a very big problem here. The real issue here appears with those "dead" triangular regions at the top-right and bottom-left of the matrix, which do not correspond to a pair of query-database values (since the index for traversing the database in an anti-diagonal is `row - col` and, inside that region, it gets out-of-bounds). To solve this issue, we prepare the database sequence by adding paddings of size $N - 1$ at the start and end of the sequence. The paddings consist of an invalid symbol (in this case 'X') which cannot be matched with the other valid 'A', 'T', 'G', 'C' symbols. This also eliminates the need to add checking for out-of-bounds regions, which could increase the number of compare operations and make pipelining harder.

Generaly, this technique helps to linearize and stream the direction information into coherent parts of the direction matrix. Streaming writes reduce memory footprint and align with how hardware prefers burst-accessible linear memory.

After the above impovements, we got the following results when running synthesis:

	Top Module	Main Loop
Latency (cycles)	166595	131137
Latency (ms)	1.666E6	1.311E6
Iteration Latency	-	6
Interval	166596	2
Trip Count	-	65567
Pipelined	no	yes
BRAM (%)	28	-
FF (%)	13	-
LUT (%)	47	-

Making the database buffer a shift register

Instead of repeatedly accessing the database via a huge buffer stored in different banks, we implemented `d_buf` as a shift register — a local N-sized array where each new character is shifted in from the right, and the oldest value is dropped. This creates a sliding window, reusing previous characters while minimizing external memory reads. Combined with complete partitioning, this structure supports per-cycle data updates and full-speed streaming from the input buffer.

Bitwidth Optimization

We've used `ap_int` types for all similarity buffers and other buffers and temporary values. Since the LSAL algorithm only accumulates modest integer values (especially with small match/mismatch scores), you don't need full 32-bit integers. This saves logic resources, reduces BRAM usage, and most importantly reduces critical paths, making piplining with II=1 easier, thus improving overall latency and power efficiency. Additionally, using `char` instead of `int` for sequence data further minimizes unnecessary data width, which is particularly beneficial when transferred over AXI buses.

For example, for $N = 32$ and $M >> N$, the maximum possible similarity score occurs when the entire query sequence is matched to the database sequence, thus it is equal to $MATCH \times N = 2 \times 32 = 64$. That means that we cound even use `ap_int<7>` types for the similarity buffers.

4.0.1 Minor Improvements

- **Efficient max search** - We've added 2 new buffers, one for storing the maximum similarity score for each column and one for storing the row in which the score is located. As a final step we've added a fully unrolled loop by using `#pragma HLS UNROLL`, that accumulates the max search data to find the global maximum index. This method relieves the main pipelined loop from extra operations and improves overall latency.
- **Avoiding branching in scoring** - By introducing one extra element in `buf_prev_1` and `buf_prev_2` buffers (Now sized $N-1$) that works as a dummy cell set to 0, we avoid using conditional logic for checking if certain neighboring values exist, in the case of calculating the first cell of each row (or anti-diagonal). This again helps at minimizing latency and LUT usage.

Our final synthesis report is the following:

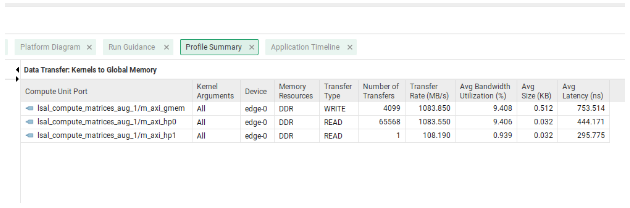
	Top Module	Main Loop
Latency (cycles)	65798	65640
Latency (ms)	6.580E5	6.560E5
Iteration Latency	-	75
Interval	65799	1
Trip Count	-	65567
Pipelined	no	yes
BRAM (%)	17	-
FF (%)	6	-
LUT (%)	26	-

Running on the FPGA

Ultimately we ran 2 versions of the algorithm on the FPGA. Both versions ran with sizes $N = 32$ and $M = 65536$. When increasing N the number of LUTs, FFs and BRAMs required increased dramatically and that posed a significant issue when generating the bitstream.

- When we first reached II=2 and a latency of 1.3ms, we tried to generate the bitstream but we got into problems. An error occurred that suggested that there was a slack between the BRAM and the direction buffer when performing reads. To solve this we reduced the cyclic partition factor from 32 to 16. When we were able to get the bitstream we ran the algorithm at a time of **4.8ms**. We also ran the software implementation of the algorithm and compared the traceback results, which seemed to be the same.
- When reaching our final results with II=1 and latency of 0.6ms, we got a similar error when generating the bitstream, this time having to do with a slack in the similarity buffers. This is when we added the dummy cell at the start of the previous similarity

buffers (one of the minor improvements that we discused earlier). At the end we were able to get the bitstream and execute the algorithm at a time of **3.7ms**. The results were correct, logicwise.



Data Transfer: Kernels to Global Memory

Compute Unit Port	Kernel Arguments	Device	Memory Resources	Transfer Type	Number of Transfers	Transfer Rate (MB/s)	Avg Bandwidth Utilization (%)	Avg Size (KB)	Avg Latency (ns)
lsal_compute_matrices_aug_1/m_axi_gmem	All	edge-0	DDR	WRITE	4099	1083.850	9.408	0.512	753.514
lsal_compute_matrices_aug_1/m_axi_hp0	All	edge-0	DDR	READ	65568	1083.550	9.406	0.032	444.171
lsal_compute_matrices_aug_1/m_axi_hp1	All	edge-0	DDR	READ	1	108.190	0.939	0.032	295.775

Figure 9: Data Transfer Profiling from Vitis Analyzer.

Future Improvements

The results so far were satisfactory, however the difference between the 0.6ms, that Vitis HLS gave us, and the 3.7ms, that was the real execution time, is troubling.

It's safe to say that our model, computationaly, is at best, since the latency of the algorithm in clock cycles approximately matches the time complexity of the fine-grained parallel algorithm ($M + N$). The difference between the estimated and the real time could be explained by poor memory usage. Besides, when passing our model from Vitis Analyzer, we got results that showed a very high number of transactions and a tiny bandwidth utilization.

One good way of maximizing the memory throughput is to take advantage of the 512 bit length of the AXI bus between the DDR memory of the Zedboard and the internal BRAM of the FPGA. This is possible by using the arbitrary data types in a much more clever way than we are currently using them (specifically `ap_uint<512>` for the query and database sequences and the direction matrix). Then we'll be able to read and write much more efficientlly and in a way that makes more sense for the hardware.

This and many other minor ideas for optimizations were considered and tested but, due to many reasons, were left to be implemented for another time in the future.

5 Conclusion

In this project, we successfully explored the design, implementation, and optimization of a hardware-accelerated sequence alignment system on FPGA using HLS tools. Through a series of experiments, we analyzed the performance limitations and potential of our solution, identifying key factors such as memory bandwidth utilization and parallelism granularity. Although certain advanced optimizations—like more effective usage of wide AXI interfaces—remain as future work, our results demonstrate a clear path toward further performance gains. The insights and tools developed during this work provide a solid foundation for ongoing improvements and practical deployments of FPGA-based bioinformatics solutions.

References

[1] Wikipedia contributors, "Smith–waterman algorithm — wikipedia, the free encyclopedia," https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm, 2024, accessed: 2025-06-13. [Online]. Available: https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm

[2] B. Strengholt and M. Brobbel, "Acceleration of the smith–waterman algorithm for dna sequence alignment using an fpga platform," https://liacs.leidenuniv.nl/~bakkerem2/cmb2014/20130627__Strengholt_Brobbel__Acceleration_of_the_Smith-Waterman_algorithm_for_DNA_sequence_alignment_using_an_FPG.pdf, Leiden University, LIACS, Technical Report / Master's Thesis, Jun. 2013, artix-7 XC7A200T implementation; 94GCU/s. [Online]. Available: https://liacs.leidenuniv.nl/~bakkerem2/cmb2014/20130627__Strengholt_Brobbel__Acceleration_of_the_Smith-Waterman_algorithm_for_DNA_sequence_alignment_using_an_FPG.pdf

[3] E. Rucci, C. Garcia, G. Botella, A. De Giusti, M. Naiouf, M. Prieto-Matias, and et al., "Swifold: Smith-waterman implementation on fpga with opencl for long dna sequences," *BMC Systems Biology*, vol. 12, no. Supplement5, p. 96, 2018. [Online]. Available: https://bmcsystbiol.biomedcentral.com/articles/10.1186/s12918-018-0614-6

[4] Intel Developer Products, "Intel® Advisor User Guide," https://www.intel.com/content/dam/develop/external/us/en/documents/advisor-user-guide.pdf, 2023, accessed: 2025-06-13.

[5] Arm Developer, "Cortex-A9 Processor," https://developer.arm.com/Processors/Cortex-A9, accessed: 2025-06-13.

[6] Design and Reuse, "The arm cortex-a9 processors," https://www.design-reuse.com/article/58929-the-arm-cortex-a9-processors/, 2023, accessed: 2025-06-13.

[7] Xilinx, "Zynq-7000-All-Programmable-SoCTechnical-Reference-Manual," https://docs.rs-online.com/993c/0900766b81624496.pdf, accessed: 2025-06-13.

[8] D. Mishra, "A comprehensive guide to the roofline model," Medium (Nerd For Tech), Dec. 2023, accessed: 2025-06-13. [Online]. Available: https://medium.com/nerd-for-tech/a-comprehensive-guide-to-the-roofline-model-fddaa506ce2b

[9] Xilinx Inc., *Vitis High-Level Synthesis User Guide*, https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug1399-vitis-hls.pdf, Xilinx, 2022, uG1399 (v2022.2). [Online]. Available: https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug1399-vitis-hls.pdf