University of Thessaly

Department of Electrical and Computer Engineering

E_CE_U_137: Embedded Systems

LAB 3

FPGAs AS ACCELERATORS

Implementation :

Tsatrafili Eirini Eleftheria

Tselepi Eleni

Professor :

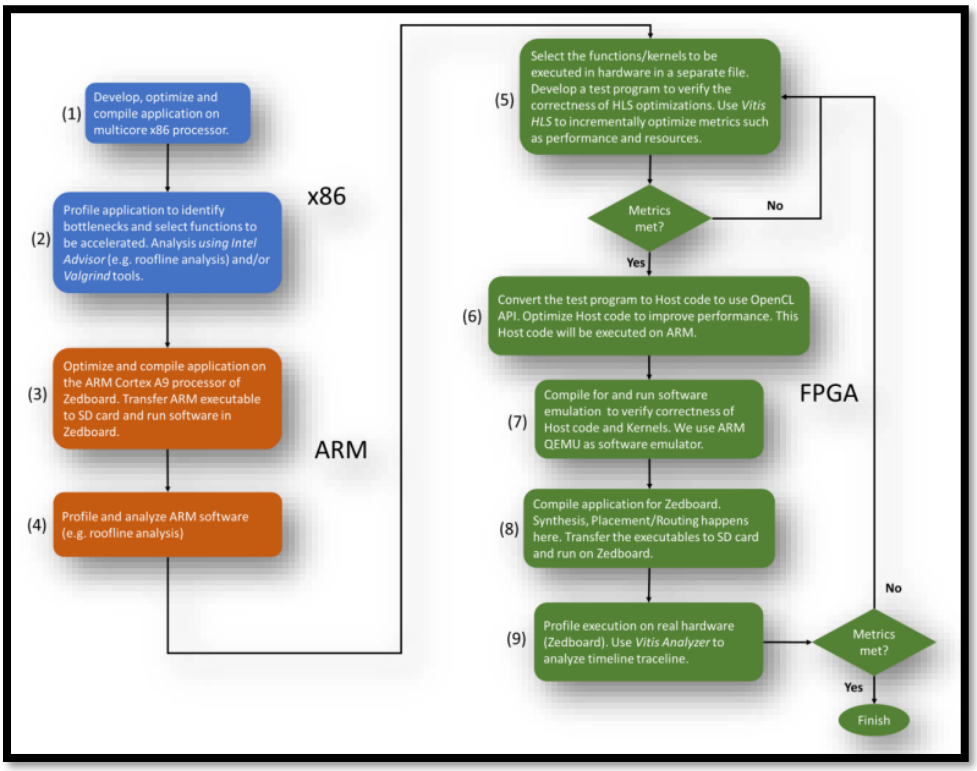
Bellas Nikolaos

Contents

Introduction

This lab covers the implementation and optimization of the Local Sequence Alignment Algorithm (LSAL) using both software and hardware approaches. LSAL is a dynamic programming algorithm widely used in bioinformatics to identify similarities between genetic sequences. Initially, the algorithm was developed and profiled on x86 and Zedboard ARM processors to evaluate its performance in software. Following this, the algorithm was implemented as a hardware accelerator on the Zedboard's FPGA fabric using Vitis High-Level Synthesis (HLS) and the OpenCL API, aiming to reduce execution time and improve performance.

The lab provides practical experience in acceleration-based computing design flow, highlighting when and how to map an application to a hardware accelerator. It discusses performance analysis, HLS techniques and optimizations, and the comparison between general-purpose CPU and FPGA-based implementations. Through incremental and test-driven development, the lab explores performance versus area tradeoffs in FPGA design, emphasizing the agility and flexibility of FPGA-based systems. The results of these experiments are documented in detail, supported by tables and figures, to illustrate the performance improvements and optimizations achieved.



Flow chart of the steps of this lab.

Implementation of the LSAL(Local Sequence Alignment Algorithm)

A Few Words About the Algorithm

In this lab, we will implement the LSAL algorithm. LSAL is used to identify similarities between a small sequence and genetic data. More specifically, the program takes two strings containing sequences of ATCG (genetic information) and searches to find the largest subsequence of the small string of length N in the large string of length M. To find the solution, two MxN matrices are constructed: the similarity matrix and the direction matrix. The similarity matrix stores the scores based on the similarity of the strings, while the direction matrix stores the orientation, so we know the path of the optimal solution. To fill the matrices, the following formula is used:

$$S_{i,j} = max \begin{cases} S_{i-1,j-1} + s(d_i, q_j) \\ \max\limits_{k \geq 1}\{S_{i-k,j} + W_k\} \\ \max\limits_{l \geq 1}\{S_{i,j-l} + W_l\} \\ 0 \end{cases} (0 \leq i \leq m-1, 0 \leq j \leq n-1)$$

where $S_{i,j}$ is the element we are calculating, $S_{i-1,j-1}$ is the top-left (northwest) element, $k = l = 1$, $W_k = W_l = -1$, $S_{i-1,j}$ is the top (north) element, $S_{i,j-1}$ is the left (west) element, and $s(d_i,q_j)$ is +2 if the corresponding elements of the sequences are the same or -1 if they are different. k, l are set to one for simplicity.

Implementation

For the implementation of the algorithm in C code, we were given a skeleton that we could use. This specific code already included a timer for measuring and comparing time, which is useful for optimizations. To fill the matrices, this algorithm has one loop from 0 to N*M, meaning the matrices are one-dimensional. For this implementation, additional calculations had to be made within the loop, such as finding the column variable (i = index % N) and the row variable (j = index / N), as well as calculations to find each element for the calculation of $S_{i,j}$ (northwest = similarity_matrix[index - N - 1]).

For reasons of simplicity as well as optimization (as we will see below), we decided to also make a second implementation with two loops to calculate the matrices. In this implementation, the additional calculations mentioned above are not needed.

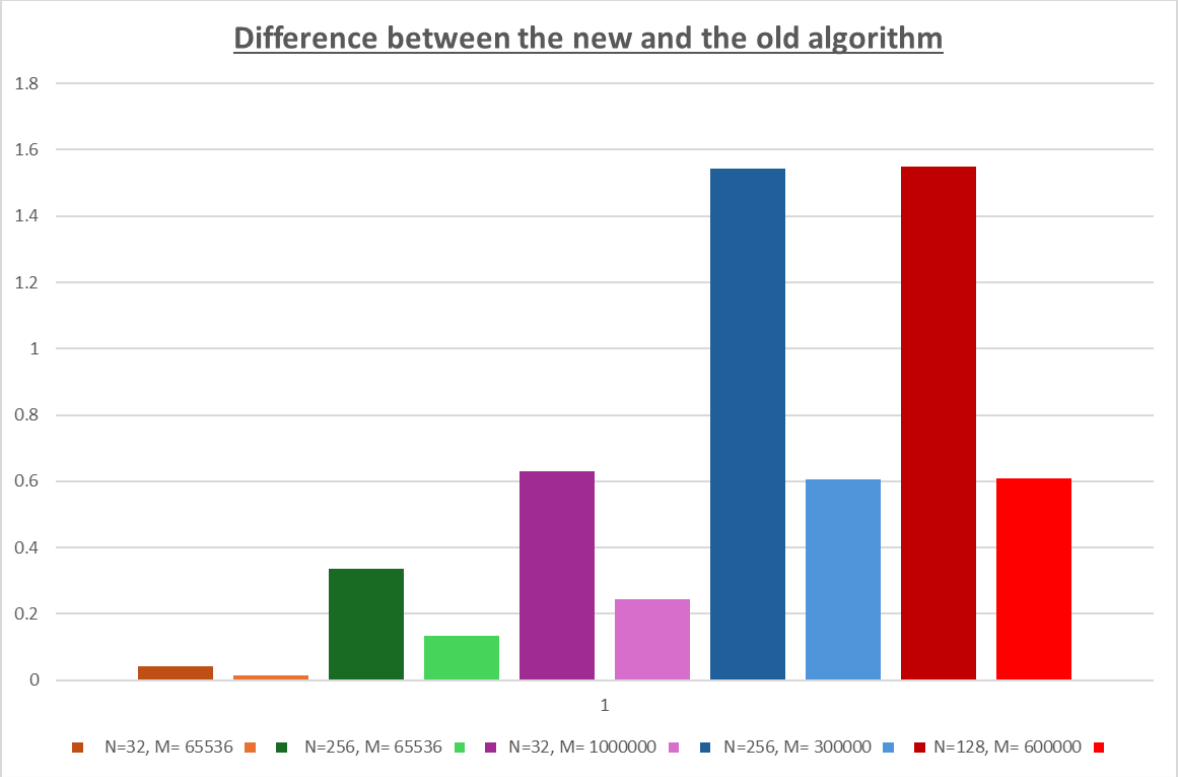
Optimizations and Results

The optimizations applied at this stage in our x86 CPU implementation concern optimizations during compilation. More specifically, optimizations -O3 and -mavx2 are used. A few words about these:

- -O3: this optimization flag enables a high level of general code optimization, including aggressive inlining, loop unrolling, vectorization, and instruction scheduling, to maximize execution speed.
- -mavx2: provides improved support for integer operations, allowing the compiler to generate code that can perform operations on multiple data points in parallel, leading to potentially significant performance improvements in applications that can leverage data parallelism.

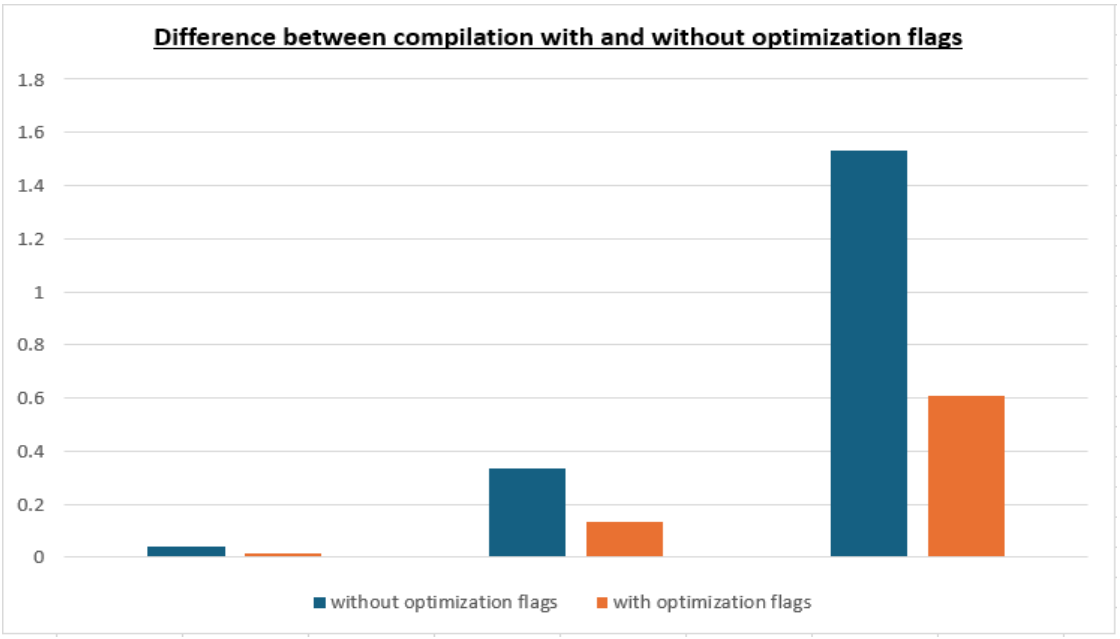
Initially, the codes were compiled without optimizations where we collected data for various input sizes and then did the same with optimizations.

x86



The graphs of the old algorithm are represented with the bold colors and the graphs of the new algorithm are represented with the pastel colors

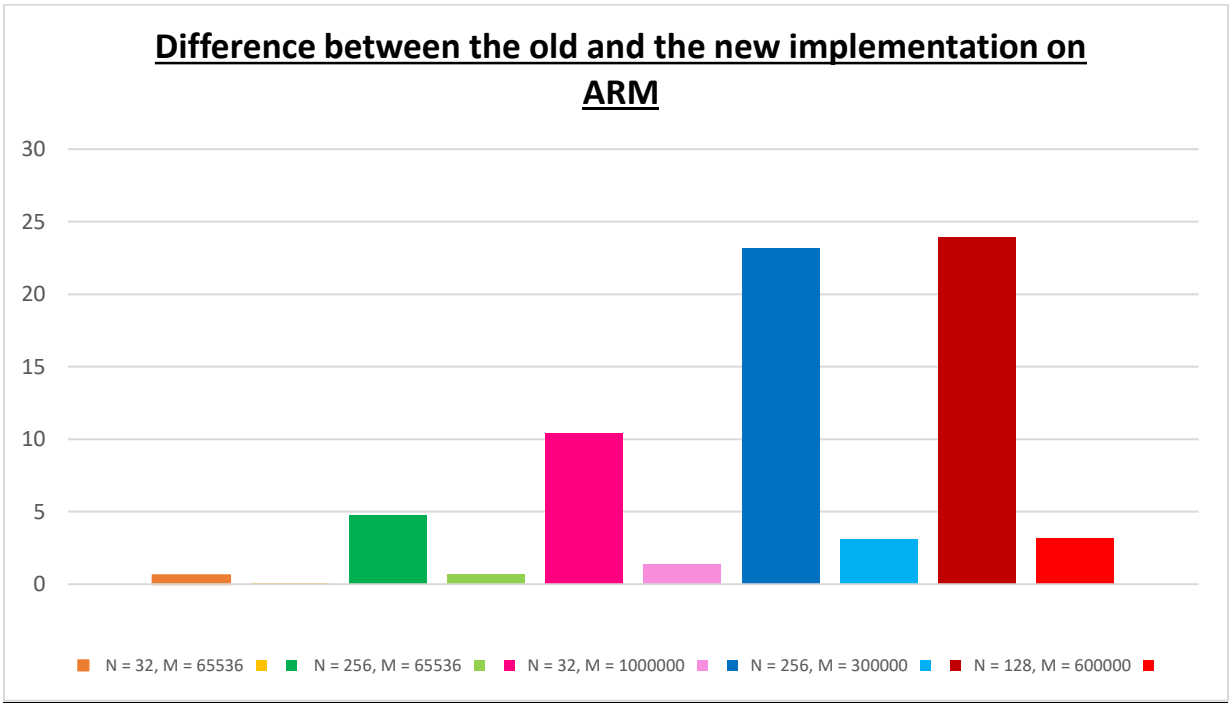
Easily we can see that the second implementation (the one with the 2 for loops) is much faster than the first one (the one with only 1 for loop). This is due to the extra calculation the first algorithm has to perform (modulo, division).



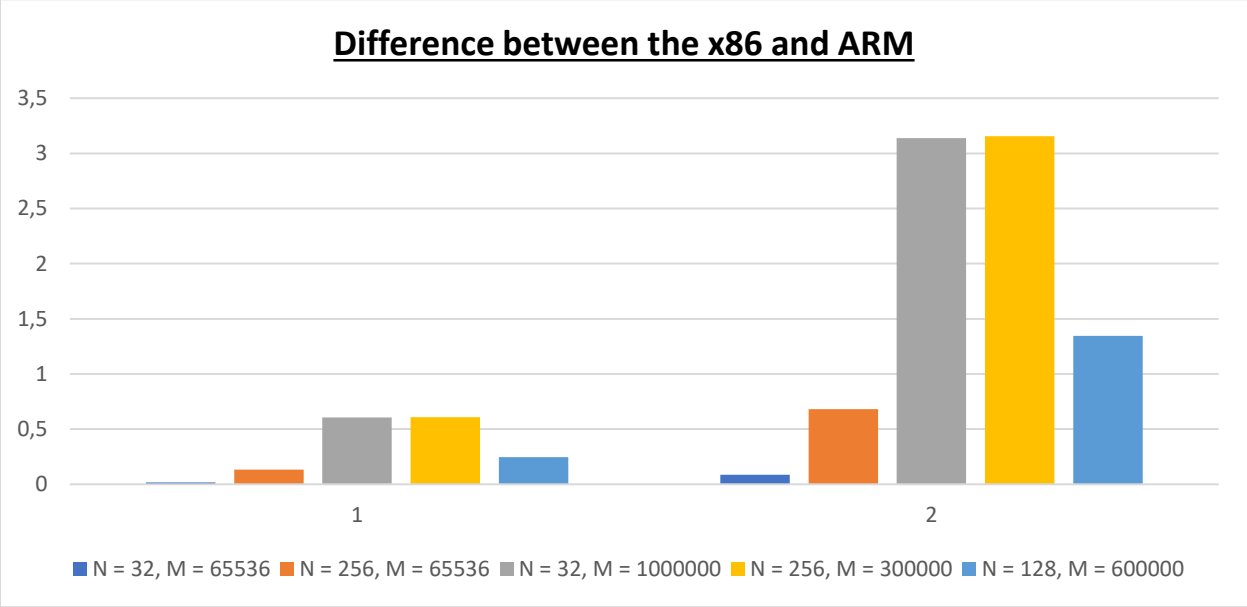
In this graph, we can observe that the flags -O3 and -mavx2 decrease the execution time. As the input numbers (N, M) get bigger, the difference between the execution time with the optimization flags and without them, becomes more and more obvious.

ARM

The next step is to optimize and compile the application on the ARM Cortex A9 processor of Zedboard. We transfer the ARM executable to an SD card and run the software on the Zedboard by using minicom.



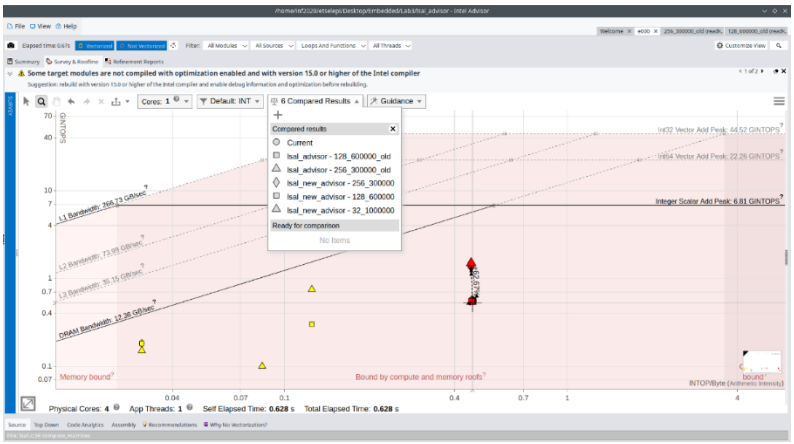
In the ARM processor, the difference between the performance of the 2 different code implementation is much more noticeable. As the sizes get bigger, the time increases exponentially.



As we observe in the graphs, the x86 performance is better than the performance of the ARM processor. The x86 CPU is faster due to a combination of higher clock speeds (the x86 CPU can execute more instructions per second), more advanced architectural features, larger and more efficient caches, better memory bandwidth, and enhanced capabilities for parallel processing and instruction execution. The ARM ZedBoard processor lacks many of these high-performance features that are present in modern x86 CPUs.

Roofline Model

The Roofline Model is visual model for assessing the performance achieved by an application relative to the limits of the hardware it is running on. In the Roofline model, an application is memory-bound if its performance point lies close to or on the memory bandwidth roof and far below the peak performance roof. The x-axis indicates how efficiently an application uses memory bandwidth relative to its workload and the y-axis indicates the application is not fully utilizing the Memory and the cashe. Our code doesn't use the cashe memory at this point.



FPGA

This part of the lab is crucial and outlines the incremental design of hardware LSAL accelerators using Vitis HLS. Writing optimized synthesizable HLS code requires a thorough understanding of the application and modifying the code to ensure Vitis HLS generates efficient hardware structures. Each optimization is applied step-by-step from the initial design until the implementation meets our performance goals and fits within the FPGA constraints. After, understanding the basic concept of High-Level Synthesis for Xilinx FPGAs by following and completing all the tutorials on the Vitis High Level Synthesis User Guide, we will synthesize and test our initial code on the Zedboard.

Vitis HLS

In order to perform HLS in Vitis, we have to create a testbench file that invokes the kernel function. The testbench code will check if the position and the value of the maximum value in the similarity matrix is correct. It will contain structures and function (ex: to create the query and database randomly) that are going to be modified as we optimize the code.

While optimizing our code, we will gather basic metrics, such as latency, iteration interval (II), clock frequency as well as resource utilization (LUTs, FFs, DSPs, BRAMs).

Non Optimized Results

The baseline accelerator may run efficiently in a CPU, but is very slow as a hardware accelerator. In the initial code that there is no optimization for the hardware we got the below results.

N	**M**	**Time(ms)**
32	32	2,448
32	65536	3111,923
256	65536	3419,310
256	300000	Failed

Considering the required time we need to achieve in this lab for sizes N=32 and M= 65536 is 10ms (the initial code is executed at 3.111,923 for these sizes), we need to perform various steps of optimizations to reach our goal.

Optimizations

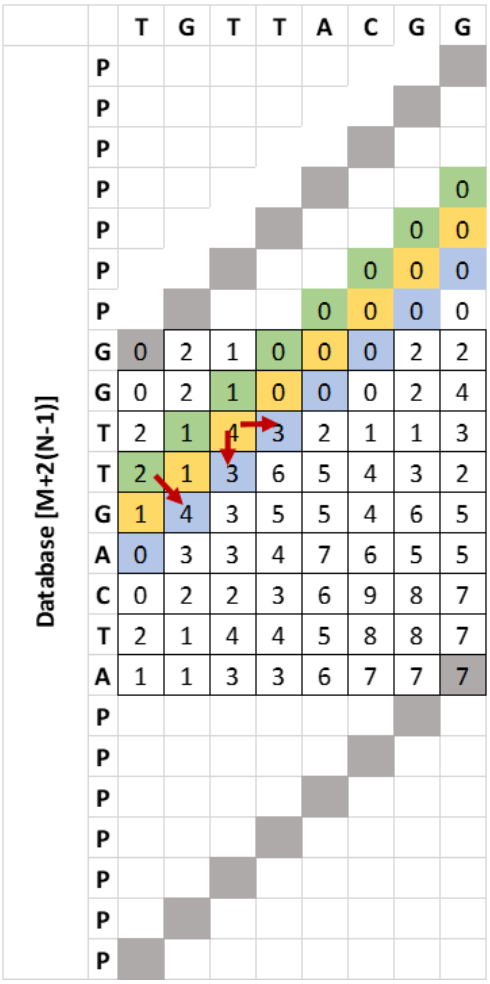
In order to begin performing the optimization, we firstly need to understand what are the bottlenecks of the baseline HW implementation. These are some potential bottlenecks:

- **Block RAM (BRAM):** If BRAMs are not used efficiently, memory access patterns may lead to reduced bandwidth. This can cause bottlenecks, especially in applications that constantly need data like ours. High memory access times can significantly slow down overall performance. Furthermore, insufficient BRAM capacity for the application's data requirements can force the use of slower off-chip memory, significantly impacting performance.

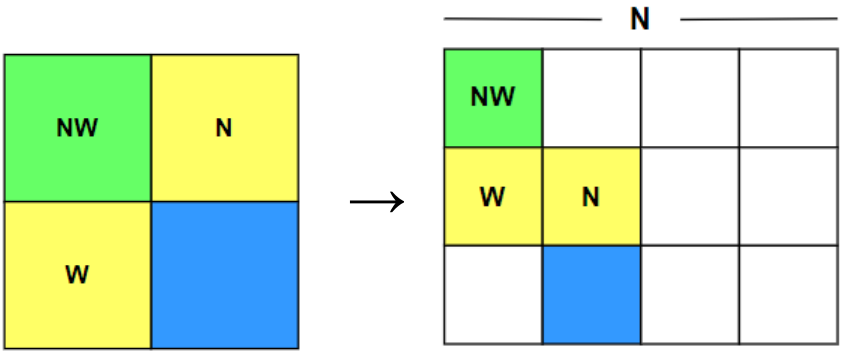
- **Resource Utilization:** The baseline implementation might not fully exploit the available compute units (e.g., ALUs, DSPs). Inefficiencies in task scheduling or load balancing can leave some processing units idle while others are overburdened.
- **Parallelism:** The baseline accelerator may not be designed to exploit parallelism effectively. Without benefiting parallel processing capabilities, the hardware accelerator cannot achieve optimal performance.
- **Algorithm Issues:** The presence of complex control logic in the hardware can lead to increased latency and reduced throughput.

Diagonal implementation

The first optimization that we consider, was minimizing the size of the similarity matrix. The baseline implementation scans sequentially all N∗M cells to update the similarity. This may be acceptable in an SW implementation, but it is clearly sub-optimal in a hardware platform that can support massive computational parallelism. The whole similarity matrix is not needed in every stage of the implementation of the algorithm to find the maximum sequence. In fact, to calculate the current element each time, we need three other elements, the north, the west and the northwest. Furthermore, the similarity matrix is only needed to find the position and the value of the max sequence, it is not needed as a whole matrix. Thus, we can solve the algorithm diagonally, using three arrays of size N instead of the whole similarity matrix. In this way, we can allocate a similarity matrix of 3 *N size, instead of a M * N one. In order to do this we have to expand our database by 2(N -1) and fill it with wrong values (P), so that they don't match with the query in the extra position, we will calculate due to the diagonal implementation. In the new similarity matrix, the three diagonal lines will be stored in the form of arrays. As we can see in the small example below the last array will be the diagonal values that need to be calculated(the blue element is calculated). The first one will contain the northwest elements(the green is the NW element of the blue) and the second one will contain the north and the west elements (the yellow) needed for the calculation of the of the last array. Every time we calculate the elements of the last array, we shift the arrays in the matrix using memcpy (the second array becomes the first, the third becomes the second and then we are ready to calculate the next diagonal line in the last array).



		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								



*Visual Representation of the conversion from a N*M similarity matrix to a 3*N similarity matrix*

The optimization of the diagonal implementation represents a significant improvement for the hardware.

1. **BRAM:** The memory requirements are drastically decreased. This not only saves on-chip memory resources but also reduces memory access times. Furthermore, it avoids the quadratic growth in memory usage associated with the size of matrix. This makes it feasible to handle larger problems.
2. **Parallelism:** Hardware accelerators, such as FPGAs can exploit this parallelism to perform multiple computations simultaneously, significantly speeding up the execution.
3. **Data Patterns:** Using three arrays to store only the necessary elements needed for each computation simplifies data management. The complexity of data access patterns is reduced, leading to more efficient use of memory bandwidth, reduced latency and higher throughput.

Buffering

The function `compute_matrices` receives `direction_matrix` (size M * N) and `database` (size M+2(N-1)) as inputs. Given the significant size of M, it is inefficient to maintain the entire direction matrix in BRAM. To address this, we employ a strategy using buffers and the `memcpy` function, similar to the diagonal implementation for the similarity matrix.

Database Buffer

Instead of storing the entire database in BRAM, we can optimize memory usage by employing a database buffer. Only a small segment of the database is needed at any given time to calculate the elements of each array in the diagonal similarity matrix. Specifically, for each array of size N, we compare the query (N) with the respective section of the database (N). By using a database buffer of N elements, we significantly reduce the amount of memory needed in BRAM. At the end of each iteration, the elements in the database buffer are shifted. The element that is no longer needed for comparison is discarded, and the new element required for the next comparison is added from the database. This shifting mechanism ensures that the buffer always contains the relevant segment of the database.

Here's the benefits of this implementation:

1. **BRAM:** Instead of occupying BRAM with the entire database, we only store the necessary segments, freeing up valuable BRAM resources.
2. **Enhanced Performance:** With a smaller, focused buffer, data access times are reduced, leading to faster computations and overall improved performance. The use of buffers allows for better parallel processing.

Direction Buffer

We address the significant size of the direction matrix (M * N) by implementing a smaller direction buffer of size N. This buffer holds only the necessary data for the current diagonal computation. In each iteration, the code computes a diagonal line for the direction matrix. These results are temporarily stored in the smaller direction buffer. Once the diagonal line is computed, it is transferred from the direction buffer to the direction matrix using the memcpy function. This approach ensures that only relevant data is stored in the direction buffer, and the final results are efficiently added to the larger direction matrix.

The benefits of this optimization are similar to the ones of the database buffer. It reduces memory usage, improves data handling, enhances performance, and ensures scalable and flexible implementation. In addition, the use of memcpy for transferring data from the buffer to the direction matrix ensures quick and efficient updates, minimizing latency.

Max_Value_Buffer

To optimize the handling of max value in our algorithm, we implemented a final buffer that solves dependency issues, we identified during analysis with HLS Vitis. Initially, the maximum value was stored in the max_index[3] (parameter of the function), with two positions for I and j indices and one for the maximum value itself. This setup created considerable dependencies that affected performance, since we constantly need to compare and change its values. To improve this, we replaced our structure with a 2D buffer. The first row of the buffer stores indices in the format i * N + j, representing the position in the similarity matrix. The second row holds the corresponding maximum values found in each column of the similarity matrix. Throughout the computation, instead of immediately determining the maximum value, we store potential maximum values in the buffer. After the algorithm finishes, we perform a linear search on the max_value buffer to identify the maximum value. By deferring the determination of the maximum value until the end of the algorithm, we reduce dependencies during computation.

Loop Pipeline

In hardware design using HLS, the #pragma HLS pipeline directive offers the optimization techniques, loop pipelining and loop flattening. These techniques maximize performance, resource utilization, and efficiency.

Loop pipelining allows multiple iterations of the loop to execute simultaneously, increasing the throughput of the loop. Each iteration of the loop can start before the previous iteration finishes. By allowing multiple iterations to be processed simultaneously, the overall latency to process all iterations is reduced.

Loop flattening involves transforming inner loops into a single loop. With a single loop structure, the control logic becomes simpler. Flattened loops often have fewer dependencies between iterations, making them easier to pipeline effectively.

Array Partitioning

To benefit from the parallelism provided by loop pipelining, we have to ensure that the hardware resources can support concurrent operations. The algorithm relies on accessing multiple elements of the similarity matrix, the max_value_buffer, and the database_buffer simultaneously. However, the default configuration of BRAM, which offers only two ports, cannot accommodate the resources we need and ultimately create bottleneck and dependencies.

The array partitioning can be adapted to different sizes and types of data structures. To solve this problem, we implement complete array partitioning. By partitioning the arrays, we eliminate the dependencies and bottlenecks caused by limited BRAM ports. Partitioning optimizes the use of memory resources, ensuring that all available hardware capabilities are utilized effectively.

Arbitrary-Precision Integers

To further optimize our implementation, we utilize arbitrary precision integers. Previously, we used standard data types such as int, short, and char. However, these types are often unnecessarily large for our needs, leaving many unused bits that consume valuable BRAM space. While this might not be an issue for smaller values of N and M, for larger designs, these wasted bits become significant. By using arbitrary precision integers, we can set exactly the number of bits required for each element, leading to more efficient memory usage. We concluded that it is beneficial to use arbitrary precision integers for all arrays and matrices in the algorithm.

Database and Query

Initially, the database and query were of type char, which has 1 byte (8 bits). However, the database only needs to store 5 elements (A, G, C, T, P), requiring only 3 bits (or 4 bits for convenience). By using ap_int<32>, which has 32 bits per entry, we can fit 8 elements per entry (32/4). Consequently, the size of the query and database buffer can decrease from N to N/8+1. The function compress() is used to convert ASCII characters into these compressed bit representations.

A	0
C	1
G	3
T	2
P	4

Max Value Buffer

The max value buffer requires special consideration. The types of the elements in the max value buffer depend significantly on the sizes M and N. To represent the vertices of the max value, the element type must fit the number $i \times N + j$, where the maximum value is $(M+2(N-1)) \times N + N$.

Similarity Matrix

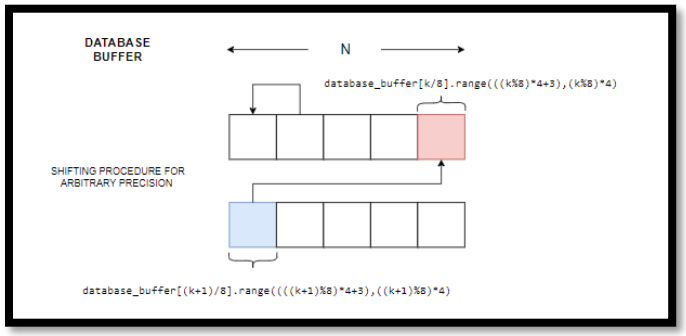
Initially, the similarity matrix was of type int. According to our algorithm, the maximum value in the similarity matrix is 2N (if the entire query matches a section of the database). We created two different implementations based on the size of N.

1. **For smaller N**: When the maximum value can fit in 1 byte (i.e., $2N \leq 255$ or $N \leq 128N$), we set the similarity matrix to ap_int<512>. Each entry of the similarity matrix can fit 64 bytes (512 bits), accommodating 64 elements per entry. The entire similarity matrix has 3N elements, so we need $\lceil 3N/64 \rceil$ entries.

2. **For larger N**: When N>128, the maximum value requires 2 bytes.

Example: Locating the Exact Position of an Element Using Ranges

Assuming N = 32, the database buffer has 4bit elements and is of type ap_int<32>, so 8 elements fit in each line and we will need a total of 4 lines. When, we implement the sifting we need to pinpoint the exact position of the element we want to shift and the position it is shifted to. We use database_buffer[k/8] to be placed to the correct row and .range(((k%8)*4+3),(k%8)*4) to be placed in the correct column and start of the element. We notice that the range is 4 bits, just like the size of the elements.



Optimized Results

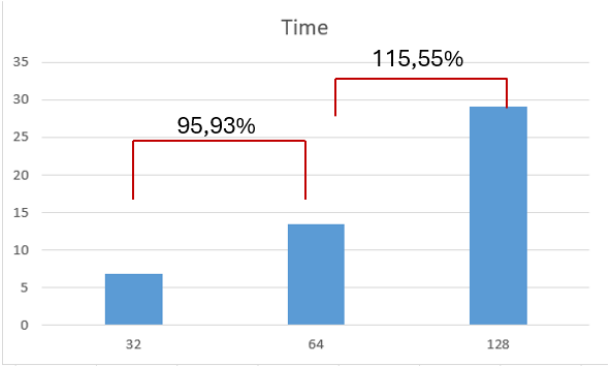
All the previously mentioned techniques were implemented in an FPGA. First, we ran our code with parameters N=32 and M=65536. The code executed in 6.89 ms with an Interval II=8 using 22% BRAMs and 20%LUTs . The dependency that prevented our code from achieving a smaller II interval was the max_value variable. This variable is used to search for the maximum value among three possible positions (west, north, northwest). It is read four times (including the center position) in the if statements and, in the worst case, is also written four times.

Next, we increased the N value to 64 while keeping M constant at 65536. The changes required in the code to accommodate the larger N value were limited to the similarity buffer. As previously mentioned, the similarity buffer is an ap_int<512> array. We used 1 byte to represent each value in the similarity buffer, allowing for a maximum value of 2^8=256. This is sufficient for N values up to 128 because the highest possible value in the similarity array occurs when every query character matches each database character, resulting in 128*2=256. For N=32, we need *32*3 (3 diagonals) = 96 elements. Each row of the similarity buffer consists of 64 bytes / 1 byte per element = 64 elements. Thus, we need 2 rows from the similarity buffer for N=32 and for N=64, we need to store 64**3* = 192 elements / 64 elements per row = 3 rows. The code ran at 13.5 ms with an interval II=18, 22%BRAMs and 55%LUTs. This means that we got an 95,93% increase in the run time .

Next we increased again the N value to 128 keeping again M constant at 65536. Again the only changes in the code were the similarity buffer size. Now we set the size at 6 because 128*3=384/64=6. For N=128 we increased also the pipeline II interval and we set the II to 38. Firstly, we have left the HLS VITIS tool to get the ideal pipeline intervale II=1 , but that couldn't be synthesizable. The idea behind choosing II =38 was that for N=64 we used an II = 18 and as I doubled the N I double the II interval. For smaller II interval the code couldn't build and as an error it said timimng violation and that it couldn't get the clock. Finally, the time we got was 29.3ms again the BRAMS was at 22% but the LUTs according to HLS Vitis were 91%.

In order to check how the M value affects the code runtime we increased M to 300000. For N=32 and M=300000 and keeping the same code the final time was 28,10 ms with an interval II=8.

As we see in the graph, as the value of N is doubled time gets increased by 95,93% when increasing N from 32 to N=64 and increased by 115,55% from N=64 to N=128. This is due the fact that the iteration interval II is also doubled. For N=32 we had II=8, and for N=64 we got II=18 and finally for N=128 the II was 38. It is important to note that the BRAM usage remains constant as N increases because the buffers are fully partitioned and implemented as Flip-Flops. However, the number of LUTs increases due to this complete partitioning.

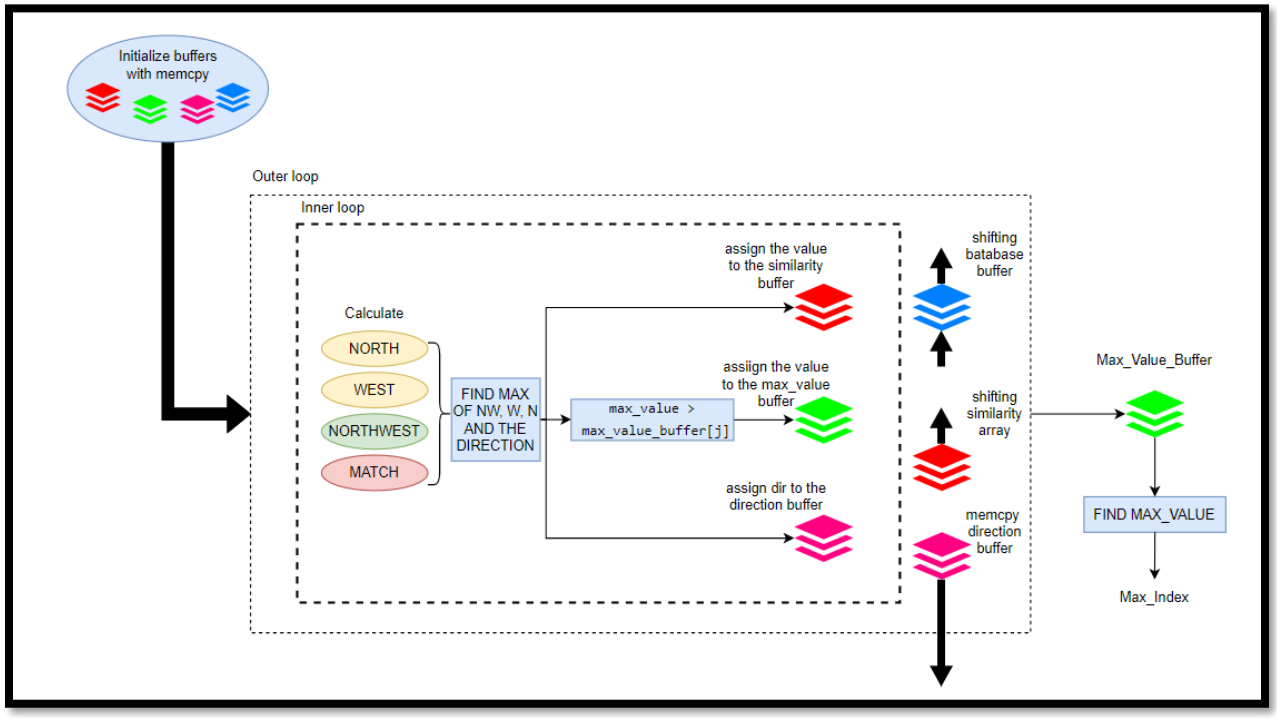


Run Time for different N

For the last result with the M set to 300000 we get approximately the same result as the experiment with N = 128 and M = 65526. For M set to 300,000, the outer loop's iterations increase, leading to a longer total runtime. As previously mentioned, the initiation interval (II) remains at 8, indicating that we did not lose parallelism. The increased total time is solely due to the higher number of iterations. Conversely, for N=128 and M=65,536, the execution achieves an II of 38, indicating a loss of parallelism. This loss of parallelism directly affects performance, causing a significant increase in execution time.

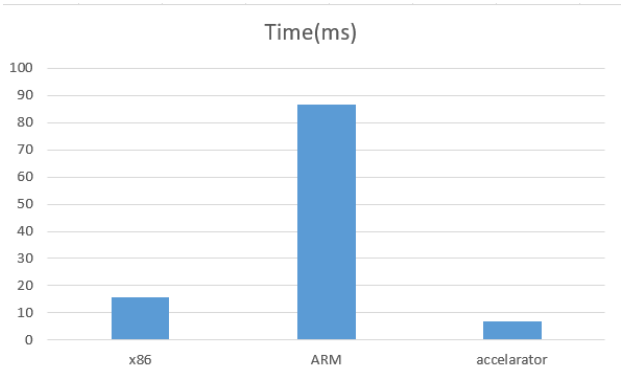
The buffers are not influenced by the size M and they are mostly affected by the value N. Therefore, a substantial increase in M may lead to a longer execution time, but the increase in resource usage will be minimal.

Conclusion - Comparing Results



Final design-Graph

In this section, we are going to compare the results from the three implementations (x86, ARM processor and hardware accelerator). The below picture depicts the run times for N=32 M=65536. As we can see the accelerator gets a speedup x2,3 over x86 and x12,61 over ARM processor. The execution time of the accelerator is decreased by 56,66% compared the execution time of the x86 and by 92,07% compared with the ARM. As we expected, software running on a processor, no matter how well optimized it is, is usually slower than a hardware accelerator implementing the same functionality.



References

1. Samuel Williams, Andrew Waterman, David Patterson,. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM 52(4): 65-76 (2009)*

2. https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html

3. Vitis High-Level Synthesis User Guide (UG1399).

4. Vitis Application Acceleration Development (UG1393)

5. Vivado High Level Synthesis Tutorial (UG871).

6. https://www.sciencedirect.com/topics/computer-science/roofline-model