Local Sequence Alignment Algorithm FPGA Implementation*

*Note: ECE340 Lab3 Accelerator on FPGA

1st Panagiotis Nikolaos Tsogkas
University of Thessaly

2nd Jason Nikas
University of Thessaly

***Abstract*—This report explores the acceleration of the Smith-Waterman algorithm for local sequence alignment using FPGA hardware. Performance comparisons are made between optimized x86 software implementations and an FPGA-based solution, highlighting architectural trade-offs, optimization strategies, and measured speedups across various input sizes.**

***Index Terms*—Smith-Waterman, FPGA acceleration, local sequence alignment, hardware optimization, performance profiling**

I. Introduction

This report will present the acceleration of the already known Local Sequence Alignment Algorithm (LSAL), implemented on an FPGA, in contrast to its x86 counterpart.

II. LSAL Algorithm

The Local Sequence Alignment Algorithm (LSAL), most notably represented by the Smith-Waterman algorithm, is a dynamic programming approach used to identify the most similar local region between two biological sequences, such as DNA, RNA, or proteins. Unlike global alignment algorithms, which attempt to align entire sequences from start to end, LSAL focuses on determining the optimal alignment of substrings, making it particularly useful for detecting conserved motifs or domains within longer sequences.

III. Optimizations

A. Data Dependencies

The Smith-Waterman algorithm exhibits well-defined but non-trivial data dependencies due to the structure of its dynamic programming recurrence. Specifically, the computation of each cell in the similarity matrix $H(i, j)$ depends on three previously computed neighboring cells: $H(i-1, j-1)$ (diagonal), $H(i-1, j)$ (up), and $H(i, j-1)$ (left), as shown in Fig 1.

This dependency pattern enforces a strict partial ordering of computations, which constrains parallel execution. A given cell cannot be computed until all three of its dependencies have been resolved. As a result, the algorithm cannot be parallelized across rows or columns directly without violating correctness.

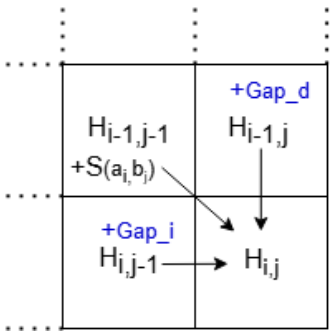


Fig. 1. Scoring method of the Smith–Waterman algorithm

B. Optimization Flow

a) Software Optimization Flow:

- Iteration I: This iteration improves the baseline implementation by applying general optimizations that include, using two nested for loops instead of one to avoid division in each iteration, reusing data where possible and adding one row and column of padding to the matrices to avoid checking for edge cases. Additionally, the design uses basic C instructions with no regard for underlying assembly, or cache locality optimizations.
- Iteration II: The second iteration of the code leverages the x86 `CMOV` instruction, by subtly guiding the GCC compiler through the use of ternary operators, as demonstrated in Appendix I. This optimization had a major performance improvement for the x86 architecture, which can be seen under section x86 Execution.
- Iteration III: The 'Unexplainable' iteration. In this version of the code, we modified the order in which the `CMOV` instructions, responsible for computing the maximum value across all entries, are executed. This adjustment resulted in a noticeable performance improvement, whose cause remains unclear. The generated assembly does not indicate any substantial change to the fundamental structure or logic of the function. It is likely that the observed behavior is influenced by the inner-workings of `CMOV` at the hardware level and how the CPU internally processes such operations.
- Iteration IV: This variation of the source code performs

a targeted optimization that is applicable to the naive algorithmic approach, which traverses the matrix row by row. The optimization involves forwarding data for the evaluation of the next entry. This method exploits the inherit data dependency shown in Fig 1 of neighboring entries as presented in Fig 2, thereby reducing the number of accesses to the similarity matrix. Specifically, the `N` value of a given cell corresponds to the `NW` value of the subsequent cell, while the computed value of the current cell becomes the `W` value for the next cell.

- Iteration V: This version seeks to enhance performance by pre-fetching data and minimizing cache misses, which would otherwise necessitate accessing the main memory (RAM). Additionally, the use of the restrict keyword in C informs the compiler that the specified pointer will not be referenced by any other pointers, thereby enabling more efficient decisions regarding vectorization. As shown in the results under Section x86 Execution, compiler vectorization was enabled; however, no performance improvement was observed.
- Iteration VI: The diagonal traversal through the similarity matrix offers the advantage of eliminating dependencies between adjacent cells, a topic that will be explored in further detail in Section Hardware - Algorithm Optimization Flow. However, this approach incurs a trade-off in the form of poor cache locality, as illustrated in Fig 3. This issue is particularly pronounced in systems with smaller caches, such as the Arm CPU inside the FPGA. Consequently, unless the parallelization potential of this method is fully exploited, it is expected to underperform relative to the naive approach.

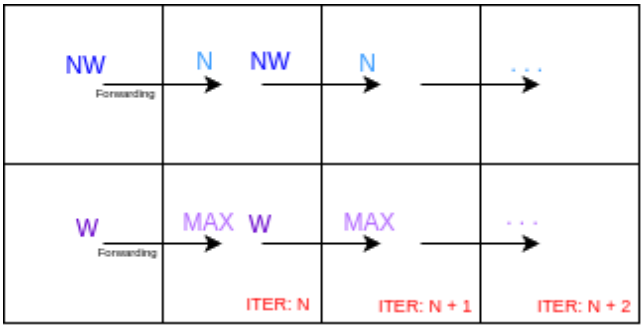


Fig. 2. Naive version forwarding logic

b) Initial Hardware - Algorithm Optimization Flow:

- The aforementioned dependencies restrict parallelization across rows or columns. However, these same dependencies enable a sequential ordering along anti-diagonals. As illustrated in Fig. 1, the computation of any entry $H(i, j)$ depends solely on elements located in the upper-left region of the matrix. This observation permits sequential evaluation of the similarity matrix anti-diagonals and parallel evaluation of all elements within it, which is particularly advantageous and a critical step towards reducing runtime in the FPGA implementation.

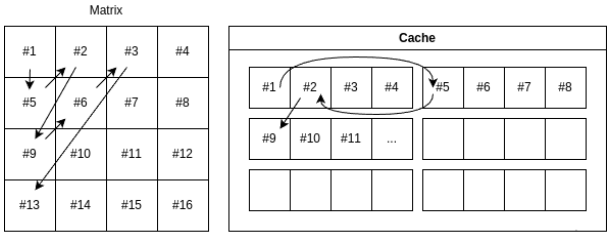


Fig. 3. Cache access pattern for anti-diagonal traversal.

- The baseline implementation of the algorithm utilizes a full similarity matrix of dimensions $N \times M$, where each entry $H(i, j)$ is computed based on the values of its neighboring cells. However, by adopting the anti-diagonal traversal strategy, we can effectively reduce the size of the matrix to $3 \times N$, since the computation of a given anti-diagonal depends only on the two preceding anti-diagonals. We will explore more the implementation of this anti-diagonal buffer in subsections Starting Version, Hardware Bottleneck Optimizations.
- Before parallelizing each anti-diagonal it is essential to ensure that no dependencies exist between cells. The evaluation of the max index variable is of such kind. To address this, we utilize a buffer of size N to temporarily store the result of each entry in an anti-diagonal, and the corresponding max index value is calculated in the outer loop, as shown in Fig 1. This approach will be further refined in the final hardware implementation, to meet the space efficiency requirements. (see Hardware Bottleneck Optimizations).
- Optimizing data transfer is inherently complex, and for the Starting Version of the hardware implementation, we will initially focus on optimizing the RAM write-back logic. This logic handles the transfer of the final destination matrix back to RAM, enabling the Arm processor to later validate the results. In this version, we will employ a buffer of size N to temporarily hold the data for each anti-diagonal. This data will then be copied back to RAM using a `memcpy` instruction, providing a functional and simple solution for the initial hardware prototype.

C. Observations and Expectations

At this point in the project we expect Arm runs to be slower than on x86, as the acceleration part will come after using the FPGA fabric to create an accelerator that will implement the logic mentioned in Hardware Optimization Flow.

IV. Execution

A. x86 Execution

1) Comparing the different iterations:

- Iteration I: Implementing these simple optimizations results in up to 1.43x speed up.

Input $M * N$	Original	Iteration I	Speed up
$16 * 32$	$6\mu s$	$5\mu s$	1.2x
$32 * 32$	$10\mu s$	$8\mu s$	1.25x
$32 * 65536$	$15.89ms$	$11.35ms$	1.4x
$256 * 65536$	$134.79ms$	$94.25ms$	1.43x
$256 * 300000$	$614.58ms$	$428.69ms$	1.43x

- Iteration II: This version achieves up to 1.8x speed up. Note that on smaller inputs the speed is not very accurate due to small run times.

Input $M * N$	Original	Iteration II	Speed up
$16 * 32$	$6\mu s$	$3\mu s$	2x
$32 * 32$	$10\mu s$	$6\mu s$	1.6x
$32 * 65536$	$15.89ms$	$9.33ms$	1.7x
$256 * 65536$	$134.79ms$	$75.05ms$	1.79x
$256 * 300000$	$614.58ms$	$341.15ms$	1.8x

It is also clear why this version achieves this speed up when measuring the amount of branch misses each version has. Any iteration after Iteration II which implemented the ternary operator has up to 95x less branch misses.

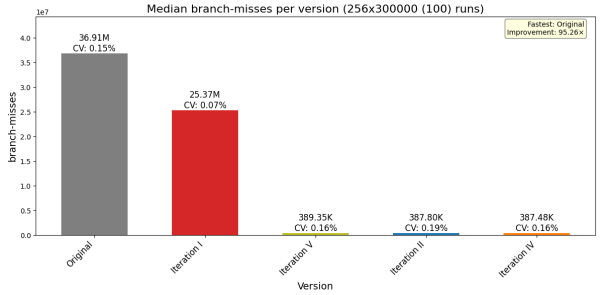


Fig. 4. Graph of branch misses for different iterations

- Iteration III:

Input $M * N$	Original	Iteration III	Speed up
$16 * 32$	$6\mu s$	$2\mu s$	3x
$32 * 32$	$10\mu s$	$4\mu s$	2.5x
$32 * 65536$	$15.89ms$	$6.08ms$	2.61x
$256 * 65536$	$134.79ms$	$46.36ms$	2.9x
$256 * 300000$	$614.58ms$	$209.67ms$	2.93x

- Iteration IV: Taking advantage of the reusable data we can reduce the memory operations and achieve better run times.

Input $M * N$	Original	Iteration IV	Speed up
$16 * 32$	$6\mu s$	$2\mu s$	3x
$32 * 32$	$10\mu s$	$4\mu s$	2.5x
$32 * 65536$	$15.89ms$	$5.26ms$	3.02x
$256 * 65536$	$134.79ms$	$35.86ms$	3.75x
$256 * 300000$	$614.58ms$	$161.3ms$	3.81x

- Iteration V explored the use of prefetching and the restrict keyword to provide optimization hints to the compiler. While theoretically promising, profiling results show that this iteration underperforms compared to Iteration IV.

Input $M * N$	Original	Iteration V	Speed up
$16 * 32$	$6\mu s$	$3\mu s$	2x
$32 * 32$	$10\mu s$	$5\mu s$	2x
$32 * 65536$	$15.89ms$	$5.56ms$	2.85x
$256 * 65536$	$134.79ms$	$37.01ms$	3.64x
$256 * 300000$	$614.58ms$	$171.75ms$	3.57x

- Iteration VI: This iteration as expected will perform worse than the previous for reason explained before. It is though a good starting point for the hardware implementation because it enables parallelization.

2) Conclusions: There were many versions that were implemented but after profiling Iteration IV seems to be the one with the better speed up. Here are the speed up factors for each input tested:

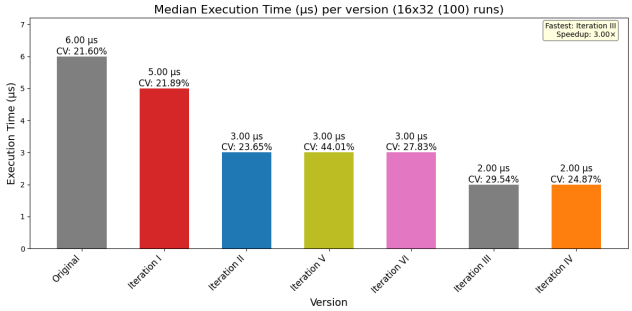


Fig. 5. $N = 16$ $M = 16$

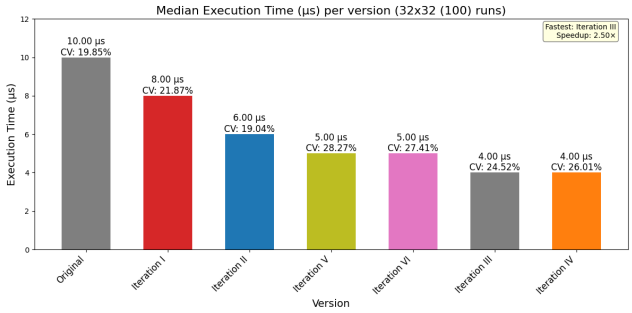


Fig. 6. $N = 32$ $M = 32$

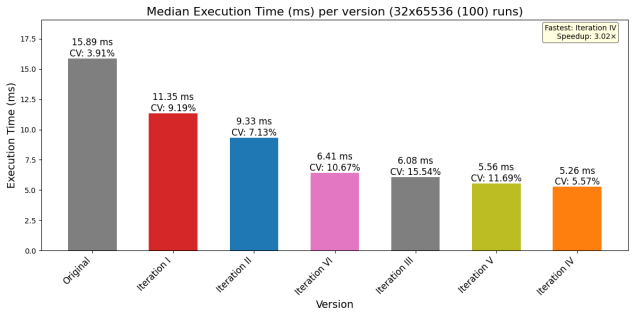


Fig. 7. $N = 32$ $M = 65536$

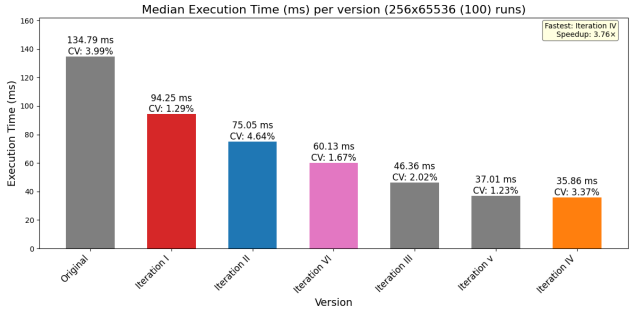


Fig. 8. $N = 256$ $M = 65536$

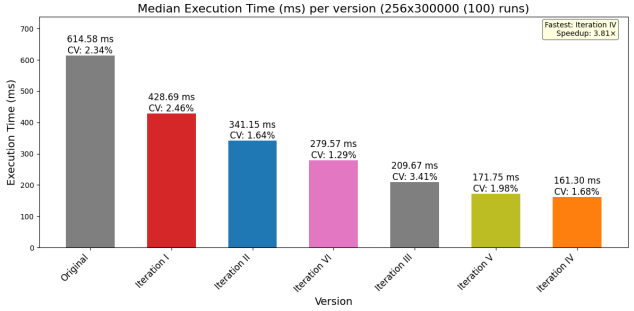


Fig. 9. $N = 256$ $M = 300000$

It is clear that Iteration IV outperforms every other iteration and achieves up to 3.81x speed up on large inputs. We should also note that profiling with smaller inputs becomes harder because of the large covariance of the execution time between executions.

B. Roofline Analysis

To create a roofline model of the different iterations we used Intel Advisor. The goal was to move our implementation as close to the peak performance as possible avoiding hitting the max memory bandwidth. This means we need to lower the memory access and reuse as much of the data as possible.

We can see that as new version are created with data reusability in mind they move towards the peak performance and away from the memory bandwidth limit. Most of the versions are to the left of the ridge point and the best iteration is still Iteration IV which is closer to the peak performance.

C. Arm Execution

On the Arm CPU we used the clock() function to profile the different iterations of software implementations. The results were similar with some exceptions. First, Iteration VI performed worse than the original version. This suggests that the Arm CPU has a smaller cache than the x86 we used to profile our implementations since Iteration VI has really bad locality and reusability of data. Another interesting observation is that on Arm, Iteration I and II didn't differ a lot on execution time. This is probably because the CMOV operation that is available on the x86 instruction set is missing from the Arm instruction since it is RISC.

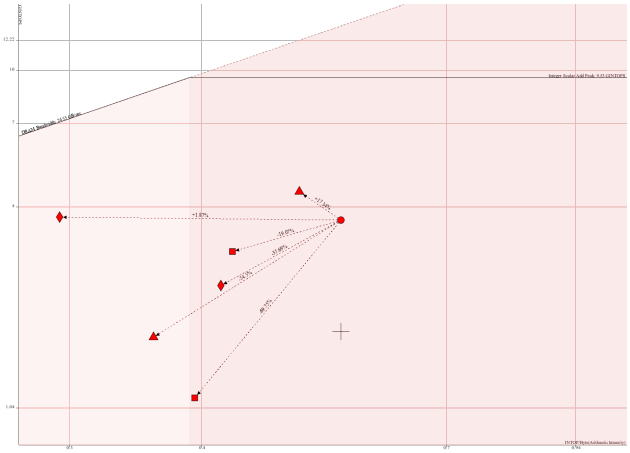


Fig. 10. Roofline model comparing all the different software iterations

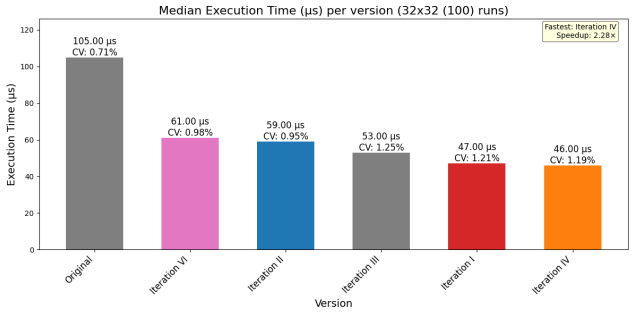


Fig. 11. $N = 32$ $M = 32$

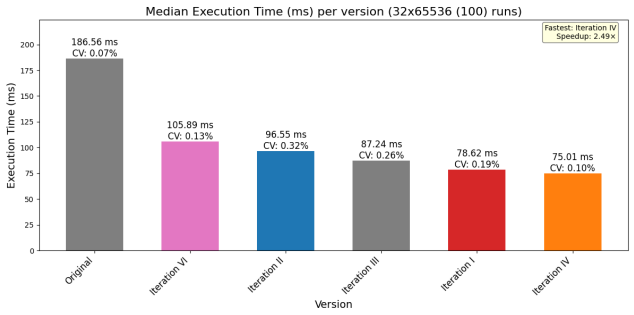


Fig. 12. $N = 32$ $M = 65536$

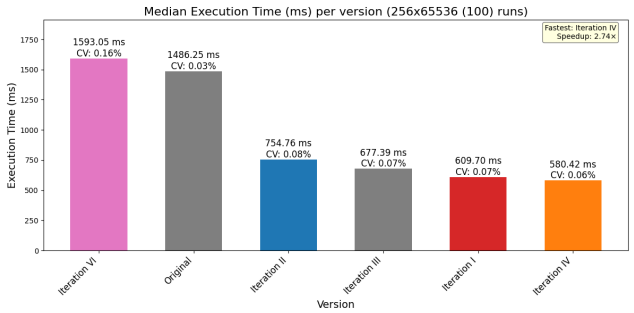


Fig. 13. $N = 256$ $M = 65536$

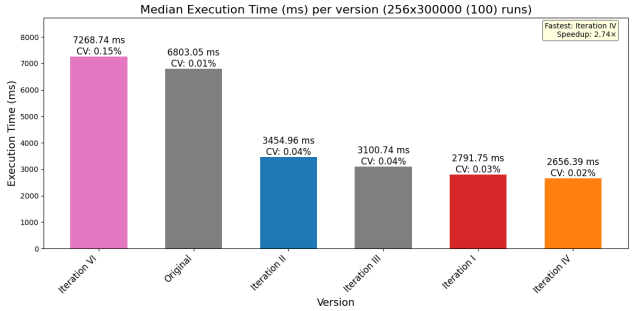


Fig. 14. $N = 256$ $M = 300000$

V. Hardware Implementation

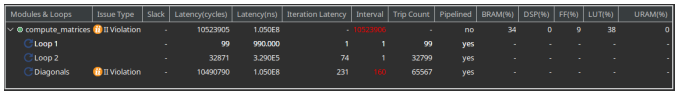
A. Starting Version

a) Version Selection: For the hardware version targeting FPGA implementation, the design must emphasize both parallelization and efficiency. The software structure that best satisfies these requirements is the version presented in Iteration VI, which processes the array along its anti-diagonals, a layout that naturally exposes parallelism. While this may not be a direct one-to-one conversion, the fundamental concept will remain consistent across these versions.

b) Expectations: In theory, this traversal strategy enables fully pipelined execution, allowing a new iteration of the anti-diagonal loop to begin on every clock cycle, provided inter-anti-diagonal dependencies are resolved in time. The pipeline interval should be configured such that sufficient latency is available for the complete evaluation of each anti-diagonal, ensuring correctness while maximizing throughput. As such, the design should avoid long critical paths.

B. Hardware Bottleneck Optimizations

- Implementation I: This implementation serves as the initial prototype of the accelerator. It follows a straightforward approach: input data is first copied into internal buffers, after which the necessary computations are performed to determine the updated values along the current anti-diagonal. Upon completion of each anti-diagonal, the implementation evaluates the maximum value and updates the global maximum if required. The computed results are then written back to memory. The corresponding pseudo code is provided in section Pseudo code.The optimizations applied in this implementation include the use of a single, padded buffer designed to store only the required anti-diagonals of the similarity matrix. This buffer is accessed in a circularly using calculated indices, effectively eliminating the need for `memcpy` operations. Furthermore, the computation of the maximum index has been moved outside the inner loop, thereby reducing the loop's computational depth and enabling faster evaluation of each anti-diagonal, reducing its critical path and loop-carried dependencies—both of which contribute to improved initiation intervals and more efficient pipelining of the anti-diagonal computation.

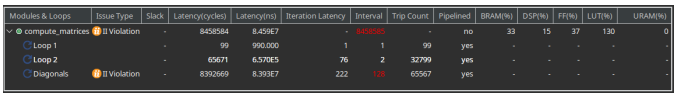


Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
compute_matrices	II Violation	-	10523905	1.050E8	-	10523906	-	no	34	0	9	38	0
Loop 1		-	99	990.000	1	1	99	yes	-	-	-	-	-
Loop 2		-	32871	3.290E5	74	1	32799	yes	-	-	-	-	-
Diagonals	II Violation	-	10490790	1.050E8	231	160	65567	yes	-	-	-	-	-

Fig. 15. Vitis analysis for Implementation I.

- Implementation II: This version builds directly upon the previous implementation, with the primary enhancement being the inclusion of Vitis HLS directives to guide the synthesis tool in generating more efficient hardware. These directives enable tighter control over the pipeline initiation interval (II), facilitating the design of higher-throughput accelerators. The full set of applied directives is provided in Section Pseudo code.Among these, the `ARRAY_PARTITION` directive plays a critical role in enabling parallel access to data structures that would otherwise be bottlenecked by block RAM `BRAM` access limitations. Since `BRAMs` typically support only a limited number of accesses per clock cycle—depending on their configuration—this directive helps mitigate access contention. Specifically, we apply cyclic partitioning with a factor of 16, which distributes sequential elements across multiple memory banks. This configuration strikes a balance between resource utilization and flexibility, supporting a wide range of input sizes (`N` and `M`) without requiring hardware regeneration for each variation. The `ARRAY_RESHAPE` directive is employed to achieve denser data packing, which can lead to improved bandwidth utilization and potentially reduce BRAM usage. By reorganizing the array layout, this directive allows multiple data elements to be accessed within a single memory word, effectively increasing data throughput. However, the effectiveness of this optimization is heavily influenced by the characteristics of the underlying BRAM architecture and the algorithm's data access patterns. If not carefully configured, the use of `ARRAY_RESHAPE` may introduce access conflicts or miss alignments that degrade performance, potentially hindering pipelining and limiting the ability to achieve low initiation intervals (II). Finally, both pipelining and loop unrolling are techniques aimed at increasing concurrency, thereby improving performance at the expense of increased area utilization on the FPGA fabric. Pipelining—illustrated in Figure 21—enables overlapping execution of loop iterations, allowing new iterations to start before previous ones have completed. While powerful, this technique is often difficult to apply effectively due to inter-iteration dependencies, which can introduce pipeline stalls or force the tool to increase the II beyond the selected value. This limitation was directly observed in Implementation I, where an `II=1` constraint was specified but ultimately not met by the tool. The presence of loop-carried dependencies and memory access bottlenecks prevented the

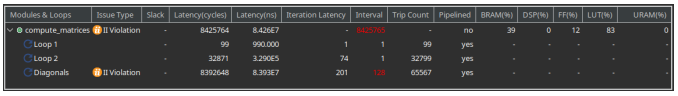
pipeline from achieving the desired throughput, highlighting the need for careful loop restructuring and memory access optimization in subsequent implementations.



Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
compute_matrices	II Violation	-	8458584	8.459E7	-	8458585	-	no	33	15	37	130	0
Loop 1		-	99	990.000	1	1	99	yes	-	-	-	-	-
Loop 2		-	65671	6.570E5	76	2	32799	yes	-	-	-	-	-
Diagonals	II Violation	-	8392669	8.393E7	222	128	65567	yes	-	-	-	-	-

Fig. 16. Vitis analysis for Implementation II.

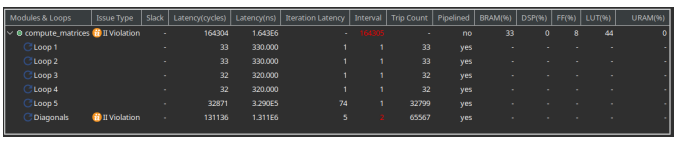
- Implementation III: Building upon the previous design, this implementation addresses a critical resource utilization issue observed in Implementation II—specifically, the excessive use of Look-Up Tables (`LUTs`), which was deemed unacceptable because of resource-constrains. To mitigate this, the `ARRAY_RESHAPE` directive is replaced with the `BIND_STORAGE` directive. This change explicitly instructs the synthesis tool to map arrays to `BRAM` resources instead of `LUTs`, preserving `LUTs` for other logic-intensive operations. By offloading storage to `BRAM`, we free up valuable logic fabric, enabling greater opportunities for parallelization and improved overall hardware efficiency.



Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
compute_matrices	II Violation	-	8425764	8.426E7	-	8425765	-	no	39	0	12	83	0
Loop 1		-	99	990.000	1	1	99	yes	-	-	-	-	-
Loop 2		-	32871	3.290E5	74	1	32799	yes	-	-	-	-	-
Diagonals	II Violation	-	8392648	8.393E7	201	128	65567	yes	-	-	-	-	-

Fig. 17. Vitis analysis for Implementation III.

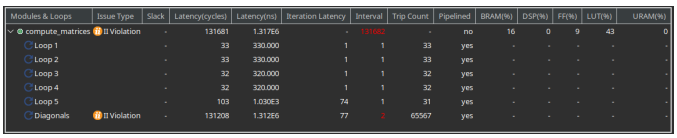
- Implementation IV: In this iteration, more substantial optimizations are introduced to address the still-high resource utilization observed in previous versions across all types. The primary strategy involves minimizing variable bit-widths to precisely match the range of required values, eliminating any unnecessary bytes. This targeted reduction in data sizes is expected to significantly decrease the consumption of `LUTs`, flip-flops (`FFs`), and `BRAM`. Additionally, complete (`ARRAY_PARTITION` with `type=complete`) partitioning is applied to small arrays where full parallel access is critical, enabling maximum concurrency and further aiding pipeline performance.



Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
compute_matrices	II Violation	-	164304	1.643E6	-	164305	-	no	33	0	8	44	0
Loop 1		-	33	330.000	1	1	33	yes	-	-	-	-	-
Loop 2		-	33	330.000	1	1	33	yes	-	-	-	-	-
Loop 3		-	32	320.000	1	1	32	yes	-	-	-	-	-
Loop 4		-	32	320.000	1	1	32	yes	-	-	-	-	-
Loop 5		-	32871	3.290E5	74	1	32799	yes	-	-	-	-	-
Diagonals	II Violation	-	131136	1.311E6	5	2	65567	yes	-	-	-	-	-

Fig. 18. Vitis analysis for Implementation IV.

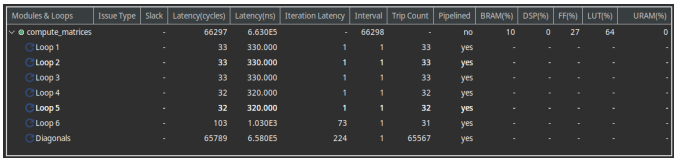
- Implementation V: Additionally to bit-width reductions, this implementation introduces an algorithmic optimization by relocating the maximum index computation outside both the inner and outer loops. This restructuring enables a single linear pass over the relevant elements after the completion of the outer loop, rather than repeatedly computing the maximum during each iteration. This not only reduces redundant computations but also shortens the critical path within the main computation loops, contributing to improved pipelining and overall performance. Finally, we observe that the database buffer—although stored in block RAM—does not require the entire content of `string2` to be loaded simultaneously. Since only `N` elements are needed at any given time for diagonal computation, we implement a shift-based sliding window mechanism: at each iteration, a new byte from `string2` is loaded, and the existing values are shifted left to maintain the active window. This optimization significantly reduces memory bandwidth usage and enables more efficient utilization of BRAMs, leading to both performance gains and lower resource consumption.



Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
compute_matrices	II Violation	-	131681	1.317E6	-	131682	-	no	16	0	9	43	0
Loop 1		-	33	330.000	1	1	33	yes	-	-	-	-	-
Loop 2		-	33	330.000	1	1	33	yes	-	-	-	-	-
Loop 3		-	32	320.000	1	1	32	yes	-	-	-	-	-
Loop 4		-	32	320.000	1	1	32	yes	-	-	-	-	-
Loop 5		-	103	1.030E3	74	1	31	yes	-	-	-	-	-
Diagonals	II Violation	-	131208	1.312E6	77	2	65567	yes	-	-	-	-	-

Fig. 19. Vitis analysis for Implementation V.

- Implementation VI: From Implementation II onward, we were only able to reliably achieve an initiation interval (II) of 2. Attempts to enforce a lower II resulted in negative worst negative slack (WNS), indicating timing violations. At this stage, a key data dependency was identified as the primary bottleneck preventing the achievement of `II=1`. According to the Vitis tool's analysis, there exists a loop-carried dependency between the max-value tracking logic and the `prev_diag` pointer, which references one of the three diagonal buffers. This dependency effectively forces each iteration to wait for the previous one to complete in order to prepare the necessary diagonal data.



Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
compute_matrices		-	66297	6.630E5	-	66298	-	no	10	0	27	64	0
Loop 1		-	33	330.000	1	1	33	yes	-	-	-	-	-
Loop 2		-	**33**	**330.000**	**1**	**1**	**33**	**yes**	-	-	-	-	-
Loop 3		-	33	330.000	1	1	33	yes	-	-	-	-	-
Loop 4		-	32	320.000	1	1	32	yes	-	-	-	-	-
Loop 5		-	**32**	**320.000**	**1**	**1**	**32**	**yes**	-	-	-	-	-
Loop 6		-	103	1.030E3	73	1	31	yes	-	-	-	-	-
Diagonals		-	65789	6.580E5	224	1	65567	yes	-	-	-	-	-

Fig. 20. Implementation VI

- Implementation VII: At long last, this version consolidates all previous optimizations into a single, highly efficient implementation. It builds directly upon Implementation VI, incorporating both proven transformations and additional theoretical improvements aimed at reducing timing slack and minimizing resource usage to the theoretical limits of the algorithm. Notably, the final

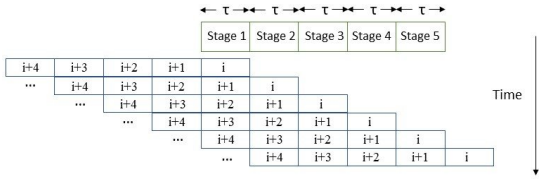


Fig. 21. Pipelining example with II=1

source code includes aggressive bit-width reductions, complete array partitioning, pipelining with `II=1`, and a shift-based sliding window mechanism for the database buffer. These techniques collectively result in minimal BRAM usage, maximized concurrency, and improved timing closure — achieving the most performant and resource-efficient version to date.

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
compute_matrices	-	-	66296	6.630E5	-	66297	-	no	5	0	23	58	0
Loop 1	-	-	102	1.020E3	4	1	99	yes	-	-	-	-	-
Loop 2	-	-	32	320.000	1	1	32	yes	-	-	-	-	-
Loop 3	-	-	32	320.000	1	1	32	yes	-	-	-	-	-
Loop 4	-	-	103	1.030E3	73	1	31	yes	-	-	-	-	-
Diagonals	-	-	65790	6.580E5	225	1	65567	yes	-	-	-	-	-

Fig. 23. Implementation VII

VI. Debugging hardware implementation

A. Hardware Execution

Unfortunately, we were not able to thoroughly test all software versions on the actual hardware. All presented implementations successfully passed every stage up to and including hardware emulation, but failed during deployment on the FPGA. Despite extensive testing, we were unable to pinpoint the exact issue, though we confidently determined that the problem lies in the kernel code itself, not in the emulation environment or the bitstream synthesis tools — as detailed in subsection Debugging strategy.

One modified variant of Implementation IV did run successfully on the board with an input size of 32×65536, but due to time constraints, we were unable to benchmark it further or determine its limits. The other implementations failed to execute on the FPGA, so we cannot draw any conclusions about their actual hardware runtime.

The version that did run was estimated by Vitis HLS to execute in $1.6ms$, but the observed runtime on the board was approximately $8.8ms$.

B. Debugging strategy

In order to find a bug that was hiding between the software emulation and hardware we needed to eliminate any possible causes for it. Since the HLS development cycle includes a lot of steps there could be many causes that affect the final outcome. We needed to check each step thoroughly to pinpoint the exact cause.

a) Synthesizing the hardware: We first made sure that the synthesizing tool was working correctly and didn't actually produce any logs or messages we missed. After extensively checking the logs and the final report produced we came to the conclusion that there wasn't a problem with the tool.

b) Host program on Arm: This is another part of the implementation we suspected could be wrong. Since the host program is compiled for Arm but is emulated on x86 we thought there could be a case were something behaves differently between the two. After creating a new version of the host program starting from scratch the problem still persisted so we moved on to the last part.

c) Kernel code: This is surely the hardest part you could locate the bug since we had very little information about what RTL is actually created from a specific version of the kernel code. Due to limited time we were unable to locate the problem in any of our implementations.

VII. Appendix I

The CMOV (Conditional Move) instruction is a CPU instruction that conditionally moves a value into a destination register based on some condition. Basically, if the condition is true, the value of a source operand is moved to the destination register. Otherwise, the destination register remains unchanged. Both generally and in our case, CMOV is used for branch avoidance, especially in cases where the branch condition is highly unpredictable, effectively avoiding the performance overhead of taking a branch, or even worse miss-predicting a branch.

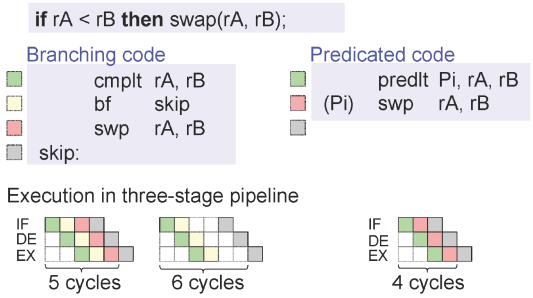


Fig. 24. CMOV vs Branch pipelining inside the CPU

Below is a basic example of how a CMOV instruction is fundamentally different from a simple branch instruction.

```
if (ecx == 5) {
    eax = TRUE
} else {
    eax = FALSE
}
```

Assembly code using basic branches:

```
cmp ecx,5          ; is ecx==5
jne unequal        ; what is the address of
                     the next instruction?
                     conditional branch
mov eax,TRUE       ; possibility one
jmp fin
unequal:           ; possibility two
mov eax,FALSE
fin:
nop
```

Assembly code using CMOV:

```
cmp ecx,5
mov eax, FALSE     ; mov doesn't affect
                     flags
mov ebx, TRUE      ; because CMOV doesn't
                     take immediate src
                     operands
cmove eax, ebx     ; executes as MOV if
                     zero-flag is set,
                     otherwise as NOP
nop                ; no stall
```

Specifically for the gcc compiler in order to hint at the usage of a CMOV instruction, the following syntax is advised:

```
cond_var = <condition>;
var = <condition> ? <val> : <val>;
```

Syntax to be avoided:

```
if (<condition>) {
    var = <val>;
} else {
    var = <val>;
}
```

VIII. Pseudo code

Algorithm 1 Simple Antidiagonal Traversal

Input: Strings S_1, S_2
Output: Maximum index, Direction matrix
Copy S_1 and S_2 into local buffers
for each diagonal index d from 0 to total_diagonals **do**
  for each row r in current diagonal **do**
    Compute character match/mismatch score
    Retrieve values from diagonals:
    - North
    - West
    - Northwest
    Compute maximum among these and zero
    Store result in current diagonal
    Record direction of maximum
  end for
  Find maximum value in current diagonal
  if a new maximum is found **then**
    Update global max index
  end if
  Rotate diagonal buffer pointers
  Write back computed directions in direction matrix
end for

Algorithm 2 LSAL Intricate Matrix Computation

Input: Query string $S_1[0 \dots N-1]$, Database string $S_2[0 \dots M-1]$
Output: max_index, Direction matrix
Initialize 3 circular diagonal buffers to 0
Copy query and initial database window into local buffers
for each diagonal index d from 0 to $M + N - 2$ **do**
  Determine current, previous, and pre-previous diagonal indices
  for each row r in current diagonal **do**
    Compare $S_1[r]$ with current character in database window
    Compute score from:
    - Northwest cell (diagonal $d - 2$)
    - North cell (diagonal $d - 1$)
    - West cell (diagonal $d - 1$)
    Select maximum among those scores and zero
    Store direction of the selected maximum
    Write result to current diagonal buffer
    if new score $>$ previously stored row max **then**
      Update per-row maximum score and its diagonal index
    end if
  end for
  Shift the database window by 1 character
  Copy direction buffer to the output direction matrix
end for
Initialize global maximum value and corresponding coordinates
for each row r **do**
  if row max score $>$ global max **then**
    Update global max value and its coordinates
  end if
end for
Compute final max index using diagonal and row position
Return max_index