ECE340 - Embedded Systems

Lab Report

Liaskonis Spiros, Gantsios Georgios

Accelerating Local Sequence Alignment Algorithm (LSAL) using FPGAs

03381, 03283

June 28, 2024

`sliaskonis@uth.gr, ggantsios@uth.gr`

Contents

Abstract

The Local Sequence Alignment Algorithm (LSAL) is a crucial tool in bioinformatics for identifying regions of similarity within DNA and RNA sequences. Field Programmable Gate Arrays (FPGAs) offer a promising solution in accelarating LSAL due to their parallel processing capabilities and reconfigurable architecture. Purpose of this project is to explore the implementation and acceleration of LSAL on the ZedBoard FPGA platform. We present the design and optimization strategies employed to enhance the performance of LSAL, addressing the challenges of large query and database sizes. Our results demonstrate significant improvements in computational speed and efficiency.

1 Introduction

Bioinformatics integrates biology, computer science, and information technology to analyze and interpret vast amounts of biological data. Among the computational techniques used, sequence alignment is essential for comparing DNA, RNA, or protein sequences. Sequence alignment algorithms are divided into global and local alignment categories. Global alignment algorithms attempt to align entire sequences, while local sequence alignment algorithms (LSAL) focus on identifying the most similar subsequences within larger sequences.

The Local Sequence Alignment Algorithm (LSAL) is widely used to spot conserved patterns and functional domains within DNA and RNA sequences, aiding in the understanding of evolutionary relationships, gene function, and regulatory mechanisms. However, with the rapid growth of biological databases, traditional implementations of LSAL struggle to handle large-scale queries efficiently.

To address this challenge, we explore the use of Field Programmable Gate Arrays (FPGAs) to accelerate LSAL. FPGAs are known for their parallel processing capabilities and flexibility, making them suitable for computationally intensive tasks. In this project, we focus on implementing LSAL on the ZedBoard FPGA platform, a widely-used development board featuring a Xilinx Zynq-7000 SoC. Our goal is to optimize the performance of LSAL for large query and database sizes, leveraging the hardware capabilities of the ZedBoard FPGA to achieve significant speedup and efficiency improvements.

2 LSAL Implementation for x86 and ARM architecture

2.1 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.

2.1.1 Similarity Matrix

Let $A = a_1a_2 \dots a_m$ and $B = b_1b_2 \dots b_n$ be the sequences to be aligned, where m and n are the lengths of A and B respectively.

1. Determine the substitution matrix and the gap penalty scheme.
   - $s(a, b)$ - Similarity score of the elements that constituted the two sequences
   - W_k - The penalty of a gap that has length k
2. Construct a similarity matrix S (Figure 1 (a)) and initialize its first row and first column. The size of the scoring matrix is $(n + 1) \times (m + 1)$. The matrix uses 0-based indexing.

$$S_{k0} = S_{0l} = 0 \quad \text{for} \quad 0 \le k \le m \quad \text{and} \quad 0 \le l \le n$$

3. Fill the similarity matrix using the equation below.

$$S_{ij} = \max \begin{cases} S_{i-1,j-1} + s(a_i, b_j), \\ \max_{k\ge1}\{S_{i-k,j} - W_k\}, \\ \max_{l\ge1}\{S_{i,j-l} - W_l\}, \\ 0 \end{cases} \quad \text{for} \quad (1 \le i \le m, 1 \le j \le n)$$

where

$S_{i-1,j-1} + s(a_i, b_j)$ is the score of aligning a_i and b_j,
$S_{i-k,j} + W_k$ is the score if a_i is at the end of a gap of length k,
$S_{i,j-l} + W_l$ is the score if b_j is at the end of a gap of length l,
0 means there is no similarity up to a_i and b_j.

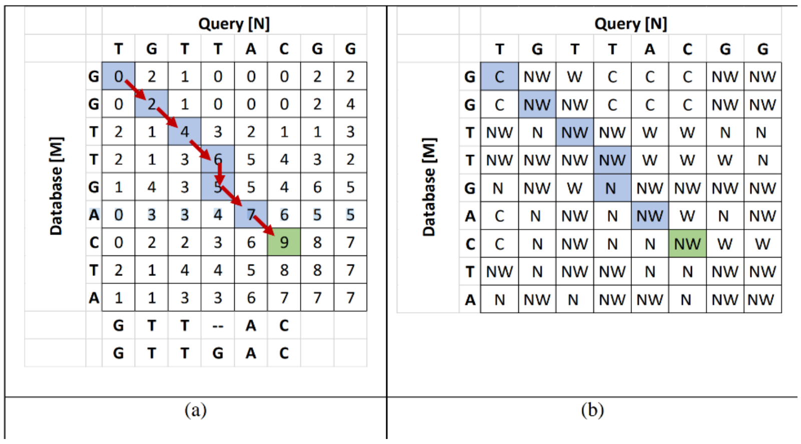
In our implementation k = 1 and W_k = -1. So the equation becomes simpler.

$$S_{ij} = \max \begin{cases} S_{i-1,j-1} + s(a_i, b_j), \\ \max_{\{} S_{i-1,j} - 1\}, \\ \max_{\{} S_{i,j-1} - 1\}, \\ 0 \end{cases} \quad \text{for} \quad (1 \le i \le m, 1 \le j \le n)$$

In our case, we only need to read 3 elements (one on the North, one one the West and one on the Northwest) for the calculation of the current similarity element.

2.1.2 Direction Matrix

A second matrix (called direction matrix) is formed to hold the direction path through matrix S to produce an alignment (Figure 1 (b)). The cell value of the direction matrix indicates the direction followed to reach that cell. The optimal alignment is produced by starting from the position of the maximum score in matrix S and following the directions found at the same index in the direction matrix until a zero in S is found.



(a)

Query [N]

Database [M]

	T	G	T	T	A	C	G	G
G	0	2	1	0	0	0	2	2
G	0	2	1	0	0	0	2	4
T	2	1	4	3	2	1	1	3
T	2	1	3	6	5	4	3	2
G	1	4	3	5	5	4	6	5
A	0	3	3	4	7	6	5	5
C	0	2	2	3	6	9	8	7
T	2	1	4	4	5	8	8	7
A	1	1	3	3	6	7	7	7
	G	T	T	--	A	C		
	G	T	T	G	A	C		

(b)

Query [N]

Database [M]

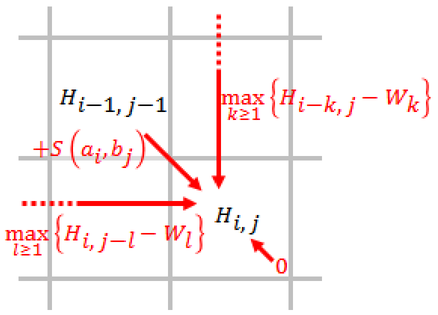
	T	G	T	T	A	C	G	G
G	C	NW	W	C	C	C	NW	NW
G	C	NW	NW	C	C	C	NW	NW
T	NW	N	NW	NW	W	W	N	N
T	NW	NW	NW	NW	W	W	W	N
G	N	NW	W	N	NW	NW	NW	NW
A	C	N	NW	N	NW	W	N	NW
C	C	N	NW	N	N	NW	W	W
T	NW	N	NW	NW	N	N	NW	NW
A	N	NW	N	NW	NW	N	NW	NW

Figure 1: LSAL example of: (a) Similarity Matrix, (b) Direction Matrix

2.2 Optimizations

2.2.1 Data Depedency

Because of data dependencies (Figure 2) Smith-Waterman algorithm can only calculate similarity matrix S cell by cell.



$H_{i-1,j-1}$
$\max_{k\geq 1}\{H_{i-k,j}-W_k\}$
$+s(a_i,b_j)$
$H_{i,j}$
$\max_{l\geq 1}\{H_{i,j-l}-W_l\}$
0

Figure 2: Data Depencdency of each Similarity cell

We can see that each cell depends on 3 neighbouring cells. One to the North, West and Nortwest which we will be referring to as N, W and NW respectively.

2.2.2 Fine-grain parallelism

Typical dynamic programming-based algorithms, like Smith-Waterman algorithm, compute an $S_{m,n}$ matrix (m and n being the sequence lengths) depending on the three entries $S_{i-1,j}$, $S_{i,j-1}$, and $S_{i-1,j-1}$. Fine-grain means that processors will work together in computing the S matrix, cell by cell. Some researchers organized the parallel machine as an array of processors to compute in diagonal-sweep fashion the matrix S (see Figure 1.3). Query sequence length determines the maximum number of processors able to be assigned, and processors remain idle at begin/end steps.

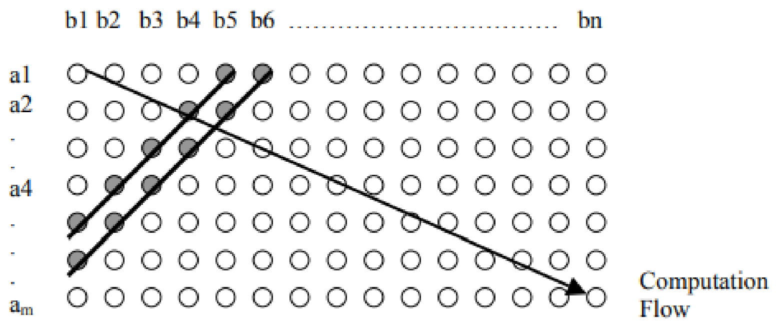


Figure 3: Diagonal Sweep fine-grained workload distribution for multiprocessors machines to avoid data dependencies in Smith-Waterman algorithm

Considering this, instead of calculating S matrix row-wise we can calcuate it iterating over S's anti-diagonals.

2.3 Algorithm Execution - Results

Our project requires implementing the LSAL algorithm and comparing running times amongst implementation variances and machines. For example, running Smith-Waterman algorithm on zedboard's ARM CPU has totally different results from running Fine-grain parallelism optimazation on FPGA. Below are described our implementations and the results of each implementation on Zedboard's x86 and ARM

2.3.1 Our Implementations

We focused on 3 variations/implementations of the algorithm whcih were implemented in 3 different files

- **lsal.cpp:** This file contains a simple implementation of Smith-Waterman algortihm. Note that the first loop is seperate in order to avoid checking for the special case if element is in the first row, N=0 and NW=0. Also, in the second loop we must check if element is in the first column, W=0 and NW=0.

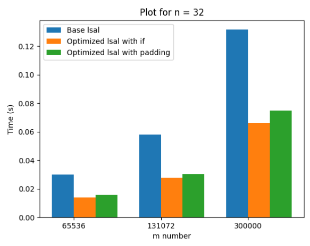
- **lsal_opt.cpp:** In this implementation we calculated seperately the first element of the first row to avoid checking that in the first loop and we added padding in the second loop before the first column to avoid the second check.
- **lsal_opt_pad.cpp:** In this implementation we added a row and a column of padding in order to avoid checking for the 2 special cases and made the code a lot simpler than the second code.

The execution times include only the execution time of the compute_matrix function in which the LSAL algorithm is implemented. That is, the extra time for padding in the matrix is not included.

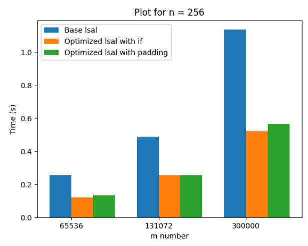
In the following diagrams the results of the implementations are represented in blue ●, orange ●, and green ●respectively for different query (N) and database (M) sizes.

2.3.2 x86

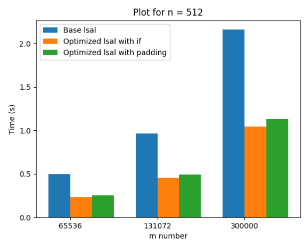
Firstly, we ran our 3 variations on our x86 CPU and the results we got are the following.



(a) Query length (N) = 32



(b) Query length (N) = 256



(c) Query length (N) = 512

Figure 4: LSAL runtimes for different query sizes in x86 CPU of our 3 variations.

We can see that the 2 implementation is the fastest one. The implementation with padding is barely slower and the first implementation is clearly the slowest one. Even though the second implementation seems a good choice for a CPU it is far from the optimal solution for the FPGA due to frequent if statements and minimum parallelism.

2.3.3 Roofline Analysis

We tested the different designs of the lsal algorithm (the original algorithm, the lsal_opt and the lsal_opt_pad) using the Intel Advisor and got the following results:

We see (Figure 5) that because all solutions are to the right of ridege point, optimal solutions are consider the ones that are higher. Those are the 2nd and the 3rd

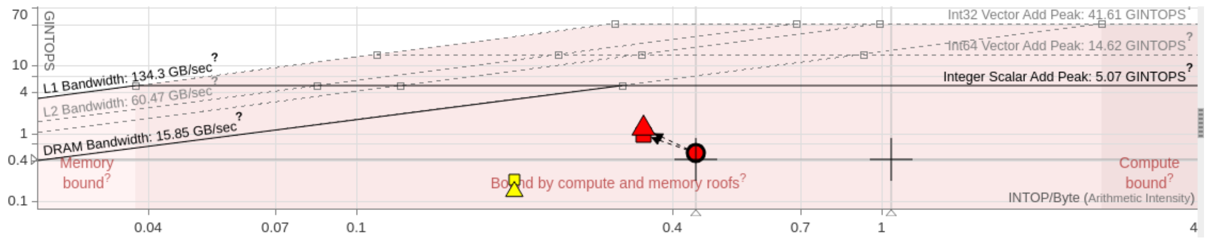
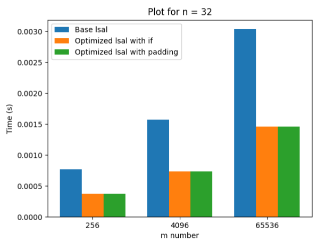


Figure 5: Roofline model for (N,M)=(64,307200)

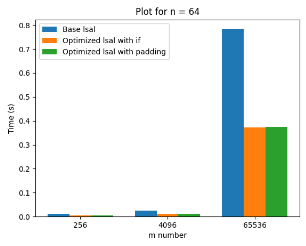
implementation represented by the triangle and the square. The 1st implementation has poorer performance because it is in a lower point even though it is considered computed bound at a higher level.

2.3.4 ARM

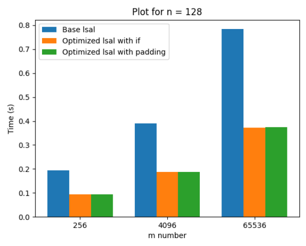
We ran the same code variations on Zedboard's ARM processor which is the Dual-core ARM Cortex™-A9 processor and got the following results.



(a) Query length (N) = 32



(b) Query length (N) = 64



(c) Query length (N) = 128

Figure 6: LSAL runtimes for different query sizes in Zedboard's ARM processor of our 3 variations.

Here we, also, notice that there is not much difference in the 2nd and the 3rd implementations' execution time.
We can conclude that because Dual-core ARM Cortex™-A9 processor is older and thus slower than our x86 CPU it requires more time for the same database and query sizes. Also, for query length 64 we observe the biggest time increase for increase of database length.
We avoided enormous query and database lengths because of ARM's limited capabilities.

3 LSAL Implementation on FPGA

3.1 Unoptimized code on FPGA-Vitis HLS

First, we execute the initial implementation of the LSAL algorithm (with no optimizations) on Vitis HLS, in order to get some initial estimations about the execution time and FPGA resources.

For the sizes of $N = 32, M = 65536$ we get the following estimations/results:

3.2 Optimizations

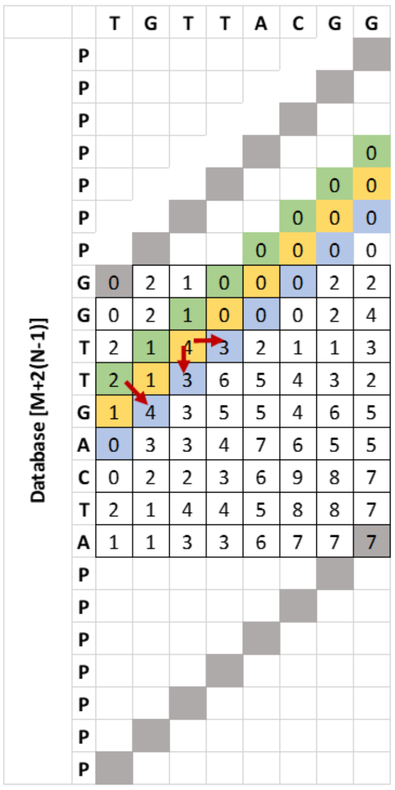
The initial implementation can be improved if we apply pipeline and loop unrolling in both loops. The results after these two optimizations are the following:

However, the current structure of the code doesn't give us enough room for further optimizations. This is due, as we argued earlier to data dependencies.

3.2.1 Algorithm optimizations

In order to unlock parallelism we can compute the similarity matrix elements as described in Fine-grain parallelism section. This allows us to calculate N elements (all the elements of an antidiagonal) in parallel by using more FPGA resources. Although, this creates an unstable number of elements calculated per iteration (as N-1 initial and N-1 last iterations are differing). To fix the issue we padded the table as shown below. Obviously, elements of padding are considered 0 and database characters that correspond to the padding areas are set to P to differ from all A, T, C, G and thus avoiding effecting the outcome of the algorithm.

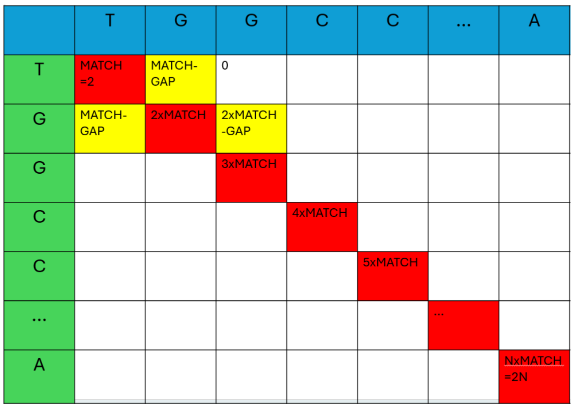
Also, we can avoid storing the similarity matrix almost entirely if we consider that for each element we use N and W (previous anti-diagonal) and NW (pre-previous anti-diagonal). Therefore, to calculate an anti-diagonal we only need 3 anti-diagonals. The current one, the previous one and the one before the previous one(Figure 6).



	T	G	T	T	A	C	G	G
P								
P								
P								
P								0
P							0	0
P						0	0	0
P					0	0	0	0
G	0	2	1	0	0	0	2	2
G	0	2	1	0	0	0	2	4
T	2	1	4	3	2	1	1	3
T	2	1	3	6	5	4	3	2
G	1	4	3	5	5	4	6	5
A	0	3	3	4	7	6	5	5
C	0	2	2	3	6	9	8	7
T	2	1	4	4	5	8	8	7
A	1	1	3	3	6	7	7	7
P								
P								
P								
P								
P								
P								
P								

Figure 7: Similarity Computation in FPGA

Another observation we made relates to the maximum value that can be contained in a similarity matrix of size MxN. In fact to achieve the maximum value we have to follow the diagonal direction permanently since deviation from the diagonal either horizontally or vertically stacks some GAP. Therefore to achieve the maximum value depends on whether the Query is contained exactly the same within the Database. If so then the maximum value is MATCH x N (Figure 7). In our case 2N. Therefore if we declare (in C language) the elements and arrays that store similarity values as a char [-128, 127] value range type instead of int we can make implementations with query length up to floor(127/2)=63.



	T	G	G	C	C	...	A
T	MATCH =2	MATCH-GAP	0				
G	MATCH-GAP	2xMATCH	2xMATCH -GAP				
G			3xMATCH				
C				4xMATCH			
C					5xMATCH		
...						...	
A							NxMATCH =2N

Figure 8: Moving in the diagonal creates the max similarity value if the Query is contained exactly the same within the Database.

Generally, for implementations of query length N, max value of our variables must be at least 2N but we want to use the data type with the less amount of bits possible in order to avoid allocating extra resources (LUTs, FFs and BRAMs).
To determine the number of bits required to store the maximum value $2 \times N$ including an extra bit for the sign, follow these steps:

1. Identify the Range:

The value $2 \times N$ is the maximum positive value. Therefore, we need to represent values from $-2 \times N$ to $2 \times N$.

2. Bit Representation:

To store the maximum value $2 \times N$, we need to find the smallest integer k such that:

$$2^k - 1 \geq 2 \times N$$

$$2^k \geq 2 \times N + 1$$

Taking the base-2 logarithm on both sides:

$$k \geq \log_2(2 \times N + 1)$$

Thus, the smallest integer k is:

$$k = \lceil \log_2(2 \times N + 1) \rceil$$

3. Include the Sign Bit:

Since we also need an extra bit to indicate the sign, the total number of bits required is:

$$k_{\text{total}} = \lceil \log_2(2 \times N + 1) \rceil + 1$$

Therefore, the number of bits required to represent values from $-2 \times N$ to $2 \times N$ including the sign bit is:

$$k_{\text{total}} = \lceil \log_2(2 \times N + 1) \rceil + 1$$

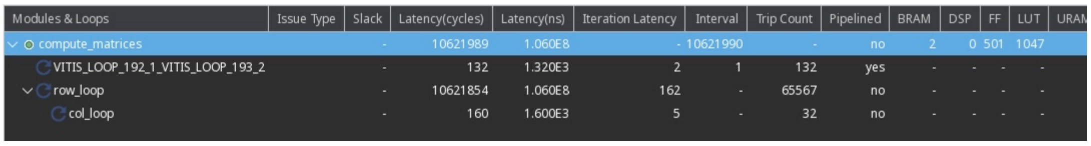
So for our experiments, the following table demonstrates the data type used for similarity score variables in our code depending on query length (N).

Table 1

N	Minimum Data Type Length	Data Type
32	8	char
64	9	short
128	10	short

3.2.2 Pragmas

After implementing the algorithm using only the three antiagonals as mentioned before (and removing the similarity matrix entirely) we test test the algorithm on vitis and we get the following results:

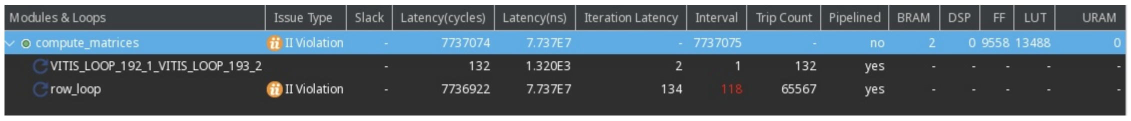


Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
compute_matrices		-	10621989	1.060E8	-	10621990	-	no	2	0	501	1047	
VITIS_LOOP_192_1_VITIS_LOOP_193_2		-	132	1.320E3	2	1	132	yes	-	-	-	-	
row_loop		-	10621854	1.060E8	162	-	65567	no	-	-	-	-	
col_loop		-	160	1.600E3	5	-	32	no	-	-	-	-	

Figure 9: Vitis results after algorithmic optimizations

At this point no optimizations (except the algorithmic ones mentioned above) are applied. We observe that the latency of the current design is at 106ms whereas the utilization of all resources is nearly at 0% (since no resources are used for an optimization).

From here, the first optimization that can be applied is loop unrolling and pipelining. The outer loop (row_loop) is pipelined and the inner loop is fully unrolled in order to unlock more parallelism. The iteration interval for pipelining is set to default in order to let vitis choose the most appropriate one that the design can currently achieve. The results after pipelining and unrolling are the following:

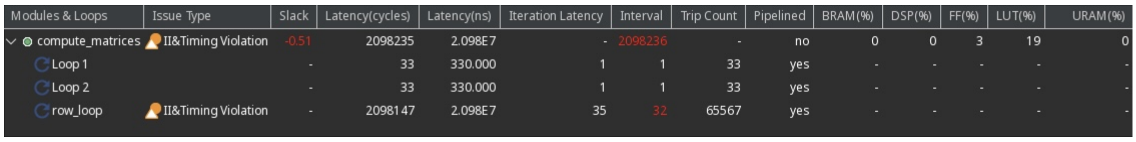


Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
compute_matrices	II Violation	-	7737074	7.737E7	-	7737075	-	no	2	0	9558	13488	0
VITIS_LOOP_192_1_VITIS_LOOP_193_2		-	132	1.320E3	2	1	132	yes	-	-	-	-	-
row_loop	II Violation	-	7736922	7.737E7	134	118	65567	yes	-	-	-	-	-

Figure 10: Vitis results - Pipelining and unrolling

The iteration interval, as expected, has been reduced to 134 (compared to 162) which results in a new latency of 77.3 ms. However, from figure 9 can be observed that these optimizations also resulted in bigger area with FFs increased 19% and LUTs 13% compared to the previous design (which is expected since pipelining the loop leads to duplication of resources such as registers, logic elements and memory, which are used to handle overlapping operations).

At this point, most of design's bottlenecks are caused when reading data from the memory. Here, most of the data used in the design reside in the main memory, except the similarity buffer (a small buffer of size 3*(N+1), where +1 is added as padding in the first column in order to avoid if statements in cases where we don't have to compute values such as west and northwest). This buffer is stored in BRAMs, which in this case since its only 2 BRAMs, cause memory bottlenecks due to limited ports. Thus, in order to achieve a higher bandwith, array partitioning can be used. Complete array partitioning for this buffer is the prefered choice since its reading/writing memory pattern is quite complex and other array partitions such as block and cyclic won't fix the current problem. However, complete array partition for a buffer of this size, results in long synthesis times in Vitis as greater values of N is used. In our case, where N = 32, Vitis takes a very long time to synthesize the design. After experimenting, we decided to seperate the similarity buffer in three different buffers, one for each row of the buffer. The complete array partitioning for these smaller individual buffers was used and the results were the following:



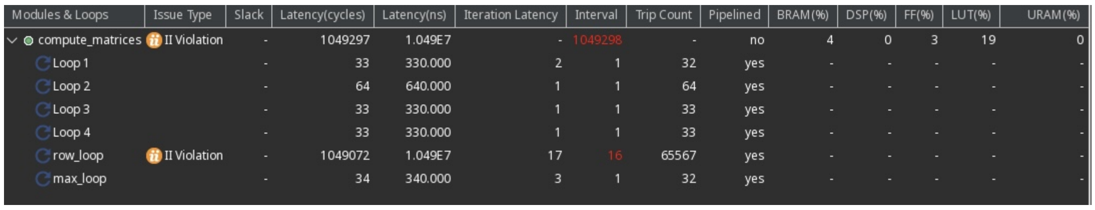
Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
compute_matrices	II&Timing Violation	-0.51	2098235	2.098E7	-	2098236	-	no	0	0	3	19	0
Loop 1		-	33	330.000	1	1	33	yes	-	-	-	-	-
Loop 2		-	33	330.000	1	1	33	yes	-	-	-	-	-
row_loop	II&Timing Violation	-	2098147	2.098E7	35	32	65567	yes	-	-	-	-	-

Figure 11: Array partitioning on similarity buffer(s)

The iteration interval was reduced even more at the value of 32 which resulted in a latency of 20.9 ms. As of the utilization resources, no BRAMs are used since the only data that was stored in BRAMs is now residing in individual registers.

The similarity buffer is not the only data that needs to be read in each iteration of the inner loop. The query as well as the database buffer is read from the main memory in each iteration of the inner loop. In order to fix that, we copy the query and database into buffers, which we used in the design. However, the database (of size M+2*(N-1)) is quite large and would result in huge resource utilization. We observe from the algorithm, that in each iteration of the outer loop, we don't need the whole database string. We only need N values of the database string which are compared with the N query values. Thus, in each outer loop, we copy the needed database values from the main memory into the database buffer using memcpy. One last observation made here is that the database buffer doesn't need to be copied entirely in each iteration of the outer loop. Only one new element is added in the database buffer. Thus, we completely remove the memcpy function and instead we shift all the database buffer elements one place to the left and read only one element (the new database element) from the memory into the last database buffer position. This results in smaller area and thus smaller resource utilization which in our case is crucial for when larger values for N are used.

Even with the optimizations we mentioned in the previous section, there are still data dependencies in the code. Specifically, we observed is a read after write dependency of variables that hold max value and max index. To parallel the inner loop iterations of each anti-diagonal completely, we must use a different max value variable for each column of the matrix. Therefore, we defined a max_value_array[N] and respectively a max_index_array[N] where we hold the maximum values and indices of each column respectively and at the end of the kernel function we find the maximum value in the array in order to arrive at the final max index. We ended up storing the 2 tables in BRAM in a cyclic way and factor 4.



Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
compute_matrices	II Violation	-	1049297	1.049E7	-	1049298	-	no	4	0	3	19	0
Loop 1		-	33	330.000	2	1	32	yes	-	-	-	-	-
Loop 2		-	64	640.000	1	1	64	yes	-	-	-	-	-
Loop 3		-	33	330.000	1	1	33	yes	-	-	-	-	-
Loop 4		-	33	330.000	1	1	33	yes	-	-	-	-	-
row_loop	II Violation	-	1049072	1.049E7	17	16	65567	yes	-	-	-	-	-
max_loop		-	34	340.000	3	1	32	yes	-	-	-	-	-

Figure 12: Replacing the find max value algorithm

After changing the way that max value is calculated iteration interval is cut in half which results as expected in a smaller latency of just 10.4 ms. As of the utilization resources we have an increased number of BRAMs which is now at 4% as well as FFs and LUTs at 3% and 19% respectively.

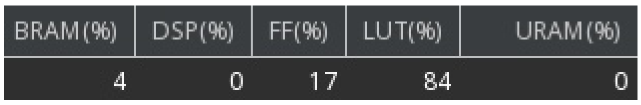
As mentioned above, all of these tests were done for the values of N = 32 and M = 65536. We observe from table 1 that for the value of N = 32, we don't need an array of type short in order to calculate the values of the similarity buffer. Thus for values of up to 63, we change the array type from short to char.

3.3 Execution of LSAL on FPGA

Quick Note: Note that the Resource Utilization depends only on query size N.

Despite that both N and M affect execution time.
N by increasing Iteration Interval (unless increasing partition factor but this increases Resource Utilization even more).
M increases time by increasing the number of outer loop iterations.

At this point we can start executing the code on the zedboard. After applying all the optimization, we ended up executing the algorithms for the values of N = [32, 64, 128] and M = [65536, 307200]. For the bigger value of N that we executed the algorithm we got the following resource utilization:



BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
4	0	17	84	0

Figure 13: Resource utilization for N=128, M=307200

The biggest constrain here is the number of LUTs which in the case of N=128 stands at 84% of the total LUTs of the device.

Important: When synthesizing the design for execution in Zedboard, the II differs for different values of N.

The number of initiation intervals (II) in the kernel code significantly impacts successful synthesis for execution on the Zedboard. Here's a breakdown of this critical factor for different values of N:

- **N = 32:** Synthesis reports indicate that for N equal to 32, the actual required II is *8*. Setting a lower II value during synthesis will lead to errors.
- **N = 64:** The default II value of 16 (automatically set by Synthesis) will cause synthesis failure. To ensure successful synthesis, one **must explicitly define the II in your kernel code and set it to *18***.
- **N = 128:** Synthesis again sets the default II to 32. However, for N equal to 128, this value is wrong and leads to timing violations during synthesis. To address this, one **must manually set the II to *34*** in your kernel code.

Table 2 and Figure 8 below show the runtime results of our experiments for various values of N and M.

Experiment	N	M	Time (ms)
1	32	65536	6.233205
2	64	65536	12.885118
3	32	307200	25.671153
4	64	307200	56.652430
5	128	65536	23.384481
6	128	307200	105.660737

Table 2: Runtime results for different N and M

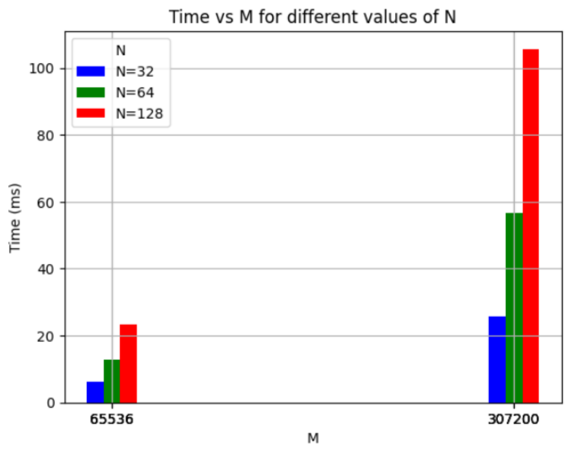
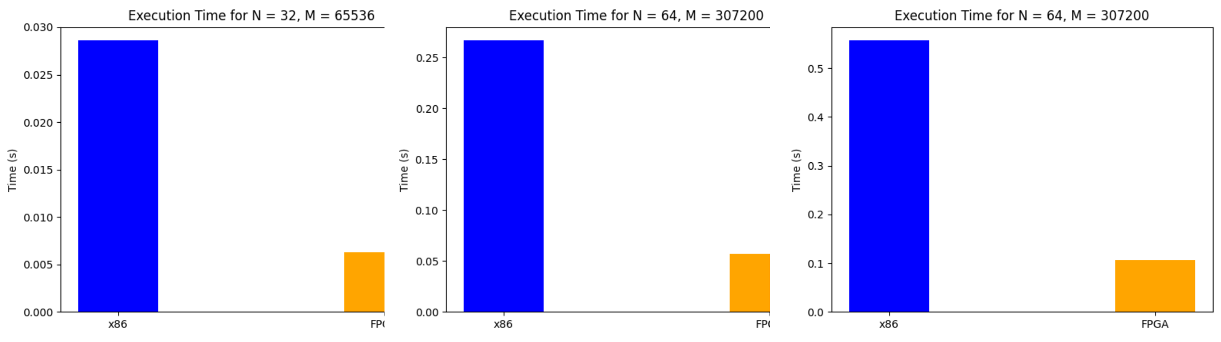


Figure 14: Runtimes in FPGA for different N and M

Comparing these runtimes with the ones of x86 and ARM we see that FPGA performance is indeed superior, especially from ARM. In terms of performance we get approximately 5x better performance when executing the same algorithm on FPGA compared to an x86 machine. This performance hit is even larger when comparing the FPGA executions with the ones on the ARM-Cortex. In the graphs below we can see some of the comparisons between FPGA and x86 (intel) for various values of N and M.



(a) N,M=(32,65K) (b) N,M=(64,300K) (c) N,M=(128,300K)

Figure 15: FPGA-x86(intel) comparison for different N,M values