UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

FPGA IMPLEMENTATION AND OPTIMIZATION OF A LOCAL SEQUENCE ALIGNMENT ALGORITHM

Diploma Thesis

Anastasia Eirini Tsitsopoulou

Supervisor: Nikolaos Bellas

February 2023

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

FPGA IMPLEMENTATION AND OPTIMIZATION OF A LOCAL SEQUENCE ALIGNMENT ALGORITHM

Diploma Thesis

Anastasia Eirini Tsitsopoulou

Supervisor: Nikolaos Bellas

February 2023

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΒΕΛΤΙΣΤΟΠΟΙΗΣΗ ΕΝΟΣ ΑΛΓΟΡΙΘΜΟΥ LOCAL SEQUENCE ALIGNMENT ΣΕ FPGA

Διπλωματική Εργασία

Αναστασία Ειρήνη Τσιτσοπούλου

Επιβλέπων: Νικόλαος Μπέλλας

Φεβρουάριος 2023

Approved by the Examination Committee:

Supervisor **Nikolaos Bellas**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Christos Antonopoulos**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Dimitrios Katsaros**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Date of approval:

Acknowledgements

First and foremost, I would like to express my immense gratitude to my advisor, Prof. Nikolaos Bellas, for his unwavering support and valuable guidance throughout the development of this thesis. I would also like to thank PhD candidates Alexandros Patras and Maria Gkeka for patiently answering my questions and warmly welcoming me in 314B.

To mom & dad, thanks for constantly figuring out the solutions to the problems I cannot solve on my own. This is your accomplishment as much as it is mine. Never stop being as fascinating as you are.

To my friends, overlapping or not with family, if I have been half as helpful to you as you have been to me, I will consider myself a successful person. Thanks for each and every push.

Last but definitely, *definitely* not least, to R: I would not be writing this if not for you. Thanks for dragging me over the finishing line. We made it.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

The declarant

Anastasia Eirini Tsitsopoulou

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα διπλωματική εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελούν αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλουν οποιασδήποτε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχουν έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/ δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Δηλώνω επίσης ότι τα αποτελέσματα της εργασίας δεν έχουν χρησιμοποιηθεί για την απόκτηση άλλου πτυχίου. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής.

Η Δηλούσα

Αναστασία Ειρήνη Τσιτσοπούλου

ABSTRACT

Field-programmable gate arrays (FPGAs) are integrated circuits with the ability to be configured by a designer according to the needs of a specific task. As a result, FPGAs combine the computational flexibility of a software platform with the advantages that parallel execution on a hardware platform offers. This combination renders them a valuable tool in tackling large-scale problems with vast demands in fast execution and limited resources, leading to their consistent use in various domains, including Bioinformatics, where methods are characterized by analyses of large and complex datasets of genetic information.
One of these methods, Local Sequence Alignment (LSAL), discovers areas of similarity between a query and a DNA database. This thesis aims to implement the LSAL algorithm on an FPGA, with the use of a high-level synthesis (HLS) tool, which offers the ability to reconfigure the hardware by manipulating a C++ code, and eventually try to achieve a performance that is comparable with an optimized software implementation.

ΠΕΡΙΛΗΨΗ

Οι πλακέτες FPGAs (field-programmable gate arrays) είναι κυκλώματα τα οποία παρέχουν στον προγραμματιστή την δυνατότητα να διαμορφώσει την λειτουργία τους σύμφωνα με τις ανάγκες ενός συγκεκριμένου προβλήματος προς επίλυση. Ως αποτέλεσμα, οι FPGAs συνδυάζουν την ευελιξία ενός συστήματος software με τα πλεονεκτήματα που προσφέρει η παράλληλη εκτέλεση πάνω σε hardware. Χάρη σε αυτόν τον συνδυασμό, οι πλακέτες FPGA αποτελούν ένα πολύτιμο εργαλείο στην προσπάθεια επίλυσης προβλημάτων μεγάλης κλίμακας, όπου προέχει η γρήγορη εκτέλεση και η οικονομία σε υλικό, το οποίο οδηγεί στην συνεχή χρήση τους σε διάφορους τομείς, όπως η Βιοπληροφορική. Οι μέθοδοι της βιοπληροφορικής συχνά χαρακτηρίζονται από αναλύσεις μεγάλων και περίπλοκων σετ δεδομένων.

Μια από αυτές τις μεθόδους, ονομαζόμενη Local Sequence Alignment (Τοπική Ευθυγράμμιση Αλληλουχιών), εντοπίζει περιοχές μεταξύ 2 αλληλουχιών DNA όπου εμφανίζονται ομοιότητες. Αυτή η διπλωματική στοχεύει στην υλοποίηση του αλγορίθμου LSAL σε μια πλακέτα FPGA, με την χρήση ενός εργαλείου high-level synthesis (HLS), το οποίο δίνει στον προγραμματιστή τη δυνατότητα να διαμορφώσει την λειτουργία της πλακέτας γράφοντας κώδικα σε γλώσσα προγραμματισμού υψηλού επιπέδου, και εν τέλει στην επίτευξη μιας υλοποίησης που μπορεί να συναγωνιστεί έναν βελτιστοποιημένο κώδικα software.

Table of Contents

Chapter 1

1.1 General Overview

1.1.1 FPGAs

A field-programmable gate array (FPGA) is an integrated circuit with the ability to be configured by a designer according to the needs of a specific task. In contrast with application-specific integrated circuits (ASIC), FPGAs can be reprogrammed after manufacturing, thanks to the array of programmable blocks they contain, alongside the interconnects that wire them together. The designer can guide the reconfiguration of an FPGA with the use of a hardware description language (HDL). Besides the programmable logic blocks, FPGAs also contain memory elements, such as flip-flops and complete memory structures. [1]

As hardware platforms, FPGAs provide great levels of parallel computing. Their parallel nature can therefore be taken advantage of in using FPGAs as hardware accelerators to perform specific tasks in more efficient ways in terms of time and resources when compared to a software platform. As a result, FPGAs are widely used in a number of large-scale applications in various domains, Bioinformatics being one of them.

1.1.2 Sequence Alignment in Bioinformatics

The field of Bioinformatics is responsible of developing methods and software tools for studying and analyzing biological data to better understand evolutionary aspects of molecular biology [2]. The rapid expansion of genetic data calls for approaches that rely on parallel computing, in order to create tools that are able to handle large and complex biological datasets of genetic information as efficiently and fast as possible. The bioinformatics method we study in this thesis is sequence alignment, a method of arranging sequences of DNA, RNA, or proteins to identify regions of similarity between two sequences, usually using databases of large sizes.

Specifically, the algorithm we study is the Smith-Waterman algorithm for Local Sequence Alignment, proposed by Temple F. Smith and Michael S. Waterman in 1981 [3]. As a dynamic programming algorithm, it is guaranteed to find the optimal alignment with respect to the scoring system we empirically choose. The goal of the algorithm is to match two DNA (in our case) sequence, a query of size N and a database of size M, with M usually being vastly greater than N. Being a local sequence alignment method, matching means the algorithm locates segments where the two sequences share similarities. The output produced will be a similarity matrix S, a direction matrix D, as well as the index of the largest value in the similarity matrix, which are afterwards used in obtaining the optimal alignment.

The goal of this thesis is to:

- Study the Smith-Waterman Algorithm for Local Sequence Alignment,
- develop and optimize a software implementation running on an x86 platform, and
- optimize a hardware implementation running on a FPGA, attempting to reach an execution faster than the software one.

1.2 The Smith-Waterman Algorithm

Similarity Matrix

Given a query sequence A of length N to be aligned with a database sequence B of length M, we construct two matrices of size $N \cdot M$, the similarity and the direction matrix. The similarity matrix S is filled by scoring each element from left to right, top to bottom, while making one-to-one comparisons between all components in the two sequences according to the equation:

$$S_{i,j} = \max \begin{cases} S_{i-1,j-1} + s(a_i, b_j) \\ \max\limits_{k \geq 1}\{S_{i-k,j} + W_k\} \\ \max\limits_{l \geq 1}\{S_{i,j-l} + W_l\} \\ 0 \end{cases} \quad (0 \leq i \leq m-1, 0 \leq j \leq n-1) \qquad (1.1)$$

Where:
$S_{i-1,j-1} + s(a_i, b_j)$ is the aligning score of the base pair a_i, b_j, called the northwest value,
$S_{i-k,j} + W_k$ is the score when a_i is at the end of a gap of length k, called the north value,
$S_{i,j-l} + W_l$ is the score when b_j is at the end of a gap of length l, called the west value.

The value $s(a_i, b_j)$ is the similarity score, chosen to be positive in case of a match between a_i and b_j, and relatively lower in case of a mismatch. In this version of the algorithm, we choose the following similarity function:

$$s(a_i, b_j) = \begin{cases} +2 & if\ a_i = b_j \\ -1 & if\ a_i \neq b_j \end{cases} \qquad (1.2)$$

The parameter W is the gap penalty and k, l are the gap lengths. Our version of the algorithm uses $W = -1$, and $k = l = 1$, so Equation (1.1) can be written as:

$$S_{i,j} = \max \begin{cases} S_{i-1,j-1} + s(a_i, b_j) \\ S_{i-1,j} - 1 \\ S_{i,j-1} - 1 \\ 0 \end{cases} \quad (0 \le i \le m-1, 0 \le j \le n-1) \qquad (1.3)$$

When filling the matrix, it is important to take into consideration that the first row should have northwest and north values of 0 and the first column should have northwest and west values of 0.

An element with a negative score indicates no similarities (from any direction) between the sequences up to this particular element. Therefore, the score is set to zero to eliminate influence from previous alignment.

Direction matrix

While filling the similarity matrix, the direction matrix D is also being constructed, with each element $D_{i,j}$ being assigned one of four values-directions: north, west, northwest and center, depending on which of the four values of equation (1.3) was chosen as the maximum for $S_{i,j}$.

$$D_{i,j} = \begin{cases} NW, & if\ S_{i,j} = S_{i-1,j-1} + s(a_i, b_j) \\ N, & if\ S_{i,j} = S_{i-1,j} - 1 \\ W, & if\ S_{i,j} = S_{i,j-1} - 1 \\ C, & if\ S_{i,j} = 0 \end{cases} \qquad (1.4)$$

In case of a draw between the four competing values for $S_{i,j}$, the priority followed is C, NW, N, W.

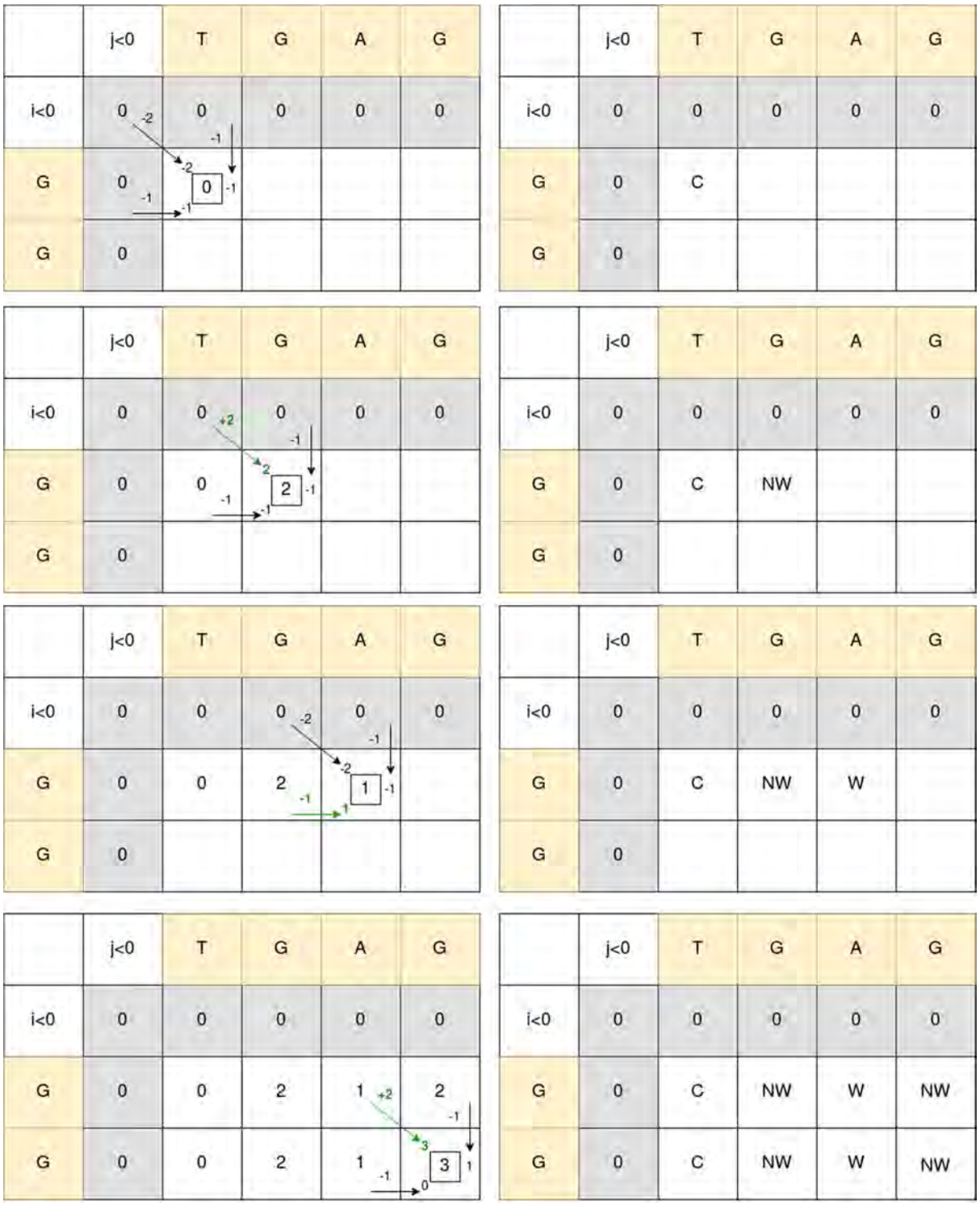
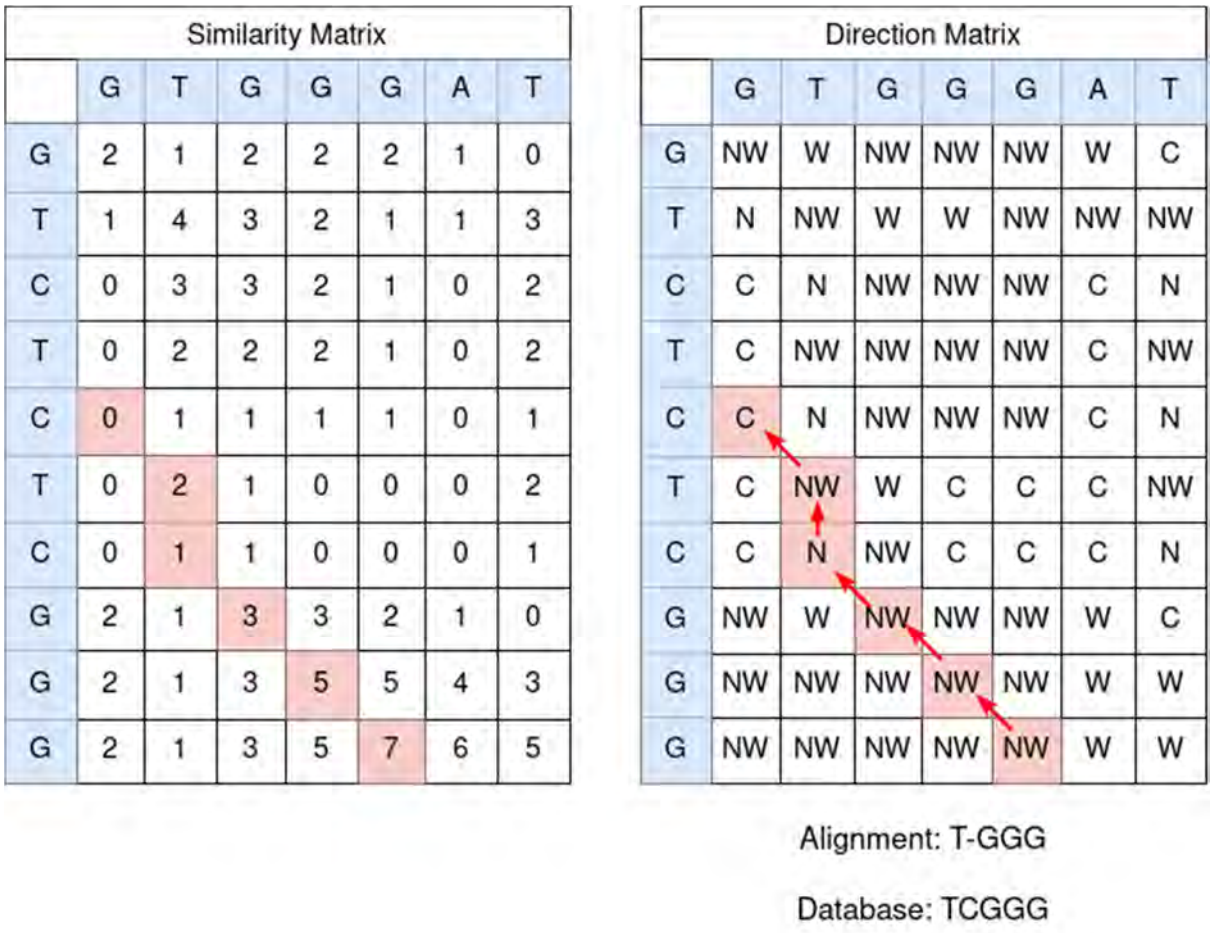


Figure 1.1: Several steps of filling the similarity and the direction matrices

Traceback

As the similarity and direction matrices are being filled, we locate the cell with the largest similarity score. Starting from this cell, we move following the directions indicated by the

direction matrix until we reach a cell of value C in the direction matrix or 0 in the similarity matrix. This backtracing produces the optimal local sequence alignment.



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

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

Figure 1.2: The traceback process and the alignment produced.

Reaching a north or west value on the direction matrix while backtracing introduces a gap into the alignment result, while reaching a northwest value introduces a match or a mismatch. In case of multiple highest scores, we use the first one we encounter. [4]

The traceback process, which has a linear complexity of $O(m+n)$, will not be implemented within the confines of this thesis, as we focus on accelerating the most demanding parts of the algorithm in terms of execution time and resources. As a result, our code is limited to producing the position of the maximum value of the similarity matrix as well as the direction matrix, so that the traceback can take place at a later step.

Our initial implementation of the Smith-Waterman LSAL algorithm is presented below:

Algorithm 1: Smith-Waterman LSAL Algorithm

```
Input: string1[N], string2[M];
Result: similarity_matrix, direction_matrix, max_index
  max_value = 0;
  for each i,j:
    north = northwest = west = 0;
    if i,j in first row then
      west = similarity_matrix[i, j-1];
    else if i,j in first column then
      north = similarity_matrix[i-1, j];
    else
      north = similarity_matrix[i-1, j];
      northwest = similarity_matrix[i-1, j-1];
      west = similarity_matrix[i, j-1];
    end
    match = (string1[j] = string2[i]) ? MATCH : MISMATCH;
    north += GAP_PENALTY;
    northwest += match;
    west += GAP_PENALTY;
    similarity_matrix[i,j], max_pos = find_max_value(north, northwest, west);
    direction_matrix [i,j] = max_pos;

    if max_value < similarity_matrix[i,j] then
      max_value = similarity_matrix[i,j];
      max_index = (i,j);
    end
  end
```

Algorithm 2: find_max_value function

```
Input: north, northwest, west
Result: max_value, pos;

  if max(north, northwest, west, 0) = north then
    max_value = north;
    pos = N;
  end
  if max(north, northwest, west, 0) = northwest then
    max_value = northwest;
    pos = NW;
  end
  if max(north, northwest, west, 0) = west then
    max_value = west;
    pos = W;
  end
  if max(north, northwest, west, 0) = 0 then
    max_value = 0;
    pos = C;
  end
```

Chapter 2

2.1 Software Implementation

2.1.1 Software Optimization

We begin by executing our initial code on an x86 system with the following resources:

- Intel® Core™ i7-7500U CPU @ 2.70GHz
- Logical CPU Count 4
- Linux Operating System
- 6GB RAM
- L1 Data Cache: 64kB
- L1 Instruction Cache: 64kB
- L2 Cache: 512kB
- L3 Cache: 4MB

N	M	execution time (ms)
32	32768	26,326
32	65536	28,319
32	131072	40,636
64	32768	33,252
64	65536	41,924
64	131072	78,595
128	32768	42,535
128	65536	78,286
128	131072	155,848

Table 2.1: Execution times of the initial code on the x86 system for several pairs of N, M

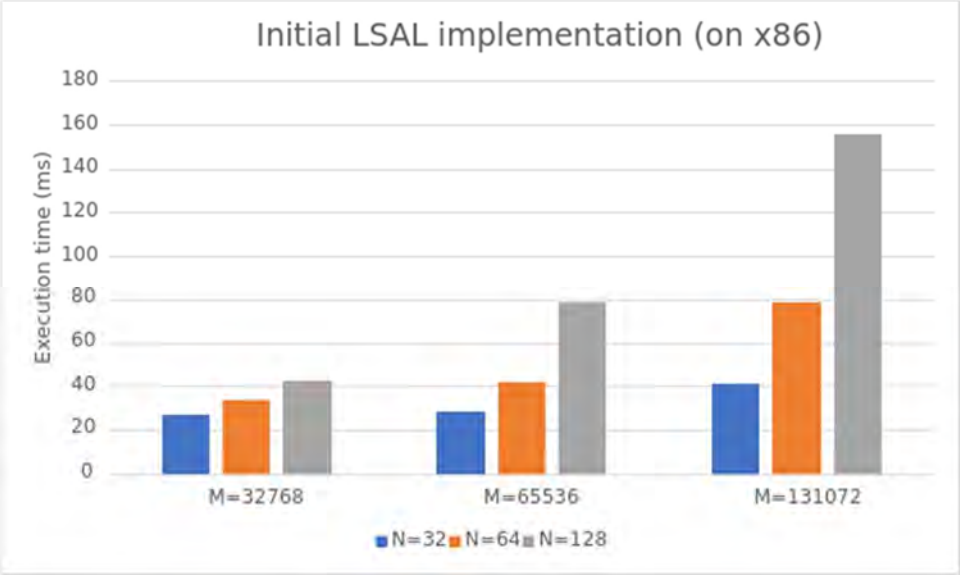


Figure 2.1: Execution times of the initial code on the x86 system for several pairs of N, M (graph)

We want to achieve a faster execution time on the x86 in order to then challenge the hardware implementation as much as possible. In this attempt, several software optimizations are applied to the initial code, culminating in the following software implementation:

Algorithm 3: Smith-Waterman LSAL Algorithm - Optimized

```
Input: string1[N], string2[M];
Result: similarity_matrix, direction_matrix, max_index
  max_value = 0;
  for i=1; i<M; i++:
    for j=1; j<N; j++:
      match = (string1[j] = string2[i]) ? MATCH : MISMATCH;
      north = similarity_matrix[i-1, j] + GAP_PENALTY;
      northwest += similarity_matrix[i-1, j-1];
      west = similarity_matrix[i, j-1] + GAP_PENALTY;
      similarity_matrix[i,j], max_pos = find_max_value(north, northwest+match, west);
      direction_matrix [i,j] = max_pos;

      if max_value < similarity_matrix[i,j] then
        max_value = similarity_matrix[i,j];
        max_index = (i,j);
      end
    end
  end
```

In this implementation, the values N and M have been augmented by one, so that we have matrices of size N+1 * M+1, where the first row and column is initialized with zeros. This eliminates the need for the if-statement in the initial version of the algorithm.

N	M	execution time (ms)
32	32768	22,522
32	65536	24,908
32	131072	35,015
64	32768	25,084
64	65536	36,333
64	131072	66,794
128	32768	36,392
128	65536	66,843
128	131072	133,545

Table 2.2: Execution times of the optimized code on the x86 system for several pairs of N, M

Figure 2.2: Comparison between execution times of the initial and the optimized code (graph)

As we can observe in Table 2.2 and Figure 2.2, the final optimized software implementation succeeds at having a faster execution time, with an average speedup of 1,18 across executions for different N, M pairs.

2.1.2 Software Profiling using Roofline Models

Apart from only measuring the execution time for various query and database lengths, it is useful to also study the roofline model of the LSAL algorithm. The roofline model is a visualization of performance estimates of an application, used to determine whether the application is bound by memory bandwidth or computational intensity, as well as show the maximum performance we can achieve with the current available hardware. [5]

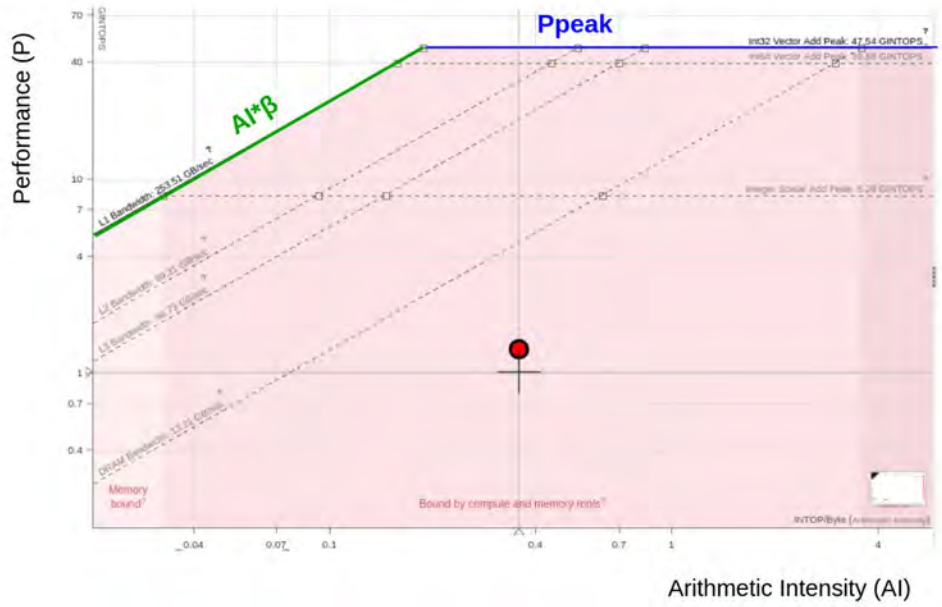


Figure 2.3: A basic roofline model

The roofline plot is derived by the equation:

$$P \leq \min(P_{peak},\ AI \cdot \beta) \tag{2.1}$$

Where:

- P_{peak} represents the maximum performance we can achieve, based on the given hardware resources, such as number of cores and functional units, etc. Performance is measured in GINTOPS (Giga Integer Operations per Second).
- The arithmetic intensity (AI) of the application is the following ratio:

  $$AI = \frac{\#\ executed\ operations}{\#\ bytes\ transferred\ between\ memory\ \&\ CPU} \tag{2.2}$$

  AI is measured in INTOP/byte.
- β is the memory bandwidth and depends, as P_{peak} only on the specific platform onto the application is being executed.

The red dot represents the performance an implementation achieves. If the dot falls into the left, triangle area, the performance is bound by the AI*β line, therefore the implementation is considered to be of low intensity and subsequently, memory bound. Alternatively, if the dot falls into the right area, the implementation is of high intensity, thus CPU bound, as it is limited only by the platform resources.

The goal is to push the dot, or the dots, upwards and to the right, toward the horizontal line P_{peak}. An upwards push translates to better parallelism, while a push to the right translates to less dependence on memory bandwidth. [6]

The roofline models can be automatically computed using Intel Advisor, a design and analysis tool by Intel. For several pairs of N and M, we compare the roofline models between the initial version of the algorithm, which is represented by squares, and the optimized version, which is represented by dots.

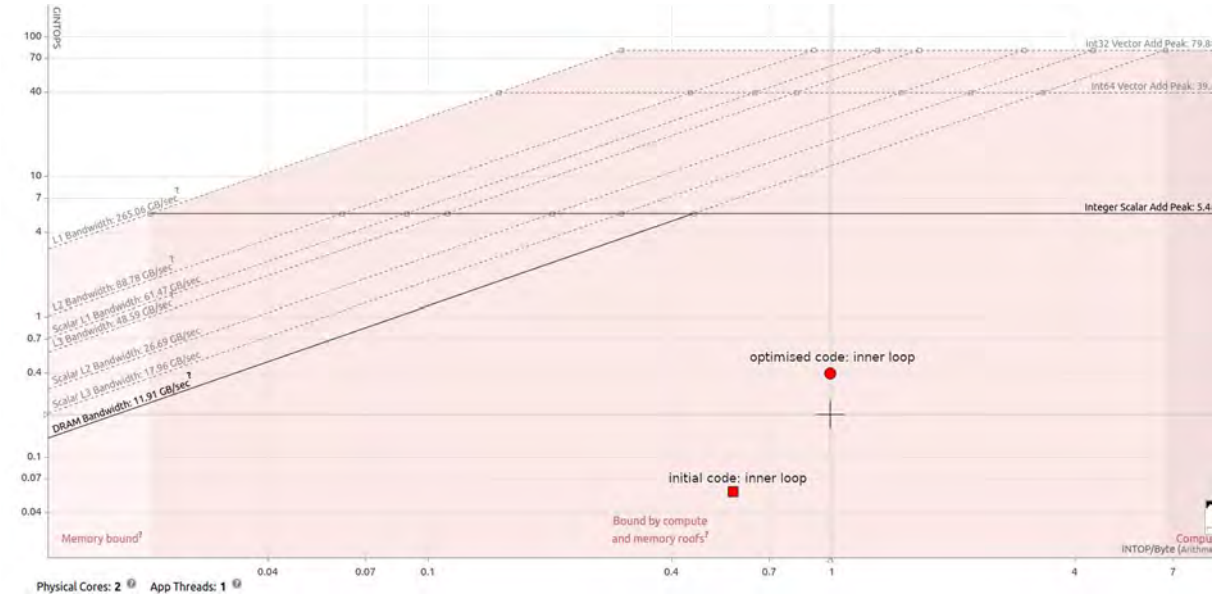


Figure 2.4: Roofline model for N=32, M=32K

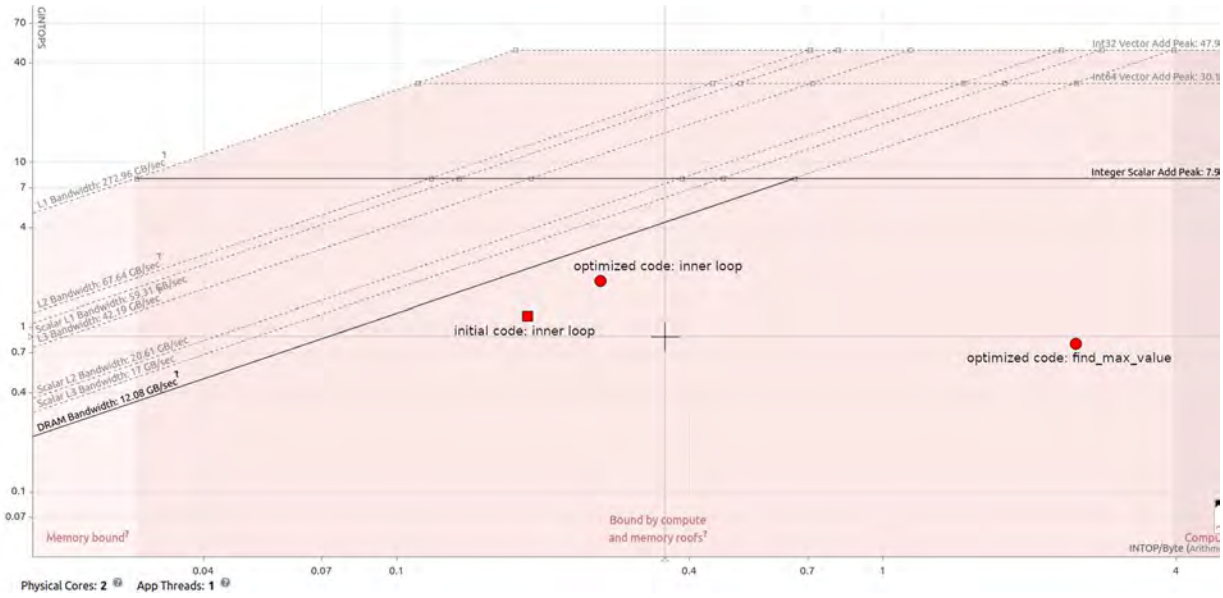


Figure 2.5: Roofline model for N=32, M=64K

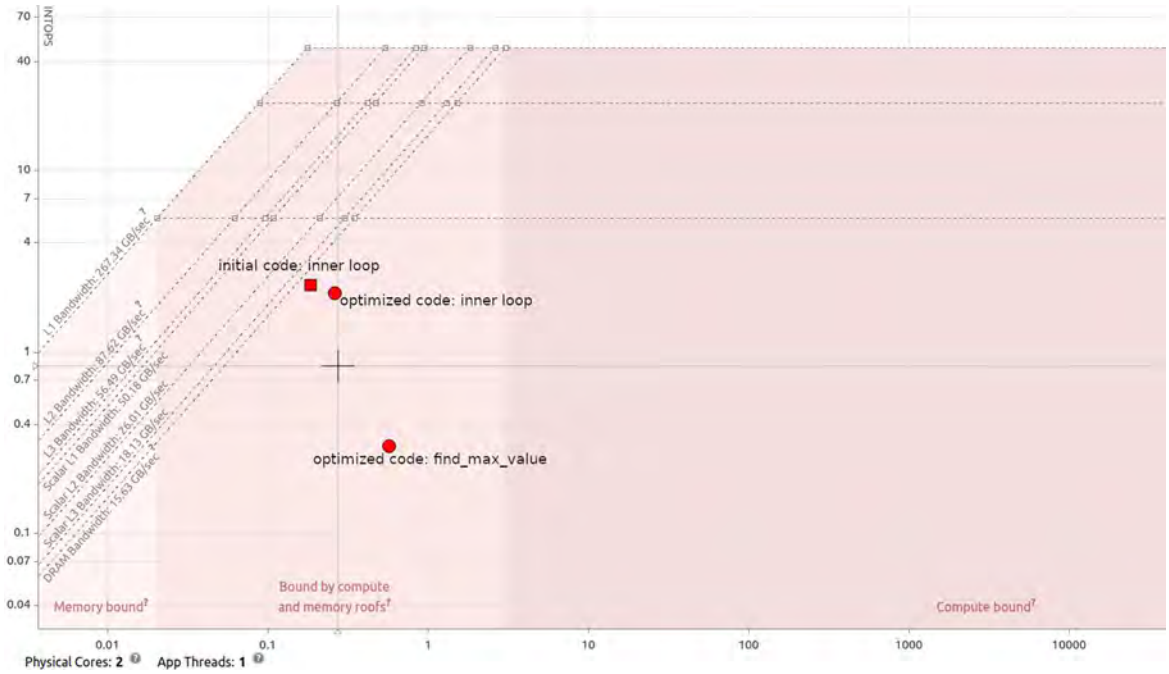


Figure 2.6: Roofline model for N=32, M=128K

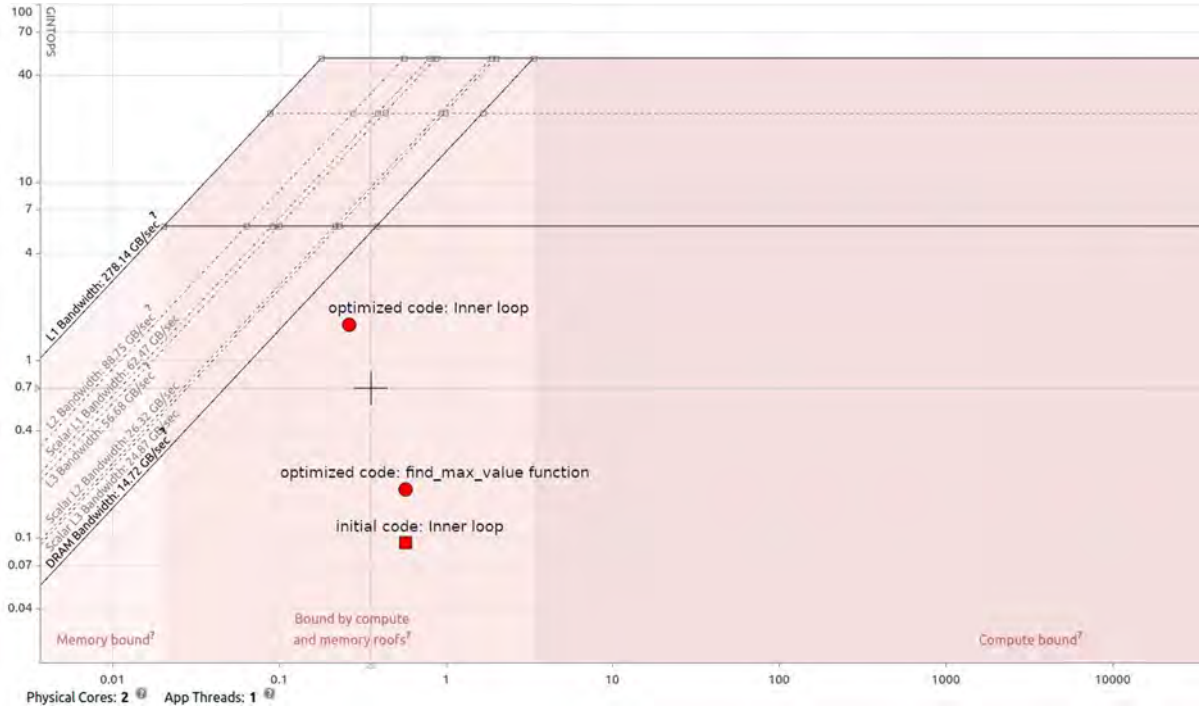


Figure 2.7: Roofline model for N=64, M=32K

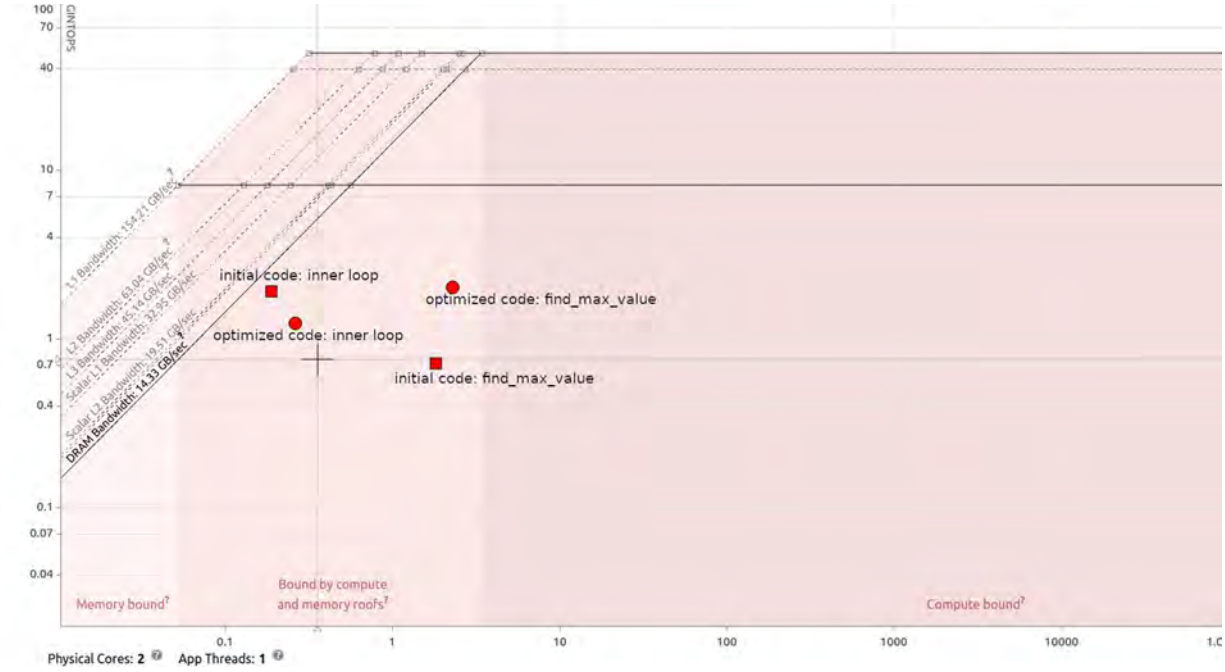


Figure 2.8: Roofline model for N=64, M=64K

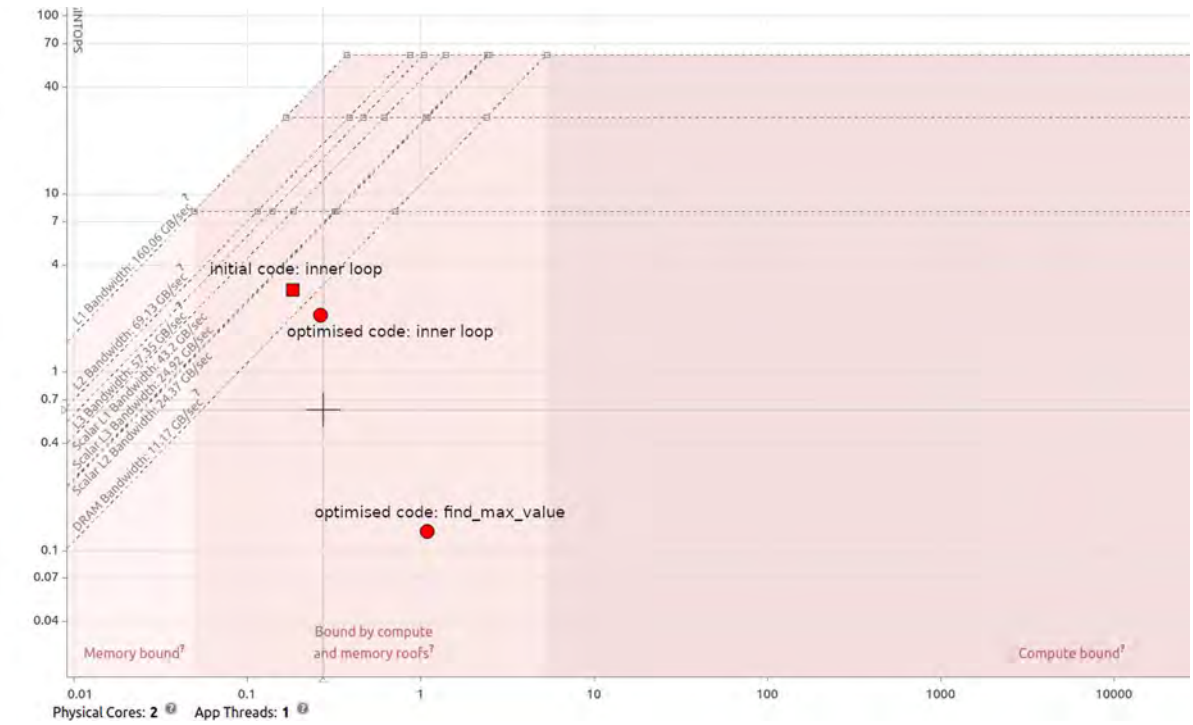


Figure 2.9: Roofline model for N=64, M=128K

In most of these cases, the performance dot has moved upwards and to the right, closer to the compute-bound area of the graph and closer to the AI*β line, meaning an increase in GINTOPS as well.

Furthermore, dots that represent the find_max_value function also appear in the roofline model. The function, which determines which neighboring cell is contributing to each value of the constructed matrices, appears, as expected, to be mostly compute-bound.

In conclusion, we can argue that we have successfully optimized our LSAL implementation to a certain extent on a software platform such as the x86, before attempting to reach a better performance with the FPGA implementation in the next chapter.

2.2 Execution on Arm

Alongside the programmable logic, the Zedboard provides an Arm Cortex-A9 dual core processing system with a frequency of 1.0GHz [7] . It is interesting to observe the performance our code can achieve on this processor as well. However, since the Zedboard is a low-power embedded platform, it does not provide rich analysis tools. Therefore, we will settle for a simple measurement of execution time of our code for a number of different pairs of N and M.

N	M	execution time (ms)
32	32	62,952
32	64	125,489
32	128	250,638
64	32	126,584
64	64	252,611
64	128	505,353
128	32	252,625
128	64	505,153
128	128	1013,249

Table 2.3: Execution times of the optimized code on the Arm CPU for several pairs of N, M

Figure 2.10: Comparison between execution on the x86 system and the Arm CPU

Figure 2.11: Ratio of Arm to x86 execution times

It is clear that the Arm processor is not particularly powerful compared to the x86 system. Specifically, the Arm processor is around 7.5 times slower than the x86 when we run examples of larger databases and queries, thus larger N*M matrices.

Chapter 3

3.1 Hardware Optimizations

Moving from a software implementation to a hardware design on the FPGA means we are expected to apply a different approach in any further development of this application. FPGAs provide great flexibility, similar to a software platform, due to the fact that their functionality can be customized by the designer every time a particular task needs to be implemented, according to the task's specific needs. This is achieved by having an array of logic blocks that can be reconfigured as specified by a hardware description language (HDL) such as Verilog or VHDL.

However, in order to provide a higher level of abstraction to the designer, high-level synthesis (HLS) tools are commonly used. The programmer writes code in a high-level language, such as C/C++, which we use in this thesis, while the HLS tool analyzes the code, and subsequently handles creating the register-transfer level (RTL) design in an HDL, as well as the synthesis to the gate level. The HLS tool used here is Vitis HLS by Xilinx.

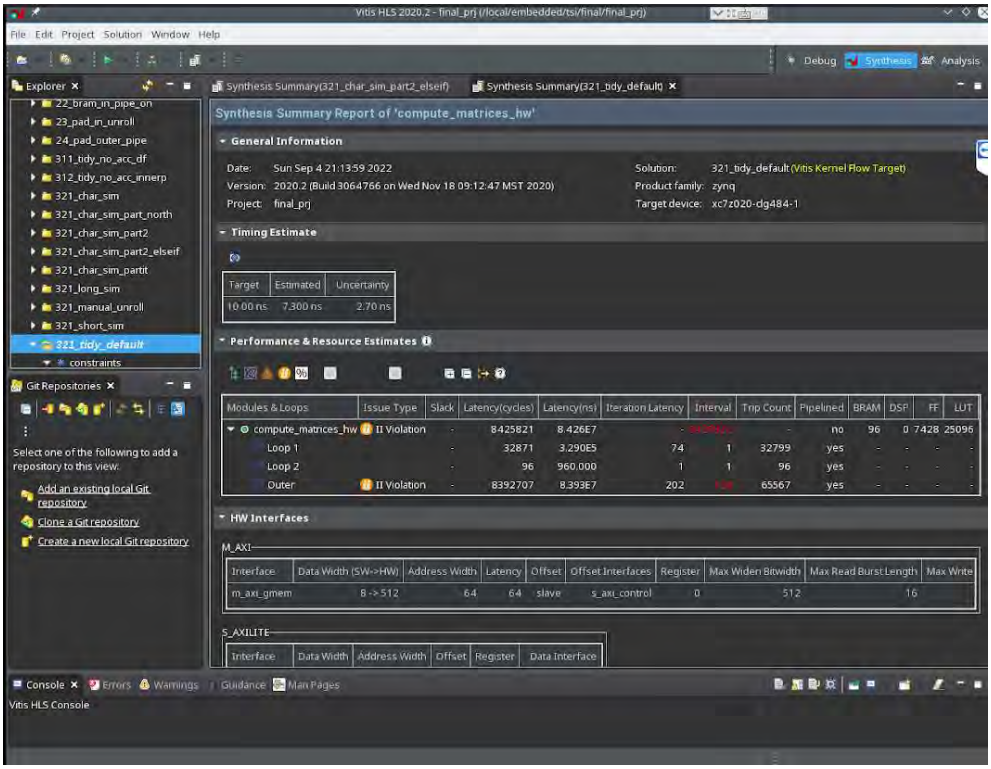


Figure 3.1: Vitis HLS Screenshot

FPGAs are widely used in applications with large sections of parallel computing and moreover provide the advantage of using the optimal amount of resources.

The FPGA used in this thesis is part of the Zedboard Zynq-7000 Development Board by Diligent. It provides 140 units of block RAM, 106400 flip-flops (registers) and 53200 Look Up Tables. [7]

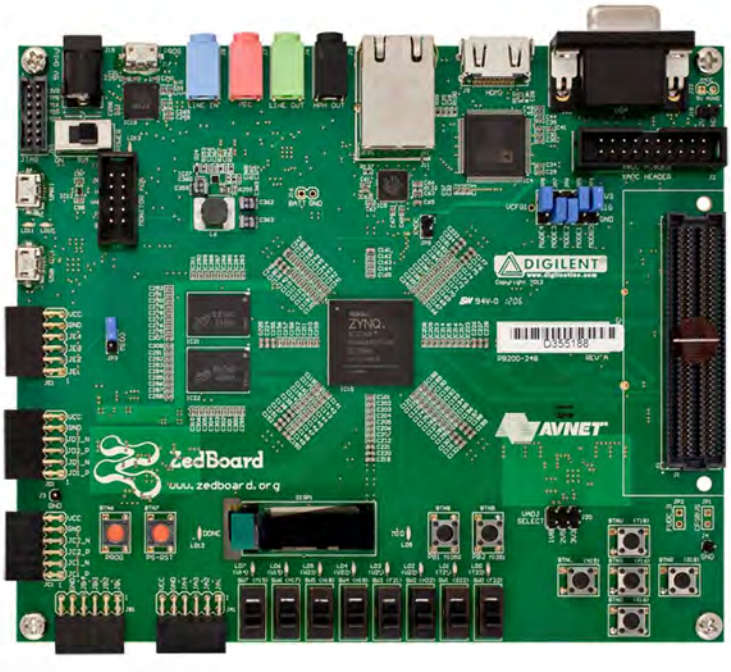


Figure 3.2: The Zedboard Zynq-7000 Development Board [8]

The code we developed in software will now be our kernel code, which will be mapped into hardware and accelerated in the FPGA. In addition to this we develop a testbench file which handles the initializations. the input/output to and from the kernel, as well as the verification of a correct hardware result.

It is expected that software and hardware optimizations differ vastly, for a number of reasons. For instance, the FPGA is not equipped with caches, contrary to a software platform, so it is not possible to preemptively fetch data we expect to reuse in blocks alongside the data we request. Fetching data should be done efficiently and in an inexpensive way. Additionally, the code might need to be re-written to expose patterns of parallelism that can be taken advantage of in a platform that can support a great degree of computational parallelism, such as the FPGA. In conclusion, software optimizations are not guaranteed to yield a satisfactory result when applied on hardware.

3.1.1 Baseline code & First directives

Before we begin applying hardware optimizations, it is important to observe how the initial software code performs on the FPGA. Initially we choose a query length of N=32 and a database length of M=65536 to showcase the results of each incrementally applied optimization.

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices		-	620167239	6.202 s	-	620167240	-
Outer loop		-	620167168	6.202 s	-	-	65536

Resources	BRAM	FF	LUT
Utilization	30	10795	17654
	(10%)	(10%)	(33%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.30ns	2.70ns

Table 3.1: Vitis HLS baseline code results for N=32, M=65536

As expected, the baseline code is not adequately efficient as a hardware accelerator.

HLS UNROLL

By unrolling a loop, multiple of its iterations are able to be executed together. Applying the directive creates multiple instances of the loop body and its instructions that can then be scheduled independently, allowing Vitis HLS to aim for more aggressive optimization and reduce the latency of each loop iteration. [9]

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices		-1.10	598016142	5.980 s	-	598016143	-
Outer		-	598016000	5.980 s	9125	-	65536

Resources	BRAM	FF	LUT
Utilization	30	104132	325638
	(10%)	(97%)	(612%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	8.396ns	2.70ns

Table 3.2: Vitis HLS inner loop unroll results for N=32, M=65536

This solution is discarded due to excessively high utilization of lookup tables that is beyond the Zedboard resources.

HLS PIPELINE

Loop pipeline allows iterations of a loop to happen concurrently instead of strictly sequentially. Each iteration can begin before the previous one has completed all operations. By specifying the initiation interval II, we set the target number of cycles after which the hardware will attempt to begin the next execution of a loop iteration. In Vitis HLS, the II defaults to 1. Additionally, all nested loops inside a loop or function in which pipelining is applied will be automatically unrolled. [10]

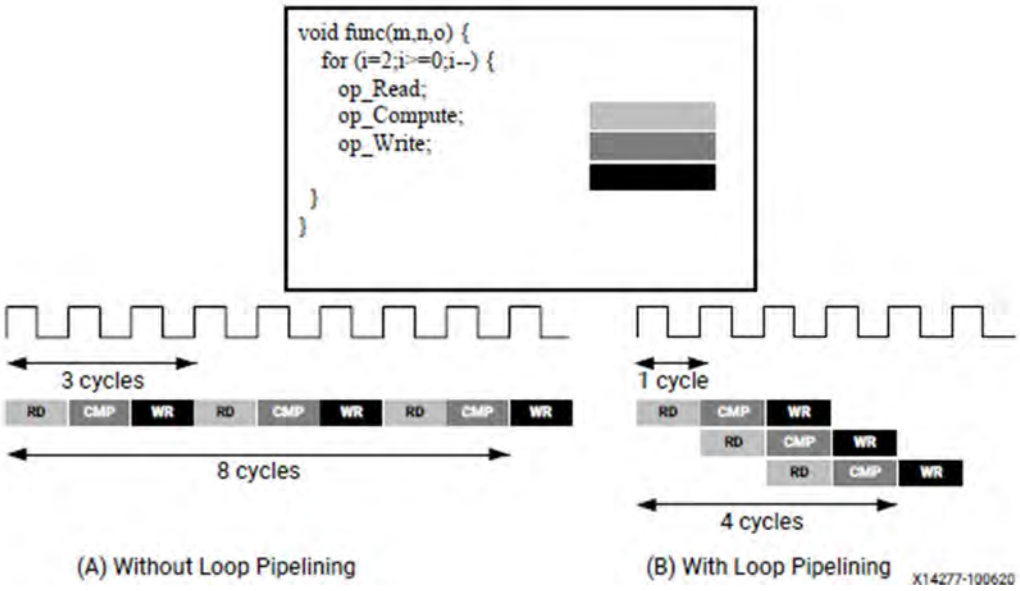


Figure 3.3: Pipelining of a simple 3-instruction loop. Pipelining directly results in overlapped execution of iterations, which reduces the overall execution time. [10]

First, we attempt to pipeline the inner loop.

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II Violation	-	320471111	3.205 s	-	320471112	-
Outer		-	320471040	3.205 s	4890	-	65536
Inner	II Violation	-	4674	4674 ns	149	146	32

Resources	BRAM	FF	LUT
Utilization	30	10770	17327
	(10%)	(10%)	(32%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.3ns	2.70ns

Table 3.3: Vitis HLS inner loop pipeline results for N=32, M=65536

As stated earlier, unrolling the inner loop results in a solution that cannot be implemented due to lack of resources. Therefore, it is unnecessary to test a solution pipelining the outer loop, which would unroll the inner loop automatically.

3.1.2 Rewriting the code to expose parallelism

The software implementation constructs the N*M similarity and direction matrices by scanning all cells sequentially. To calculate a cell [i,j] we need to access its north and northwest neighboring cells, which lie on the previous row i-1, and the west neighboring cell, belonging in the same row as cell [i,j]. Each time the west cell needs to be updated before its next one, which causes data dependency.

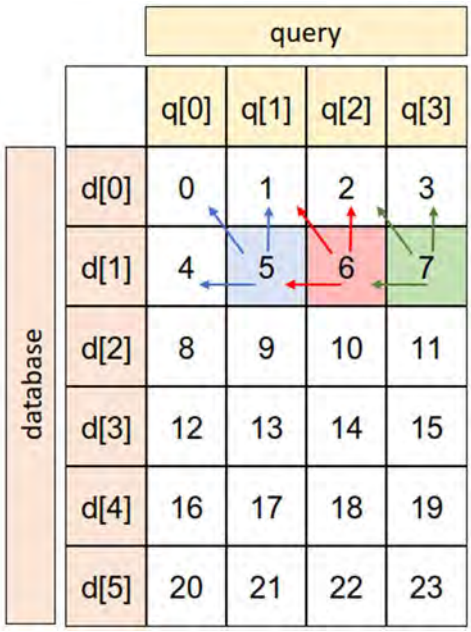


Figure 3.4: Data dependencies in a similarity matrix of N=4 and M=6, inside each cell its index is depicted. Arrows show the cells needed to calculate the cell of the same color.
All cells in a row cannot be updated at the same time since each cell requires the previous one to have already been updated.

To solve this problem, we will scan the similarity matrix (anti-)diagonally, calculating one antidiagonal before moving onto the next.

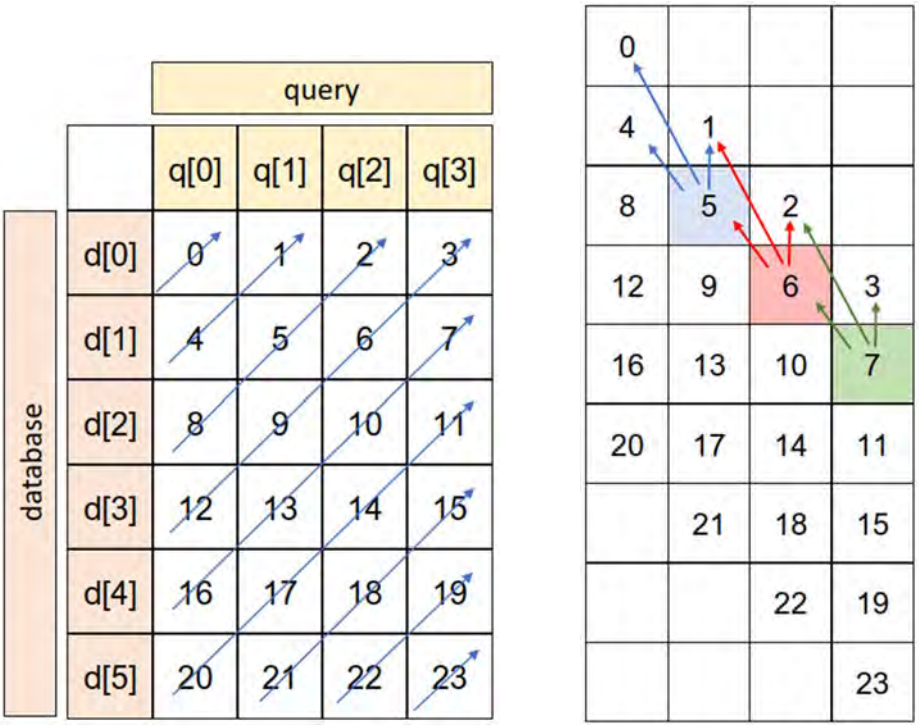


Figure 3.5: Left: scanning the similarity matrix in antidiagonals. Right: data dependencies in the produced similarity matrix where each row corresponds to an antidiagonal of the initial matrix. After restructuring the code, all elements in a row can be calculated concurrently, since all required cells belong in previous rows and have already been updated in previous iterations.

This rewriting of the code introduces a new problem that needs to be addressed: The first N-1 and last N-1 iterations of the outer loop are of varying length. This will not be optimal for a hardware implementation, due to the fact that variable bounds loops cannot be unrolled, therefore preventing successful pipelining of the loop.

We can solve that by padding the upper left and lower right triangles of the new matrix to ensure that each anti-diagonal has the same length of N. To achieve this, padding is applied accordingly to the database string2 array, by adding N-1 “invalid” cells in its beginning and end. We choose values that will produce a mismatch with the query.

Figure 3.6: Left: the padded database has a size of MM = M+2*(N-1). The algorithm begins at anti-diagonal N-1 and ends at anti-diagonal MM. Right: The M+N-1 antidiagonals as calculated and stored in memory by each iteration of the outer loop. Thanks to the padded areas, all antidiagonals have the same length N.

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II Violation	-	306329244	3.063 s	-	306329245	-
Outer_Inner	II Violation	-	306329100	3.063 s	150	146	2098144

Resources	BRAM	FF	LUT
Utilization	16	5673	12425
	(5%)	(5%)	(23%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.3ns	2.70ns

Table 3.4: Vitis HLS restructured code results for N=32, M=65536

3.1.3 Burst accesses to memory

Right now, all the required data resides in the main memory of the Zedboard. Every access of an array element in the code corresponds to fetching one element at the time from the main memory to an internal block ram. Each one of these transactions requires tens of cycles.

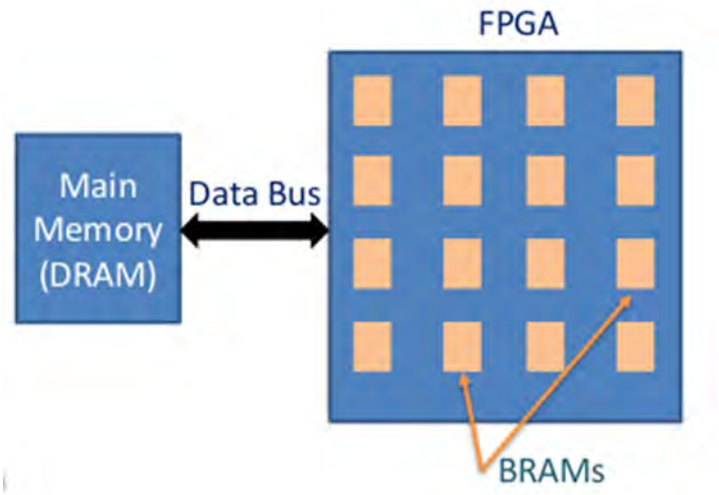


Figure 3.7: The FPGA provides a series of internal block RAMs, which allow data transferring in bursts, thus obscuring the memory access latency and improving bandwidth usage.

We can create local copies in the BRAMs and access them avoiding this large latency. In C/C++ code, this can be achieved by using the memcpy function, which is equivalent to pipelined accesses in a for-loop, in order to copy the kernel parameters into local arrays.

Now memory access will be done in bursts, which means it is possible to transfer multiple data in a single transaction, with only the first transfer being costly and all subsequent ones requiring one cycle each. [11]

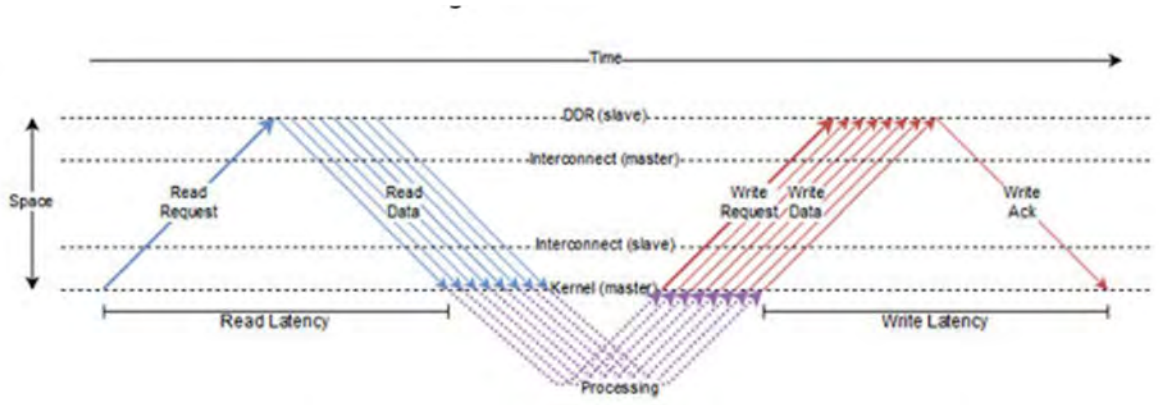


Figure 3.8: How the AXI protocol works. [11]

Regarding the output matrices, the naive approach would be to keep whole copies of them in Block RAMs and copy them all at once back to the main memory when all computations are finished. This will not fit into the available BRAMs for larger values of N,M.

Instead, since to compute one row of the similarity matrix only the two previous ones are needed, a BRAM array of 3*N will suffice. Similarly, the direction matrix can be replaced by a smaller array of size N.

```
int small_sim[N*3];
short small_dir[N];
…
//i: row index in sim_matrix
//di: row index in dir_matrix
//bram_index: row index in small_sim
Outer loop{
    Inner loop{ … }
    memcpy(similarity_matrix+(i*N), small_sim+(bram_index*N), N*sizeof(int));
    memcpy(direction_matrix+(di*N), small_dir, N*sizeof(short));
}
```

Figure 3.9: Using the memcpy function.
When an anti-diagonal has been completed, the resulting row of size N is copied back to the main memory array.

We can also retry applying the pipeline directive to the outer loop.

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II & Timing Violation	-0.38	26915583	0.269 s	-	26915584	-
Outer		-	26882470	0.269 s	410	-	65567
Inner	II & Timing Violation	-	202	2020 ns	79	4	32

Resources	BRAM	FF	LUT
Utilization	96	5356	9637
	(34%)	(5%)	(18%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.683ns	2.70ns

Table 3.5: Vitis HLS BRAMS & inner pipeline results for N=32, M=65536

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II & Timing Violation	-0.38	6327647	63.276 ms	-	6327648	-
Outer	II & Timing Violation	-	6294533	62.945 ms	172	96	65567

Resources	BRAM	FF	LUT
Utilization	96	11556	27120
	(34%)	(10%)	(50%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.683ns	2.70ns

Table 3.6: Vitis HLS BRAMS & outer pipeline results for N=32, M=65536

The resulting speed up is satisfactory, but now a timing violation is introduced to the design.

3.1.4 Inlining the find_max_value function by hand

In an attempt to reach a solution without a timing violation, we will try to replace the find_max_value function with code inside the kernel body.

This is called inlining and it allows the components within the function to be better shared or optimized more effectively along with the logic in the main kernel function. Function inlining is also performed automatically by Vitis HLS but we can test whether it is possible to achieve a better result by replacing the function with code inside the kernel.

```
small_sim[bram_index*N + ii] = west;
small dir[ii] = WEST;

if (north >= 0 && north >= test_val && north >= west){
    small_sim[bram_index*N + ii] = north;
    small_dir[ii] = NORTH;
}
if (test_val >= 0 && test_val >= north && test_val >= west){
    small_sim[bram_index*N + ii] = test_val;
    small_dir[ii] = NORTH_WEST;
}
if (0 >= test_val && 0 >= north && 0 >= west){
    small_sim[bram_index*N + ii] = 0;
    small_dir[ii] = CENTER;
}
```

Figure 3.10: the function code inside the main body of the kernel. We aim for the least amount of conditional statements as to obtain the most straightforward hardware implementation.

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II Violation	-	8425821	84.258 ms	-	8425822	-
Outer	II Violation	-	8392707	83.927 ms	202	128	65567

Resources	BRAM	FF	LUT
Utilization	96	7428	25096
	(34%)	(6%)	(47%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.3ns	2.70ns

Table 3.7: Vitis HLS inlined function results for N=32, M=65536

A little speedup is lost but the timing violation is successfully avoided.

3.1.5 Replacing 1 BRAM matrix with 3 separate ones

Using one 3*N BRAM array as a substitute to the entire similarity matrix adds a fair amount of algorithmic complexity to our code, since calculations are now introduced to define which rows should each outer loop iteration write to and read from.

```
Outer loop{
    if (bram_index == 0){
        prev = 2;
        prev_prev = 1;
    }
    else if (bram_index == 1){
        prev = 0;
        prev_prev = 2;
    }
    else if (bram_index == 2){
        prev = 1;
        prev_prev = 0;
    }
    …
    if (bram_index == 0 || bram_index == 1){
        bram_index++;
    }
    else if (bram_index == 2){
        bram_index = 0;
    }
}
```

Figure 3.11: Segment from the previous version of our code.
We define the index names: bram_index: the row we calculate in an outer loop iteration; prev: the previous row containing the north and west values; prev_prev: the row containing the northwest value. These values change in a cyclic way from 0 to 2.

This can be easily avoided by using instead 3 separate block RAMs defined as to correspond to the rows of the small_sim array.

```
memcpy(prev_prev_sim,           prev_sim, N*sizeof(int));
memcpy(prev_sim,                bram_sim, N*sizeof(int));
memcpy(similarity_matrix+(i*N), bram_sim, N*sizeof(int));
```

Figure 3.12: At the end of each outer loop iteration, we copy the elements from one array to the other in order to free the writing array bram_sim.

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II Violation	-	1082158	10.822 ms	-	1082159	-
Outer	II Violation	-	1049143	10.491 ms	88	16	65567

Resources	BRAM	FF	LUT
Utilization	94	9963	22385
	(33%)	(9%)	(42%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.3ns	2.70ns

Table 3.8: Vitis HLS three BRAM arrays results for N=32, M=65536

The initiation interval has dropped to 16. The solution is run for additional pairs of N, M lengths.

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II Violation	-	2132287	21.323 ms	-	2132288	-
Outer	II Violation	-	2099240	20.992 ms	105	32	65599

Resources	BRAM	FF	LUT
Utilization	94	15466	38836
	(33%)	(14%)	(73%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.3ns	2.70ns

Table 3.9: Vitis HLS three BRAM arrays results for N=64, M=65536

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II Violation	-	1067327	10.673 ms	-	1067328	-
Outer	II Violation	-	1050664	10.507 ms	105	32	32831

Resources	BRAM	FF	LUT
Utilization	62	15452	38690
	(22%)	(14%)	(72%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.3ns	2.70ns

Table 3.10: Vitis HLS three BRAM arrays results for N=64, M=32768

It becomes clear now that the iteration interval II drops to a value of N/2. We will study the analysis view tab of the Vitis HLS shortly, in order to better understand how to further decrease the II.

3.1.6 Using N elements from the database string in each iteration

The length MM = M+2*N-2 of the padded database string might not fit in our FPGA for large values of M.

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II Violation	-	4326190	43.262 ms	-	4326191	-
Outer	II Violation	-	4194871	41.949 ms	88	16	262175

Resources	BRAM	FF	LUT
Utilization	286	9989	22539
	(102%)	(9%)	(42%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.3ns	2.70ns

Table 3.11: Vitis HLS three BRAM arrays results for N=32, M=256K

Indeed, we get a 102% usage of BRAM for N=32, M=256K.

Fortunately, to calculate one row (or anti-diagonal) of N elements we do not need immediate access to the entire (padded) database of MM; only N elements are required each time.

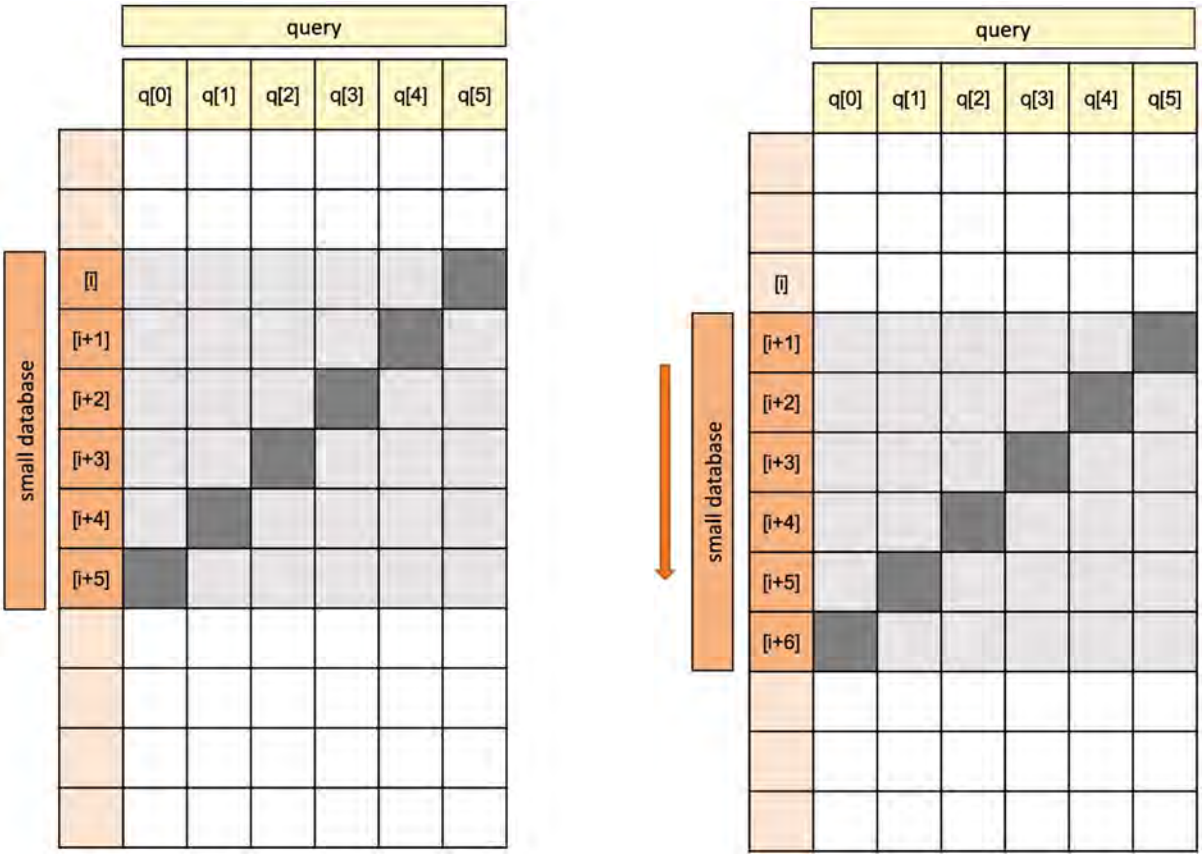


Figure 3.13: Substituting the database array string2[MM] with the array small_db[N]. After calculating one row in the outer loop, the database window small_db shifts one value downwards in the database.

```
Outer loop{
    Inner loop{
    . . .
    }
    memcpy(small_db, small_db+1, (N-1)*sizeof(char));
    small_db[N-1] = string2_main[i+1];
}
```

Figure 3.14: Fetching the next database value from the global memory to the block RAM.

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II Violation	-	12322493	0.123 s	-	12322494	-
Outer	II Violation	-	12322318	0.123 s	141	47	262175

Resources	BRAM	FF	LUT
Utilization	31	10546	23854
	(11%)	(9%)	(44%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.3ns	2.70ns

Table 3.12: Vitis HLS database window results for N=32, M=256K

The design with N=32, M=256K now fits on the Zedboard with an estimated BRAM utilization of 11%, since it only depends on the query length N.

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II Violation	-	3081917	30.819 ms	-	3081918	-
Outer	II Violation	-	3081742	30.817 ms	141	47	65567

Resources	BRAM	FF	LUT
Utilization	31	10524	23722
	(11%)	(9%)	(44%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.3ns	2.70ns

Table 3.13: Vitis HLS database window results for N=32, M=65536

Checking back on the results for N=32, we can see that the addition of the small database sliding window slows down our code by a factor of almost 3. To amend this, we can attempt partitioning the new array.

HLS ARRAY_PARTITION

The Zedboard block RAMs provide 2 access ports for writing and reading. This is restrictive since all the elements of an array cannot be accessed simultaneously, decreasing the

achievable level of parallelism and limiting the accelerator's performance. Using the ARRAY_PARTITIONING directive we can split a block RAM array into multiple smaller ones, or even into multiple registers, allowing scheduling that utilizes more or all array elements in one cycle. Here we apply complete array partitioning, meaning that we split the small_db array into registers. Larger register and LUT utilization is traded off for improved bandwidth and faster execution. [12]

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II Violation	-	1049340	10.493 ms	-	1049341	-
Outer	II Violation	-	1049196	10.492 ms	141	16	65567

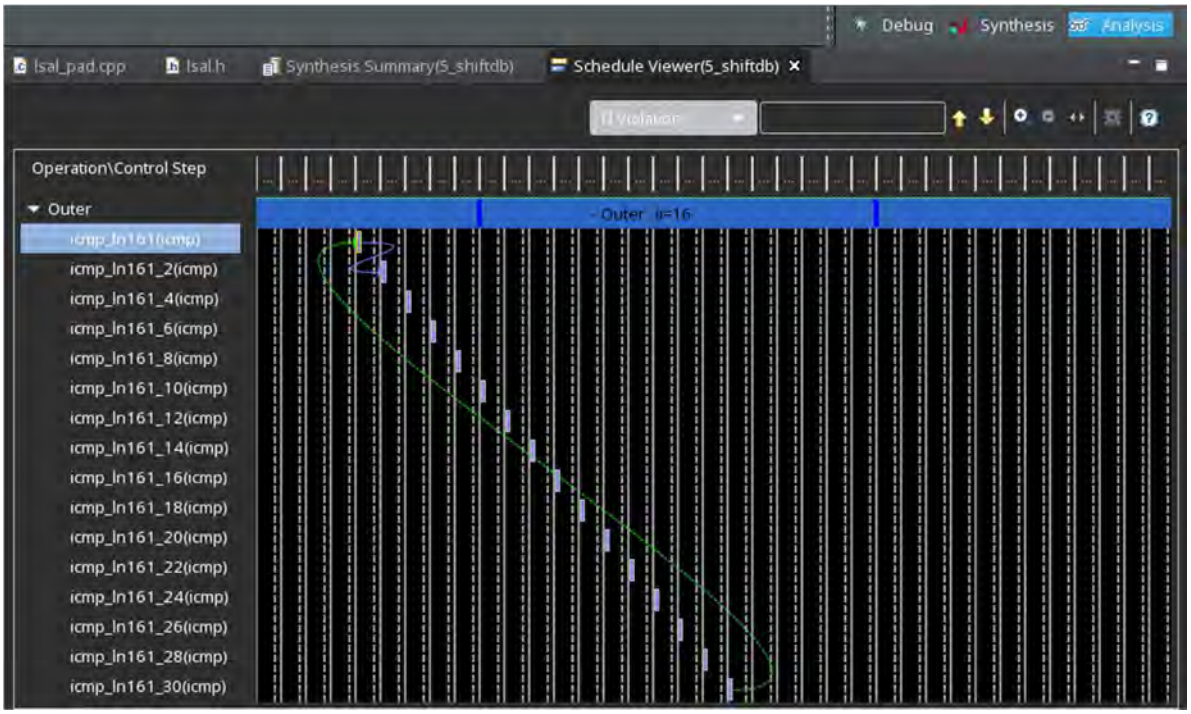
Resources	BRAM	FF	LUT
Utilization	30	10018	23324
	(10%)	(9%)	(43%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.3ns	2.70ns

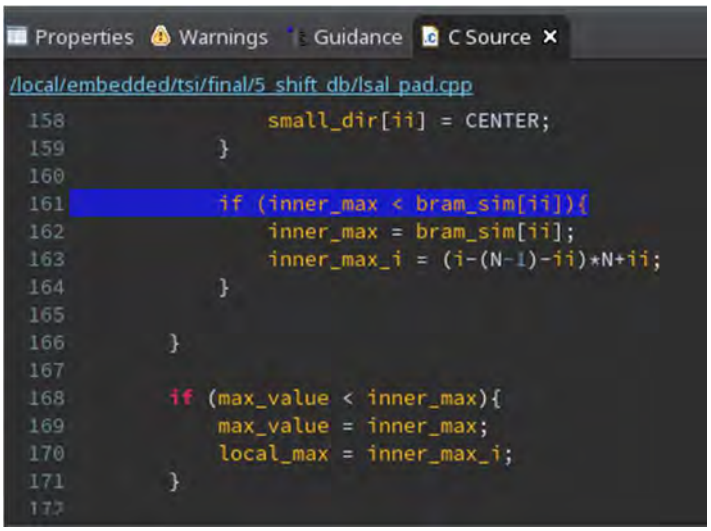
Table 3.14: Vitis HLS partitioning small_db results for N=32, M=65536

3.1.7 Unroll max index calculation

Looking at the analysis view tab it is possible to detect the II violation is caused by the if-statements that compute the max index. Remember from chapter 2, max index is the index of the maximum value in the similarity matrix. An inner loop iteration, while completing a row of both matrices, detects the max value of said row. The outer loop iteration will compare the max values of each row to detect the max value of the entire similarity matrix.



Figures 3.15, 3:16: Above: II Violation on the Vitis HLS Analysis view tab;
Below: Where in the code Vitis HLS detects the II Violation occurs



Similarly to the manual inlining of the function we applied earlier, it might be a good idea to help the HLS with some manual unrolling of the code we have found to be problematic.

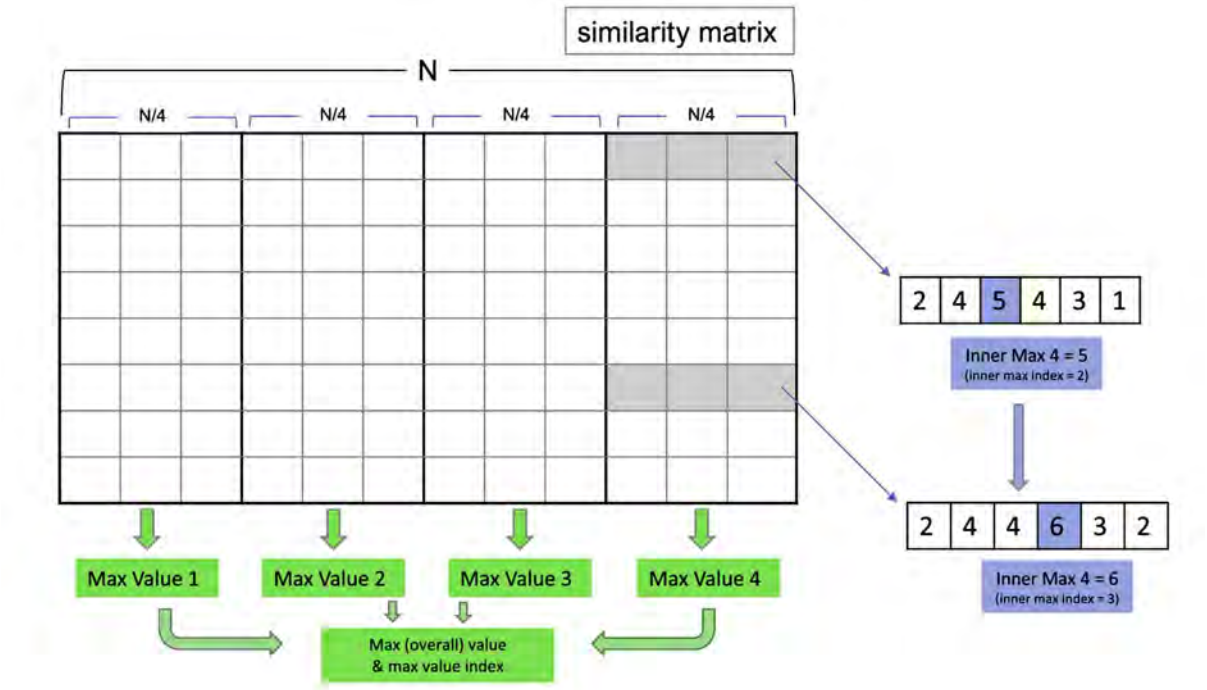


Figure 3.17: Separating the max index calculation into four parts.

We split each row into four parts and detect the max value of each part, each value stored in a different variable. Now the outer loop executes four sets of comparisons between these variables in-between iterations. A final comparison between the four winners is done at the end of the kernel function to obtain the maximum similarity value.

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II Violation	-	262550	2.626 ms	-	262551	-
Outer	II Violation	-	262405	2.624 ms	141	4	65567

Resources	BRAM	FF	LUT
Utilization	30	9541	24010
	(10%)	(8%)	(45%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.3ns	2.70ns

Table 3.15: Vitis HLS unrolled max_index results for N=32, M=65536

We have achieved an II of just 4 cycles now, or N/8 for N=32, four times less than before, just as we expected. We can also observe this on the Analysis tab.

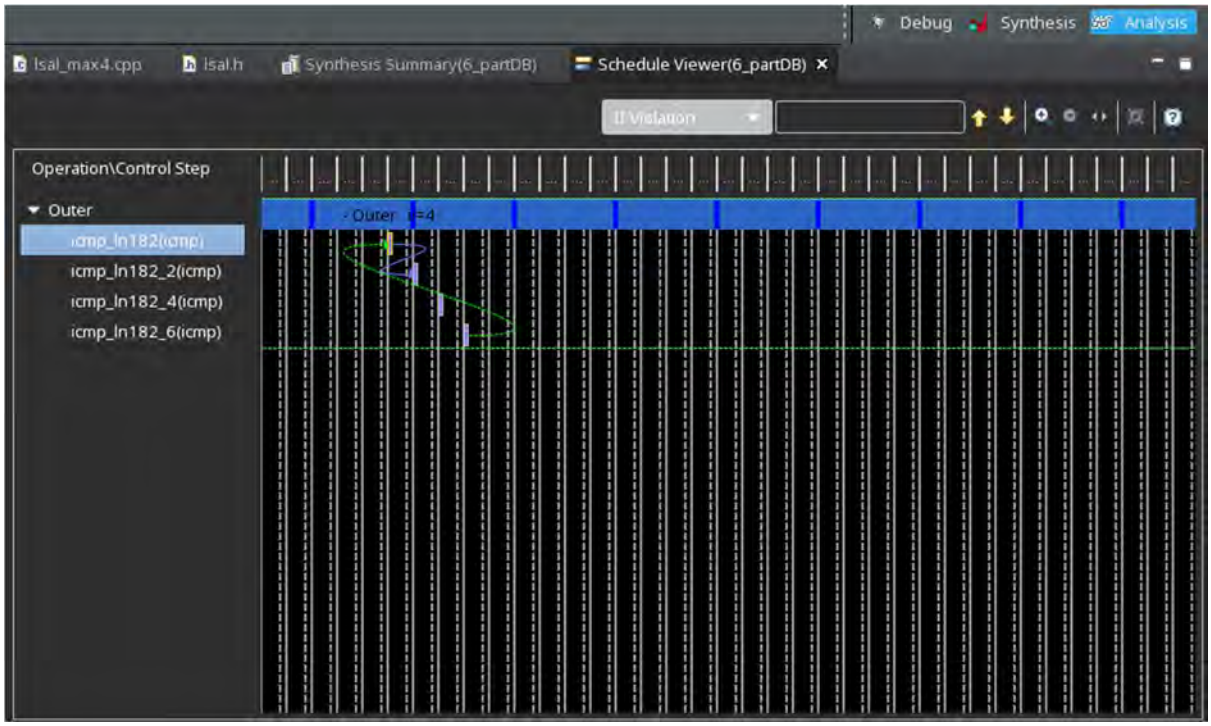


Figure 3.18: Vitis HLS Analysis Tab for unrolled max_index for N=32, M=65536

At this point, it is important to note that throughout the optimization experiments, we could not achieve an implementable solution for N=128, as this query length consistenly required a greater number of look up tables than the Zedboard can provide.

3.1.8 Experimenting with different array types

The software code uses an int array for the similarity matrix and a short array for the direction matrix. Unfortunately, the FPGA resources are limited, so it is wise to aim for an implementation that reduces the area of the design.

The values of the similarity matrix are directly related to the value of N. The maximum similarity value occurs in the case of every base of the query string1 matching with every base of the database string2.

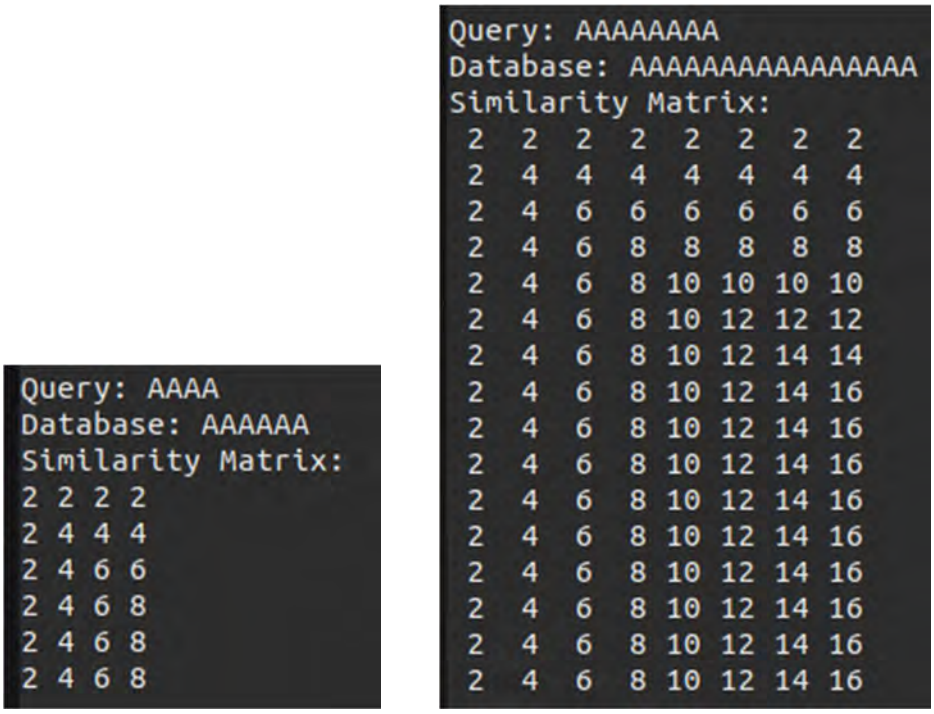


Figure 3.19: Software simulation of a total match between query and database, of lengths N=4, M=8 (left) and N=8, M=16 (right).

We observe that the largest similarity value is 2*N and appears in the last cell of each row, since in every row we begin with a score of zero to which we keep adding the match score, equal to 2.

For N=32, max similarity value <= 64.

For N=64, max similarity value <= 128.

For N=128, max similarity value <= 256.

With an unsigned char type, it is possible to store numbers up to 255, so given that the maximum value 256 is practically impossible, for a design with a query length N<=128, a similarity array of type unsigned char will suffice.

Meanwhile, direction matrix cells can only have one of four possible values (0,1,2,3 for C, N, NW, W accordingly), so we can change the type from short in the software implementation to char as well.

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II Violation	-	262549	2.626 ms	-	262550	-
Outer	II Violation	-	262404	2.624 ms	141	4	65567

Resources	BRAM	FF	LUT
Utilization	17	6185	17794
	(5%)	(5%)	(33%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.3ns	2.70ns

Table 3.16: Vitis HLS unsigned char results for N=32, M=65536

Although additional speedup is not gained, a decrease in resources utilization is achieved for all resource types.

3.1.9 Arbitrary precision types

Generally, the C/C++ data types might have a larger bitwidth than what will be needed for several variables in the HLS code, leading to synthesis of an RTL design that wastes FPGA resources. To combat this, Vitis HLS provides arbitrary precision types of a bit length that can be specified by the developer according to the design's needs. [13]

In the Smith-Waterman algorithm, the query and database strings are sequences of the four DNA bases: A, C, G, T. Therefore, they can be represented by just two bits instead of the 8 bits provided by the char type.

We can substitute the strings with arrays of arbitrary type ap_int<2>.

```
const ap_uint<2> CENTER          = ap_uint<2>("0b00", 2);
const ap_uint<2> NORTH           = ap_uint<2>("0b01", 2);
const ap_uint<2> NORTH_WEST      = ap_uint<2>("0b10", 2);
const ap_uint<2> WEST            = ap_uint<2>("0b11", 2);
```

Figure 3.20: substituting the 0,1,2,3 values with the unsigned 2-bit values 00,01,10,11

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency	Iteration Latency	Interval	Trip Count
compute_matrices	II Violation	-	262704	2.627 ms	-	262705	-
Outer	II Violation	-	262404	2.624 ms	141	4	65567

Resources	BRAM	FF	LUT
Utilization	17	6611	19901
	(6%)	(6%)	(37%)

Timing Estimation	Target	Estim.	Uncertainty
	10.00ns	7.3ns	2.70ns

Table 3.17: Vitis HLS arbitrary types results for N=32, M=65536

3.1.10 Conclusion

As demonstrated, the optimal solution is the one achieved in section 3.1.8, where the integer and short type matrices are substituted by the more area-efficient char type ones. This is the version that will be compiled for and run on the Zedboard.

The following graphs constitute an overall comparison between all execution times and usage of FPGA resources across all solutions.

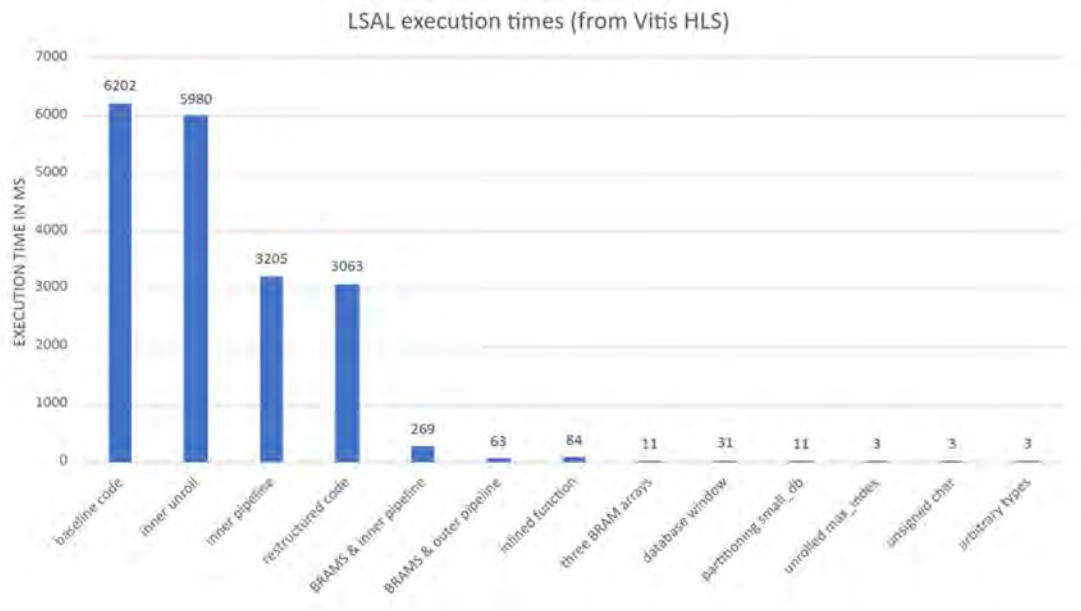


Figure 3.21: Execution time comparison of all solutions

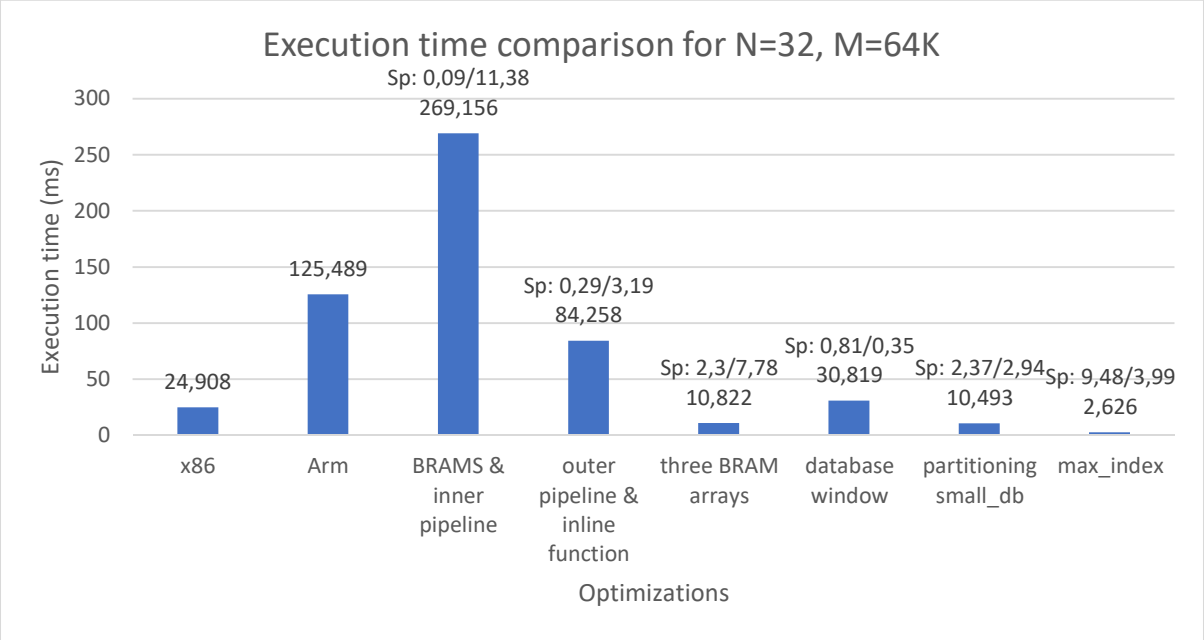


Figure 3.22: A better look at execution time of solutions from section 3.1.3 onwards

In Figure 3.22 we see a complete overview of the execution times since we brought the data into the block RAMs, where we achieve the first significant speedup. Above the execution time label, in each optimization we applied we see the speedups S1/S2, with S1 being the speedup with regard to the x86 time and S2 the speedup with regard to the previous optimization.

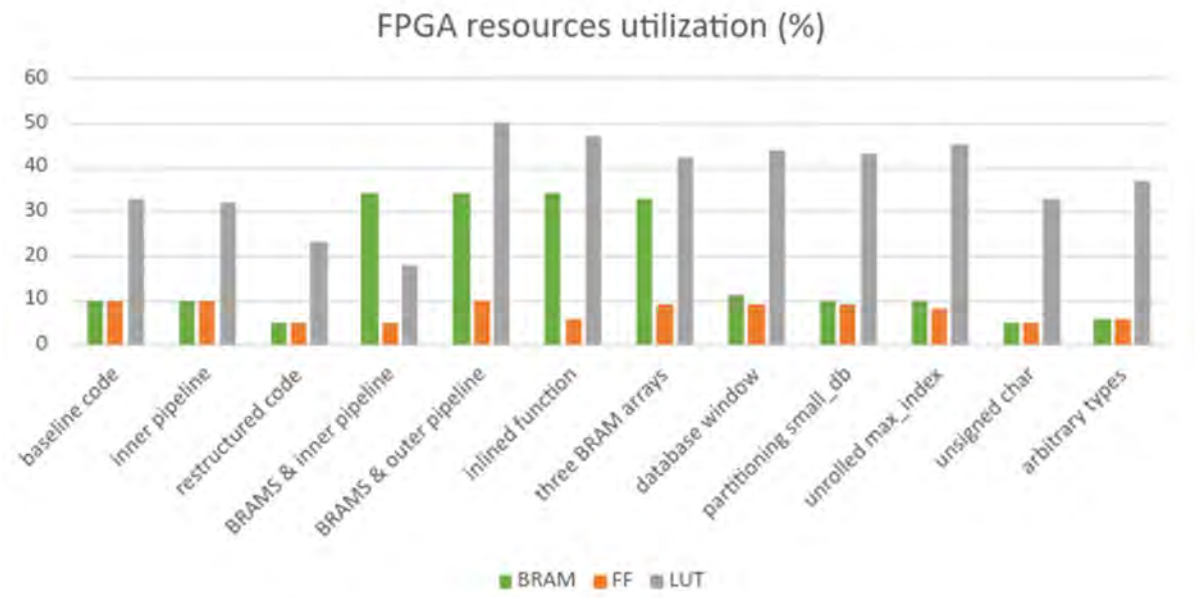


Figure 3.23: Block RAM, Flip-Flop and Look-Up Table percentage utilization across solutions.

3.2 Execution on the FPGA

The LSAL application we have developed is comprised of two files: the kernel code and the host application. The host program runs on the Arm CPU of the Zedboard and is similar to the testbench used during the hardware optimization process. Additionally, it incorporates API abstractions like OpenCL, which is a framework that manages interactions between the software host (Arm) and the hardware accelerator.

The kernel file we developed in section 3.1 is compiled with the Vitis compiler v++ into an RTL design. The output is a binary file which runs on the programmable logic region of the board.

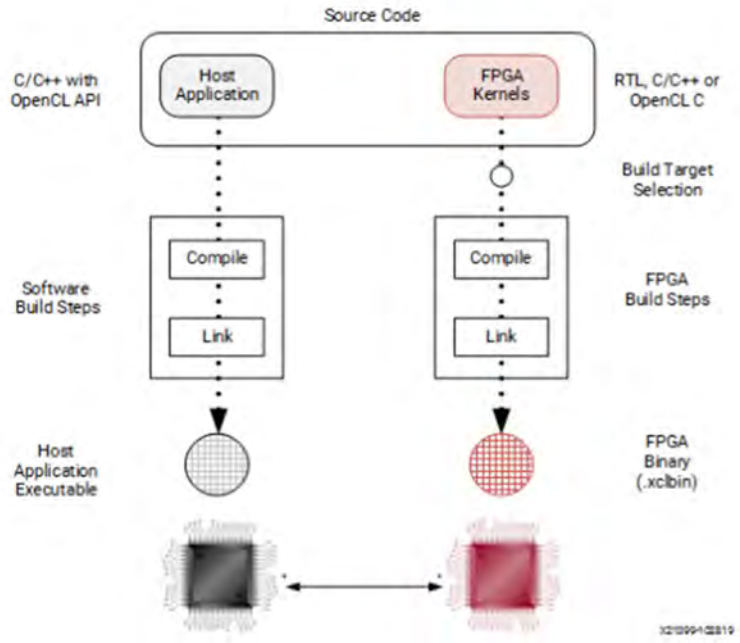


Figure 3.24: Build process for the Host and Kernel executable files

We test the performance of our application for the query and database lengths used throughout this chapter. For N=32, M=65536 the execution time is 6,322 ms, almost 3x slower than the 2,626 ms time predicted by Vitis HLS. This is considered an expected and normal divergence between the hardware simulation Vitis HLS performs and the execution on the actual hardware platform.

Figure 3.25: Comparison between LSAL execution times across all platforms, for N=32, M=65536

Figure 3.26: Comparison between LSAL execution times across all platforms, for N=64, M=65536

We have successfully managed to achieve a hardware implementation that is indeed faster than the optimized code which runs on the x86 processor. The overall speedup is x3,94 for N=32, M=65536 and x 2,87 for N=64, M=65536: the two pairs of sequence lengths we studied more specifically in this chapter.

Chapter 4

4.1 Conclusion

In conclusion, we were successful in developing a hardware implementation of the Smith-Waterman local sequence alignment algorithm for the Zedboard Zynq-7000 development kit, which achieved a faster execution time than the software implementation we developed, optimized and run on a x86 system. Rewriting the LSAL code to expose parallelism patterns, the use of the internal block RAM which significantly increase the memory bandwidth, as well as thoroughly understanding and making good use of the HLS directives that ensure a higher degree of parallel execution, were crucial to the success of this project. The final overall speedup, despite not achieving a change in order of magnitude, can nonetheless be a good starting point for further development and optimizing on the FPGA.

4.2 Future Work

In the world of optimizing hardware there is always room for further development and improvement of the already achieved performance. These are a few ways this thesis could potentially be further expanded:

First and foremost, using a larger FPGA which will provide more hardware resources would be a good place to start. Having more look-up tables available can allow for more flexibility with our code rewrites, and thus more room for experimenting with more optimizing techniques, since with more LUTs we can increase the computational complexity of the

application. This, along with a larger number of registers, means that we can also experiment more with partitioning the arrays we use in our code.

Additionally, we could look further into taking advantage of the full 512-bit length of the AXI bus between the global memory of the board and the internal block RAM of the FPGA. By using standard C/C++ data types for the kernel inputs and outputs, the bus is being underutilized since the native types are quite smaller than 512 bits long. To maximize the memory throughput by using the full width of the AXI bus, 512-bit array elements should be used for the kernel parameters. This is possible due to the arbitrary precision types the Vitis HLS provides, with which we became familiar in chapter 3.

Specifically, arrays of arbitrary type ap_uint<512> would be used for the similarity and the direction matrix. To avoid more memory transactions, we could also choose to completely discard the similarity matrix and not pass it back to the host, since the direction matrix and the max_index are a sufficient output in order to begin the back-tracing process.

Moreover, going back to eq. 1.1, we could expand the usage of the algorithm by developing solutions for larger gap lengths k, l, attempt to work around the data dependencies that will appear and study how to expose potential patterns of parallelism.

Finally, this thesis could be additionally expanded by constructing roofline models for the hardware implementations running on the FPGA, for a better insight into the bottlenecks of the application and how to handle them.

Bibliography

[1] "Field-programmable gate array - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Field-programmable_gate_array.

[2] "Bioinformatics - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Bioinformatics.

[3] T. F. Smith and M. S. Waterman, "Identification of Common Molecular Subsequences," *Journal of Molecular Biology,* 1981.

[4] "Smith–Waterman algorithm - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm.

[5] "Intel® Advisor Roofline," [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/guide/intel-advisor-roofline.html.

[6] "Intel® Advisor Roofline - Understanding a Roofline Chart," [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/guide/intel-advisor-roofline.html.

[7] Xilinx, "Zynq-7000 SoC Data Sheet: Overview (DS190)," [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview.

[8] Xilinx, "Zedboard," [Online]. Available: https://www.xilinx.com/products/boards-and-kits/1-8dyf-11.html.

[9] "Vitis High-Level Synthesis User Guide (UG1399) - pragma HLS unroll," [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-unroll.

[10] "Vitis High-Level Synthesis User Guide (UG1399) - pragma HLS pipeline," [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-pipeline.

[11] "Vitis High-Level Synthesis User Guide (UG1399) - Optimizing Burst Transfers," [Online]. Available: https://docs.xilinx.com/r/2020.2-English/ug1399-vitis-hls/Optimizing-Burst-Transfers.

[12] "Vitis High-Level Synthesis User Guide (UG1399) - pragma HLS array_partition," [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-array_partition.

[13] "Vitis High-Level Synthesis User Guide (UG1399) - Arbitrary Precision (AP) Data Types," [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Arbitrary-Precision-AP-Data-Types.