Hardware Accelerator of BWFA-MEM2 on Alveo U50 using HLS

A DISSERTATION SUBMITTED IN PARTIAL FULFILMENT
FOR THE DEGREE OF

Master of Technology

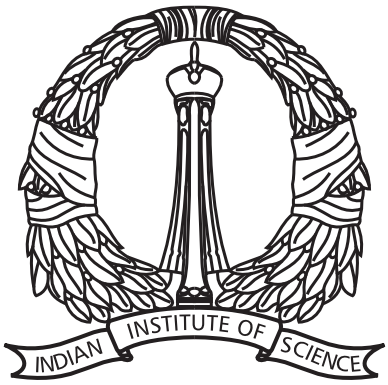
IN THE
ELECTRONIC SYSTEMS ENGINEERING

BY

SRI SAI NOMULA
SAI CHARAN KATAKAM

GUIDED BY

KURUVILLA VARGHESE



DEPARTMENT OF ELECTRONIC SYSTEMS ENGINEERING
INDIAN INSTITUTE OF SCIENCE, BANGALORE

MAY 2025

COPYRIGHT © 2025 IISC
ALL RIGHTS RESERVED

Synopsis

In bio-informatics, genome sequencing is a major field where determining the order of nucleotide bases Adenine, Thymine, Guanine and Cytosine (A, T, G, and C) is of high importance to study and analyze the genome under consideration. These nucleotide bases make up a DNA(Deoxyribonucleic acid) sequence. To identify regions of similarity between two such DNA sequences, these nucleotide bases must be arranged in manner which signifies structural or evolutionary modifications that the DNA sequences could have undergone over time. Such an arrangement is called pairwise sequence alignment and it is of high importance to identify regions of similarity and evolutionary differences between two DNA sequences. As the length of sequences to be aligned approached millions of nucleotide bases, the need to devise algorithms which reduced the time and space complexity of the computations involved in performing the alignment became of prime importance. Several possibilities of parallel and independent operations being performed in these algorithms made them suitable contenders to be accelerated by re-configurable hardware like Field Programmable Gate Arrays (FPGA). This report presents a high-performance, FPGA-accelerated DNA sequence alignment system implemented on the Xilinx Alveo U50 platform. Leveraging the parallel processing capabilities of FPGAs, the system addresses the growing computational demands of genomic data analysis, particularly for aligning sequences with lengths ranging from a few hundred to over a million base pairs.

Acknowledgements

We want to express our deepest gratitude to all those who have contributed to the completion of this thesis. Their support, guidance, and encouragement have been invaluable. We are deeply grateful to our guide **Prof. Kuruvilla Varghese** for providing us an opportunity to work under his invaluable guidance. His encouragement and support was of immense importance to our work. His expertise, insightful feedback, and unwavering support have been essential in shaping this work. We are also profoundly grateful to the project reviewers, **Prof. K.R Viveka** and **Prof. Utsav Banerjee**, for their critical insights and constructive suggestions during various stages of this project. Their probing questions and envisioned challenging scenarios significantly enhanced the depth and quality of this research.

Contents

Table of Contents **vii**

List of Figures **xi**

1 Pre-study **1**

1.1 Background 1

1.2 Why Bio-Informatics? 2

1.3 Introduction to Sequence Alignment 3

1.3.1 Scoring Alignments 4

1.3.2 Gap Penalties 5

1.3.3 Evolution of Sequences due to Mutations 7

1.4 Algorithms and Complexity 8

1.4.1 Big-O Notation 9

1.5 Overview of Sequence Alignment Methods 9

1.5.1 Dot-matrix Analysis 9

1.5.2 Dynamic Programming Algorithm 11

1.5.2.1 Formal Description of Dynamic Programming Algorithm : . 12

1.5.3 Developments to Optimize Time and Space 15

1.5.4 The Wavefront Alignment Algorithm 16

1.6 Product Survey 16

1.7 Market Survey 20

1.7.1 Parameters to be Measured, Monitored, and Controlled 20

1.7.1.1 Input Parameters 20

1.7.1.2 Monitored Parameters 20

1.7.1.3 Controlled Parameters 21

1.7.2 Target Market 21
1.8 User Survey 21
1.9 Wish Specification 22
1.9.1 Software Specifications 22
1.9.2 Hardware Specifications 22

2 Study **23**
2.1 Introduction 23
2.2 Representation of Indels 24
2.2.1 Gap Linear vs Gap Affine Algorithms 24
2.2.2 Global Sequence Alignment 25
2.3 Needleman-Wunsch Algorithm 26
2.3.1 Example of Needleman-Wunsch Algorithm 26
2.4 Smith-Waterman Algorithm 28
2.5 Gotoh Algorithm 28
2.5.1 Explanation of Gotoh Algorithm 29
2.5.2 Example of Gotoh Algorithm 29
2.6 Myers Algorithm 30
2.6.1 Edit Graph 31
2.6.2 Explanation of Myers Algorithm 31
2.7 Wavefront Alignment Algorithm 32
2.7.1 Explanation of WFA Algorithm 32
2.7.2 Example of Calculating Wavefront Vectors using WFA Algorithm . . . 33
2.8 Bi-directional Wavefront Alignment Algorithm 33
2.9 High Level Synthesis 34
2.9.1 Xilinx Vitis HLS 35
2.9.2 HLS Design Flow 36
2.9.3 C/C++ Constructs to RTL Mapping 38
2.9.4 Non Synthesizable Constructs of C/C++ 38
2.10 Installation and configuration of Alveo U50 39
2.10.1 Card Features 39
2.10.2 Card installation and validation 39
2.11 Target Specifications 40

2.11.1 Functionality 40
2.11.2 Hardware Platform 40
2.11.3 Input and Output Formats 40
2.12 Summary 42

3 Design **43**
3.1 Introduction 43
3.2 Longest Common Subsequence 43
3.3 The Manhattan Tourist Problem 44
3.3.1 Relation between Sequence Alignment and Manhattan Problem 45
3.3.2 Sequence Alignment as a graph 48
3.4 Space-Efficient Sequence Alignment 49
3.4.1 Middle Node of the Alignment 51
3.4.2 The Fast and Memory-Efficient Alignment Algorithm 52
3.5 Vitis HLS Implementation 54
3.5.1 Clock and Reset Specifications 55
3.5.2 Why Pragmas are used 55
3.5.3 Different Pragmas Available 56
3.6 Limitations of HLS Design 60

4 Engineering **61**
4.1 Introduction 61
4.2 What is Hardware Acceleration and Why it Matters 61
4.2.1 Why we need to enable it? 61
4.2.2 Why we may need to disable it? 62
4.2.3 CPU vs FPGA 62
4.3 Hardware-Software Partitioning 63
4.3.1 Profiling 63
4.3.2 Profiling our BWFA C++ Code 64
4.3.3 Analyzing the output of Profiler 66
4.4 Function Acceleration on FPGA with Vitis 67
4.4.1 Program Structure 67
4.4.2 Vitis Design Flow 68

4.5 Host Code Structure . . . 69
4.5.1 Setting up the Environment . . . 69
4.5.2 Core Commands . . . 70
4.5.3 Post-Processing . . . 71
4.5.4 Evolution of the Design: Enhancements and Optimizations . . . 72
4.6 PCIe Interface . . . 78
4.7 Kernel Code Structure . . . 80
4.7.1 Header Files . . . 80
4.7.2 Needleman-Wunsch Scoring Function . . . 81
4.7.3 BWFA Alignment Function . . . 82
4.8 Demo Setup . . . 83
4.8.1 System Architecture . . . 83
4.8.2 Data Flow Pipeline . . . 84
4.8.3 High Bandwidth Memory (HBM) Usage . . . 85
4.8.4 Host–FPGA Communication . . . 85
4.8.5 Performance Profiling . . . 85
4.8.6 Design Extensibility . . . 85

5 Results **87**
5.1 Simulations . . . 87
5.1.1 Simulation of designed BWFA C++ Code . . . 87
5.1.2 Hardware Simulation Through Command Line Interface (CLI) . . . 88
5.1.3 Hardware Simulation Through Graphical User Interface (GUI) . . . 88
5.2 Timing Report . . . 89
5.3 Resource Utilization . . . 89
5.4 Performance Analysis . . . 91
5.4.1 Length of input sequences *Vs* Speedup . . . 91
5.4.2 CPU Only performance *Vs* CPU + FPGA performance . . . 91
5.5 Correctness of Aligned Sequences . . . 92
5.6 Conclusion and Summary . . . 93
5.7 Future Scope . . . 94

List of Figures

1.1 Alignment of two sequences . . . 3
1.2 Global Alignment of two sequences . . . 4
1.3 Local Alignment of two sequences . . . 4
1.4 Example of Alignment using Constant Gap Penalty . . . 6
1.5 Example of Alignment using Linear Gap Penalty . . . 6
1.6 Example of Alignment using Affine Gap Penalty . . . 7
1.7 Changes in sequence over the time . . . 7
1.8 Example of Dot Matrix Method . . . 10
1.9 Example of Sliding Window Technique . . . 11
1.10 Dot Plot to visually compare the sequences of moufflon and takin . . . 11
1.11 Formal description of the dynamic programming algorithm. . . . 13
1.12 Traceback Matrix and Corresponding Alignments for different paths . . . 13

2.1 Insertions and Deletions for Sequence Alignment. . . . 24
2.2 Different Alignments between TREE and REED . . . 25
2.3 Example Alignment Matrix . . . 25
2.4 Alignment of Sequences using Gotoh Algorithm . . . 30
2.5 Edit Graph corresponding to the Alignment of strings “ACBBA” and “ABCA” . 31
2.6 Alignment of two sequences using WFA. . . . 33
2.7 Computed cells in the BWFA and their junctions. . . . 34
2.8 HLS Component Development Flow . . . 37
2.9 Alveo U50 Card Features . . . 39

3.1 Alignment of Sequences . . . 43
3.2 Map of Midtown Manhattan . . . 44
3.3 Description of Manhattan Tourist Problem . . . 45

3.4 An alignment of “ATGTTATA” and “ATCGTCC”. 45
3.5 Directed Acyclic Graph (DAG) 46
3.6 Alignment Graph with edges 46
3.7 Calculation of Maximum Length Path to go from Source to Sink. 47
3.8 Pseudo Code for Manhattan Tourist Problem with Down and Right Edges. . . . 48
3.9 Computing an alignment score using only two columns 50
3.10 Finding Middle Node based on two scores i.e., *FromSource(i)* and *ToSink(i)* . . 51
3.11 Divide and Conquer Approach to find the Optimal Alignment 53
3.12 Finding middle nodes within previously identified blue rectangles. 53
3.13 Pseudo Code for Linear Space Alignment 54

4.1 Representation of Hardware Acceleration 62
4.2 Terminal Commands used for profiling the C++ Code 65
4.3 Output of Profiling Tool for BWFA C++ Code 65
4.4 Analyzing the part of profiler output 66
4.5 General Form of program structure 68
4.6 Overview of Vitis Design Flow 69
4.7 Version-1 72
4.8 Version-4 74
4.9 Version-5 75
4.10 Version-6 76
4.11 Version-9 78
4.12 PCI Express topology [29]. 79
4.13 PCIe protocol stack 80
4.14 Demo Setup 83

5.1 C++ Code Simulation 87
5.2 Hardware Simulation through CLI 88
5.3 Hardware Simulation through GUI. 88
5.4 Timing summary 89
5.5 Resources Per Kernel. 90
5.6 Post Synthesis Utilization. 90
5.7 Length of input sequences *Vs* Speedup 91

5.8 CPU Only performance *Vs* CPU + FPGA performance 92
5.9 Results from the Implemented Design 93
5.10 Results from National Center for Biotechnology Information Website 93

List of Tables

1.1 Differences Between EMBL and GenBank Formats 18
1.2 Softwares used for Sequence Alignment and their features 19

2.1 Mapping C/C++ constructs to HW Components 38
2.2 Description of operators in CIGAR format. 41

5.1 Target vs. Estimated Frequency of Kernels 89
5.2 Resource Utilization Summary for Each Kernel Instance 90

Chapter 1

Pre-study

1.1 Background

Sequence Alignment is a fundamental technique in bio-informatics used to compare biological sequences (DNA, RNA, or proteins) to identify regions of similarity or difference. The wavefront Algorithm is an efficient approach for performing exact pairwise sequence alignment, since it is significantly faster than traditional Dynamic Programming as it focuses only on potentially optimal paths, especially for sequences with some degree of similarity. In addition, it guarantees the identification of the alignment with the highest overall score based on the scoring matrix. Biologists use alignment methods as a tool for finding common patterns between sequences and thus predicting the ancestral relationship, protein structure and function, identifying regions for drug design, etc. The number of sequences in the genome database is increasing exponentially each year. Approximately there are three billion pairs of nucleotides in the human genome alone. Given the large amount of data, there is always room for improvement in sequence alignment methods.

The Bidirectional Wavefront Algorithm (BWFA) algorithm is a very good approach for sequence alignment because the wavefront algorithm requires the storing multiple "wave-fronts" scores during the calculation, which can be memory-intensive for very long sequences. This can be addressed by MEM2 memory access pattern, which works by optimizing memory access patterns, thus can potentially accelerate the wavefront algorithm, especially for large sequences where memory access becomes a bottleneck. Therefore, the BWFA-MEM2 algorithm can be used for sequence alignment.

The software implementation of the BWFA-MEM2 algorithm is likely a performance bottleneck, hindering our application's overall speed. Hardware acceleration on an Alveo U50 data center FPGA card can address this by offloading the computationally intensive parts. CPUs often struggle with real-time processing due to limitations in handling high-throughput data streams. FPGAs, such as the Alveo U50, excel in low-latency and high-throughput scenarios, making them ideal for real-time applications.

Beyond performance gains, hardware acceleration offers additional benefits. FPGAs are generally more power efficient than CPUs for specific tasks, resulting in lower power consumption for our application. Alveo U50 is a Data Center FPGA Card from Xilinx which is a pro-

grammable chip that can be configured to implement custom hardware accelerators with the support of High level Synthesis (HLS).

HLS is a methodology for designing the specified hardware using a C-like programming language. HLS tools translate the C/C++ code into register transfer level (RTL) code that can be implemented on an FPGA. HLS allows the developer to design at a much higher level, thus accelerating product development, reducing costs, reducing errors, and unlocking high-performance hardware for innovative applications.

1.2 Why Bio-Informatics?

All life on Earth relies on three key types of molecules: DNA, RNA, and proteins. DNA holds the vast information that directs the functions of a cell, essentially serving as a blueprint. RNA plays a role in transferring segments of this information to various parts of the cell, where these segments are used as templates for protein synthesis. Proteins, in turn, function as enzymes that catalyze biochemical reactions, transmit signals between cells, form structural components like keratin in the skin, and carry out many of the cell's essential tasks.DNA, RNA, and proteins are all types of sequences constructed from specific alphabets: nucleic acids use a four-letter alphabet (for DNA and RNA), while proteins are composed from a twenty-letter amino acid alphabet [13].

By the early 20th century, it was recognized that DNA (deoxyribonucleic acid) is a long molecule composed of four types of bases: Adenine (A), Thymine (T), Guanine (G), and Cytosine (C). Initially, biologists identified five bases, including Uracil (U), which is chemically similar to Thymine. DNA consists of a sugar molecule, a phosphate group containing phosphorus, and one of the four nitrogenous bases (A, T, G, or C). The chemical bonds connecting the nucleotides in DNA are uniform, resulting in a consistent backbone structure [13]. The variation in DNA molecules comes from the different combinations of A, T, G, and C bases, which provide the unique characteristics of each DNA strand.

In the mid-1950s, Paul Zamecnik discovered that protein synthesis in the cytoplasm occurs with the help of large molecules called ribosomes, which contain RNA [13]. This finding led to the hypothesis that RNA could act as an intermediary between DNA and proteins. DNA serves as a template for transcribing a specific gene into messenger RNA (mRNA), which then transports the genetic code to the ribosome to direct protein synthesis. Chemically, RNA, or ribonucleic acid, is very similar to DNA, with two key differences: RNA uses Uracil (U) instead of Thymine (T), and the sugar in RNA contains an extra oxygen atom. These small differences have significant biological consequences. DNA is largely inert and typically double-stranded, which makes it suitable for storing genetic information. In contrast, RNA is more chemically active and generally single-stranded, allowing it to carry genetic messages from DNA to the protein-making machinery and participate actively in vital chemical reactions.

Bioinformatics emerged when biologists learned how to sequence DNA, leading to the creation of extensive datasets written in the four-letter DNA alphabet. Bioinformaticians apply algorithms, statistical methods, and mathematical techniques to interpret and analyze this DNA language. For instance, consider the genomic sequences of two insects that may share an evo-

lutionary relationship, like the fruit fly (Drosophila melanogaster) and the malaria mosquito (Anopheles gambiae) [13]. We might want to identify which regions of the fruit fly genome are similar or different when compared to the mosquito genome. Alignment algorithms help in comparing these two genetic sequences to highlight similarities and differences.

Recent advancements in next-generation sequencing technologies have driven a surge in genomic applications that support personalized medicine. However, these applications demand substantial computational resources because of the vast volumes of genomic data involved. A crucial initial step in many such applications is the alignment of sequencing reads to a reference genome.

1.3 Introduction to Sequence Alignment

DNA sequences consist of four characters, namely A - Adenine, G - Guanine, T - Thymine, and C - Cytosine. However, protein or amino acid sequences are represented by 20 different characters. To identify regions of similarity between two such DNA sequences, these nucleotide bases must be arranged in a manner that signifies structural or evolutionary modifications that the DNA sequences could have undergone overtime; this arrangement is called **pairwise sequence alignment**. Sequence alignment involves comparing two or more sequences to identify regions where individual characters or patterns occur in the same order. When only two sequences are aligned, it is referred to as pairwise alignment and aligning more than two sequences is known as multiple sequence alignment [1]. Typically, sequences are written in two rows, one above the other. Matching or similar characters are aligned in the same columns, while differences are represented either as mismatches or as gaps in one of the sequences. The aligned sequences have deletions, insertions and matches/mismatches as shown in Figure 1.1.

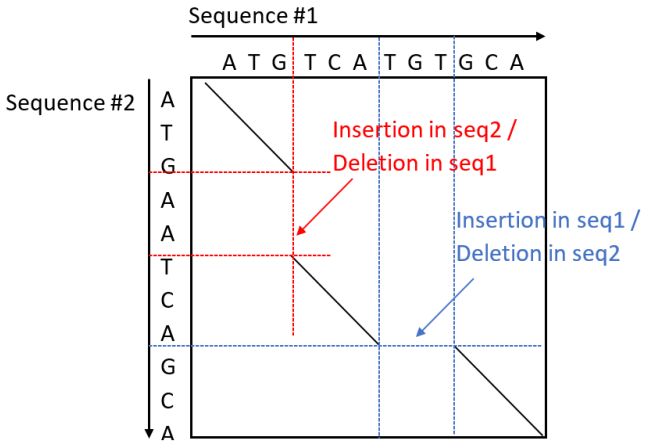


Figure 1.1: Alignment of two sequences

Sequence alignment is a powerful tool for uncovering functional, structural, and evolutionary relationships among biological sequences. Achieving the most accurate, or optimal, alignment is essential for extracting meaningful insights. When sequences from different organisms show similarity, it often suggests they originated from a shared ancestral sequence, making them homologous. The alignment helps trace the evolutionary changes that may have occurred between

the homologous sequences and their common ancestor [1]. Alignments can be categorized into two main types: global alignment, which compares sequences along their entire lengths, and local alignment, which focuses on finding the most similar regions within the sequences.

- **Global Alignment:** In global alignment, the entire sequence, including the ends of the sequence, must be aligned as shown in Figure 1.2. The goal is to match as many characters as possible across the entire length. This is suitable for aligning two closely related sequences or homologous genes, like comparing two genes with same function (in humans vs. mice) or comparing two proteins with similar function. A general global alignment technique is the Needleman-Wunsch Algorithm.

Figure 1.2: Global Alignment of two sequences

- **Local Alignment:** In local alignment shown in Figure 1.3, parts of sequences with highest number of matches or parts having high density of matches are aligned, thus forming clusters of alignment pairs, instead of extending the alignment to the whole length. This is suitable for aligning more divergent sequences or distantly related sequences. A general local alignment technique is the Smith-Waterman Algorithm.

Figure 1.3: Local Alignment of two sequences

1.3.1 Scoring Alignments

Scoring alignments is a fundamental part of sequence alignment algorithms. The alignment score quantifies the similarity between two biological sequences (DNA, RNA, or protein) and helps to determine the best possible alignment under a given scoring system. The purpose of scoring is to evaluate how well two sequences match and differentiate biologically meaningful alignments from random ones. Two corresponding metrics are commonly used to evaluate the alignment of two sequences, the similarity score, which quantifies how alike the sequences are, and the distance score, which measures the extent of their differences. The similarity score

quantifies how similar two sequences are by adding positive values for matches and penalizing mismatches and gaps. This type of scoring sequence similarity is one of the most familiar methods for biologists.
Similarity Score = (No.of Matches * Match Score) + (No.of Mismatches * Mismatch Score) + Gap Score. The other scoring method is the distance score, which measures how different the two sequences are. The goal is to minimize this score, interpreting alignment as a problem of editing distance.
Distance Score = (No.of Mismatches * Mismatch Cost) + Gap Cost. This scoring guide dynamic programming or heuristic algorithms toward optimal results. The common components of an alignment score are:

1. **Match Score:** Assigned when two identical residues (nucleotides or amino acids) align.
2. **Mismatch Penalty:** Negative score when two different residues align.
3. **Gap Penalty:** Penalizes insertions/deletions (indels). Can be constant, linear, affine, or logarithmic.

Substitution matrices are fundamental tools that are used to score matches and mismatches between residues (nucleotides or amino acids) during sequence alignment. They provide biologically informed values to indicate how likely one residue is to be replaced by another over evolutionary time. The most commonly used scoring matrices for protein sequence alignments are the PAM250 (Point Accepted Mutation) matrix, which is based on evolutionary models thus, best for closely related sequences and the BLOSUM62 (Blocks Substitution Matrix) which is empirically derived thus, best for diverse sequences. However, a number of other options are also available. These matrices provide scores for all possible pairwise substitutions, helping to capture biological relevance of amino acid similarities behind alignments. Using the appropriate matrix can increase the sensitivity and reduce false positives in alignment.

1.3.2 Gap Penalties

In biological sequence alignment, gaps represent insertions or deletions (indels) that have occurred over the course of evolution. These are introduced into sequences to maximize alignment similarity between two DNA, RNA, or protein sequences. A gap is a space inserted into a sequence to account for, Insertions in one sequence relative to another, or Deletions that may have occurred over time. In natural evolutionary processes, insertions and deletions occur less frequently than substitutions. Therefore, computational models should impose higher penalties for introducing gaps, mirroring the evolutionary rarity of insertion and deletion events. Gaps are biologically meaningful, but allowing unlimited gaps would make alignments trivially perfect. To avoid this, gap penalties are applied to control when and how gaps are introduced.

Gaps are penalized via various Gap Penalty Scoring methods i.e., cost to introduce gap. Choosing penalty values for insertions and deletions can be somewhat arbitrary, as there is no definitive evolutionary framework to specify their exact costs. If the penalties are too low, alignments may include excessive gaps, causing unrelated sequences to appear overly similar. Conversely, if the penalties are too high, gaps become rare, resulting in unrealistic alignments that overlook

meaningful insertions or deletions. Empirical research on globular proteins has produced a set of penalty values that work well for most alignment tasks and are typically used as default settings in many alignment tools. To model the biological cost of gaps more accurately, several types of gap penalties have been proposed. The most commonly used are

- **Constant Gap Penalty:** This is the simplest type of Gap Penalty. As shown in Figure 1.4, we assign the same negative score for each gap position regardless of whether it is opening or extending. This Penalty scheme has been found to be less realistic.

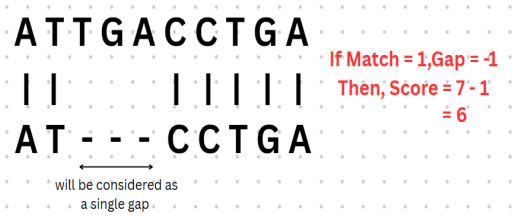


Figure 1.4: Example of Alignment using Constant Gap Penalty

- **Linear Gap Penalty:** The Linear Gap Penalty takes into account the length (L) of each insertion or deletion in the gap, where each unit length of a gap is penalized equally as shown in Figure 1.5. It assumes that opening and extending a gap have the same biological cost. Therefore, if the penalty for each inserted or deleted element is B and the length of the gap L, then the total gap penalty would be the product of two, that is, B x L. It does not distinguish between starting a new gap and extending an existing one, which is biologically unrealistic.

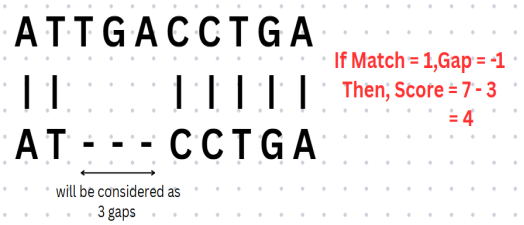


Figure 1.5: Example of Alignment using Linear Gap Penalty

- **Affine Gap Penalty:** The most widely used gap penalty function is the Affine Gap Penalty. The affine model distinguishes between the cost of opening a new gap and extending an existing one as shown in Figure 1.6. This reflects biological observations that opening a gap is often more penalizing than continuing it. The overall gap penalty W increases linearly with the length of the gap and is computed using the formula: $W = \gamma + \delta(k-1)$ where; γ = Gap Opening Penalty, δ = Gap Extension Penalty and k = length of the Gap.

- **Convex or Logarithmic Gap Penalty:** This type models the gap penalty as a non-linear function that grows with gap length but at a decreasing rate, often based on empirical data. The total gap penalty is a calculated using the formula, $W = \alpha * \log(k+1)$ where; α is a scaling factor that adjusts the severity of the penalty, k = length of the gap.

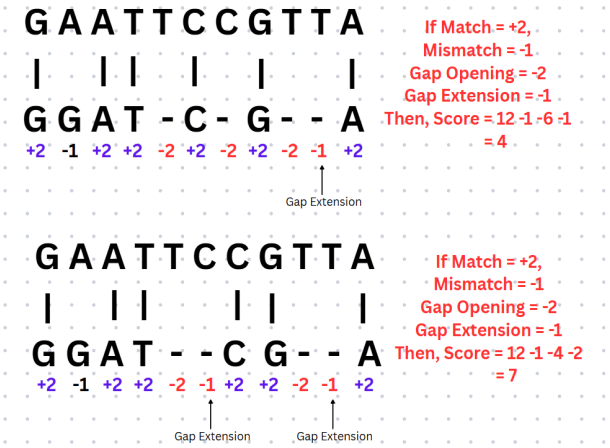


Figure 1.6: Example of Alignment using Affine Gap Penalty

1.3.3 Evolution of Sequences due to Mutations

Mutations are the fundamental driver behind the evolution of biological sequences, such as DNA and RNA. They cause variations in these sequences that can then be inherited by offspring through reproduction. Figure 1.7 shows how it works.

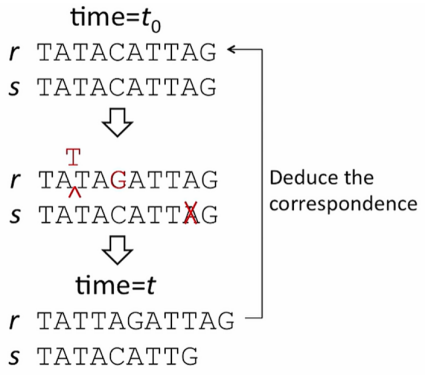


Figure 1.7: Changes in sequence over the time

Performing the alignment makes it easy to compute the similarity between two sequences, and we can observe the correspondence between them. Optimal alignments help biologists understand the relationships between sequences by identifying the most accurate arrangement of characters—indicating which characters should align and which represent insertions or deletions. This alignment insight is essential for drawing conclusions about the functional, structural, and evolutionary significance of the sequences [1]. Pairwise sequence alignment can be adapted to different protocols and applications (e.g., DNA-Seq, RNA-Seq) by using different distance functions (e.g., Gap-Linear or Gap-Affine), modifications in the alignment scope (e.g., Global or Local), or by adjusting the Scoring Penalties.

1.4 Algorithms and Complexity

An algorithm consists of a precise sequence of instructions designed to solve a particular problem without any ambiguity in the steps involved. Problems are usually described by their inputs and expected outputs, and the algorithm provides the procedure to transform these inputs into the correct outputs. To find a solution, a systematic process is needed to execute the steps defined by the algorithm. There exist many kinds of algorithms, each employing distinct strategies to tackle problems. The selection of an appropriate algorithm depends on the problem's nature, whether it involves searching, sorting, optimization, or decision-making, among other tasks [13].

1. **Exhaustive Search Algorithms:** An exhaustive search, or brute force algorithm, is a straightforward approach to solving a problem by systematically trying all possible options and selecting the correct one based on the problem criteria. It is simple but often inefficient for large datasets. In biological sequence alignment, a brute force algorithm would try all possible alignments of two sequences with gaps and score each alignment based on match/mismatch/gap penalties, then the alignment with the highest score would be chosen. This is impractical for long sequences due to the exponential number of alignments.

2. **Divide and Conquer Algorithms:** Divide and Conquer is a powerful algorithm design paradigm used to solve complex problems by breaking them into smaller and more manageable subproblems. These subproblems are solved independently and their solutions are combined to get the final answer. This is recursive in nature and works best when sub-problems are similar and independent. Divide-and-conquer algorithms for sequence alignment allow us to tackle larger alignment problems more efficiently by breaking them down into smaller, more manageable pieces. By using methods such as Hirschberg's algorithm, k-mer-based approaches, or parallel strategies, we can significantly reduce the computational load, especially when dealing with large datasets or real-time applications.

3. **Greedy Algorithms:** Greedy algorithms are a class of algorithms that make locally optimal choices at each step in the hope that these local choices lead to a globally optimal solution. The general principle of a greedy algorithm is to choose the best available option at each step based on some criterion (e.g., maximum profit, minimum cost) without considering the broader consequences of that choice. Although greedy algorithms are often faster and easier to implement, they do not always guarantee an optimal solution because they do not explore all possible choices. In sequence alignment, a greedy approach typically involves aligning portions of sequences in a step-by-step manner, usually by selecting the best possible local alignment at each stage. Greedy algorithms do not guarantee a globally optimal solution because they only make local decisions.

4. **Dynamic Programming Algorithms:** Dynamic Programming (DP) is a method to solve complex problems by breaking them down into simpler sub-problems and solving each sub-problem just once, storing its solution in a table (memoization or tabulation) to avoid redundant work. It is widely used for optimization problems, where the goal is to find the best solution among a large number of possibilities. DP is used in sequence align-

ment algorithms because it efficiently handles the combinatorial complexity of aligning sequences while ensuring that the best possible alignment is found.

1.4.1 Big-O Notation

Real computers take a specific amount of time to carry out operations like addition, subtraction, or checking conditions in a loop. One way to estimate the algorithm's running time is to multiply the number of operations by the time required for each operation. However, as computing technology continuously improves, the time per operation decreases, making this method of estimation obsolete. Instead of measuring the running time on each individual machine, we describe an algorithm's performance based on the total number of operations it executes. This approach focuses on the algorithm itself rather than the hardware it's running on [13].

Computer scientists use Big-O notation to succinctly express the running time of an algorithm. When we say an algorithm has a quadratic time complexity, or $\mathcal{O}(n^2)$, it means that the algorithm's running time is bounded by a quadratic function of n for an input of size n. Writing $f(\mathrm{n}) = \mathcal{O}(n^2)$ means that the growth of the function $f(\mathrm{n})$ does not exceed that of a quadratic function with a leading term of cn^2, for some constant c. However, it doesn't tell us if $f(\mathrm{n})$ grows slower than a quadratic function. Big-O notation simply establishes an upper bound on the growth rate of a function [13].

A related concept applies to lower bounds, where we use the notation $f(\mathrm{n}) = \Omega(g(\mathrm{n}))$ to indicate that $f(\mathrm{n})$ grows at least as fast as $g(\mathrm{n})$. If an algorithm's running time is both not faster and not slower than some function $g(\mathrm{n})$, then $g(\mathrm{n})$ is considered a tight bound for the algorithm's time complexity.

1.5 Overview of Sequence Alignment Methods

Given a set of sequences, an alignment is the same set of sequences with zero or more gaps inserted into them so that:

1. They all have the same length later on.
2. For each column, at least one of the resulting sequences is not a gap.

Sequence alignment techniques are generally divided into two main categories; pairwise and multiple sequence alignment. These can be further classified based on their strategy—global, local, or heuristic. In the case of pairwise sequence alignment, common methods include dot matrix analysis, dynamic programming algorithms, and word-based (k-tuple) approaches. Tools like FASTA and BLAST utilize these word-based strategies for efficient alignment.

1.5.1 Dot-matrix Analysis

The dot matrix method is one of the earliest and simplest techniques for comparing protein or nucleotide sequences [1]. It is especially effective in detecting insertions or deletions, which

may be less apparent using other methods. When sequences are not known to be highly similar, this approach is often used first, as it graphically represents potential alignments as diagonals within a matrix. The dot matrix offers a visual means of identifying similar regions between sequences. A key advantage of this method is that it displays all possible residue matches, allowing researchers to focus on those that appear most meaningful. Continuous diagonals indicate regions of similarity, while isolated dots typically suggest random matches with little biological significance. To enhance interpretation, the number of matches along each diagonal can be counted and compared with scores from randomized sequences, helping to distinguish statistically significant alignments.

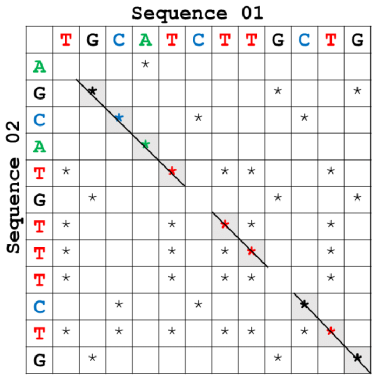


Figure 1.8: Example of Dot Matrix Method

A primary drawback of the dot matrix method is that most software implementations do not provide an explicit alignment of the sequences. Instead, they offer a visual representation or estimation of the degree of similarity between the two sequences. The dot matrix method is simple to visualize. The two sequences are represented as starting from the same point in two perpendicular axes. A dot is marked when there is a match between two base pairs; otherwise, the matrix is left empty. Long diagonally oriented dots would mean that there is a region in the two sequences that is similar. The example of this is shown in Figure 1.8.

Sliding Window Technique: The accuracy of identifying matching regions in a dot matrix can be enhanced by minimizing random matches through filtering. This is commonly done using a sliding window approach, where instead of comparing individual positions, a small segment of adjacent residues from each sequence is examined simultaneously. A dot is plotted only if the number of matches within this window meets or exceeds a predefined threshold. Since DNA sequences have only four nucleotide symbols, which increases the likelihood of random matches, they typically require larger window sizes around 15 bases with a match threshold of about 10. In contrast, protein sequences, which use 20 amino acid symbols and are less prone to random similarity, may not always require filtering [1]. However, when used, smaller window sizes of 2 or 3 residues with a match requirement of 2 can effectively highlight regions of similarity.

Dot plots for long sequences can be noisy, so to reduce the noise, i.e., dots all over the place in the plot, we use a window a threshold. Then we compare character by character within a window and require certain fraction of matches within the window in order to display it with a dot. If we choose window size as 11 and stringency as 7, it means that a dot is printed at a matrix position only if 7 of the next 11 positions in the sequences are identical. An example of

a sliding window technique is shown in Figure 1.9.

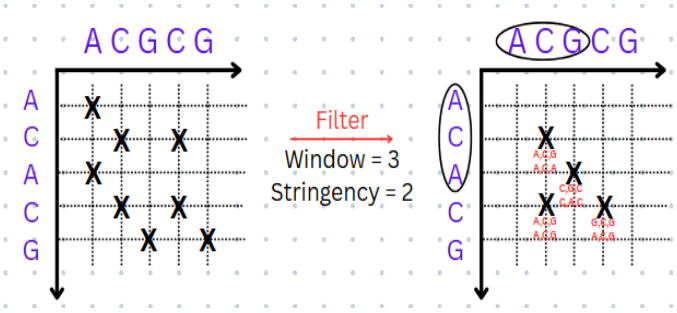
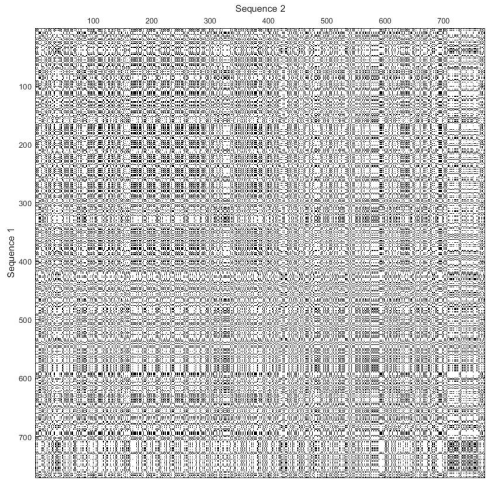
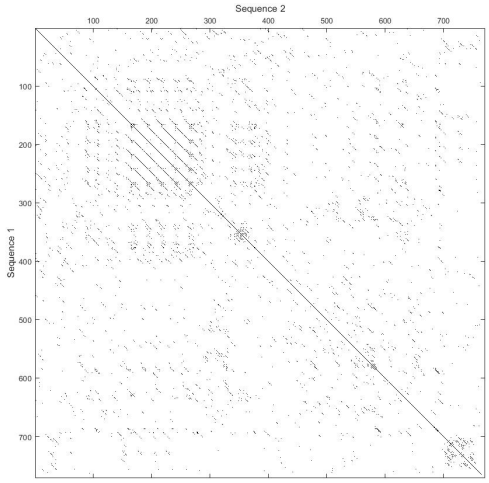


Figure 1.9: Example of Sliding Window Technique

The example in Figure 1.10 shows the similarities between the nucleotide sequences of the prion protein (PrP) of two ruminants, moufflon and golden takin, using the seqdotplot function in the Bio-Informatics Tool kit in MATLAB. The getgenbank function with the accession number for an entry in the National Center for Biotechnology Information (NCBI) database is used, that is, moufflon = getgenbank ("AB060288") and takin = getgenbank ("AB060290") then in the seqdotplot function we provide the sequences along with window size and stringency, that is, graph1 = seqdotplot (moufflon,takin,11,7).



(a) Window Size = 1 and Stringency = 1.



(b) Window Size = 11 and Stringency = 7.

Figure 1.10: Dot Plot to visually compare the sequences of moufflon and takin

1.5.2 Dynamic Programming Algorithm

Certain algorithms tackle a problem by dividing it into smaller subproblems and then combining their solutions to solve the overall problem. However, this approach can lead to solving the same subproblems multiple times, resulting in redundant work and increased running time. Dynamic programming addresses this issue by systematically storing and reusing previously

computed results, thereby significantly reducing unnecessary computations and improving efficiency. To find an optimal alignment in which all possible matches, insertions and deletions have been considered to find the best one is computationally so difficult because of the larger length [13]. For example, a sequence with a length of 300 requires 10^{88} comparisons to be made.

To make the alignment task more manageable, Needleman and Wunsch approached the problem by incrementally constructing the alignment, comparing two amino acids at a time. They began from the ends of the sequences and moved forward step-by-step, evaluating each pair of amino acids. This process considered different scenarios, matching pairs, mismatches, and insertions or deletions, to ensure the highest possible number of character matches. In computer science, this strategy is known as Dynamic Programming (DP), a method used to align sequences of DNA or proteins. The DP algorithm offers a dependable way to compute alignments and has been mathematically validated to generate the optimal alignment between two sequences based on defined scoring rules [1]. The Needleman and Wunsch approach generated,

1. Each potential alignment considers every possible combination of matches, mismatches, and insertions or deletions.
2. A scoring system was used to evaluate the alignment, with the goal of identifying the alignment that yielded the highest score. The alignment with the maximum score was considered the optimal alignment.

1.5.2.1 Formal Description of Dynamic Programming Algorithm :

Figure 1.11 illustrates the possible moves to reach a specific matrix position (i,j). These moves can originate from the previous row and column at position (i-1, j-1), or from any position within the same row or column. Three distinct paths exist in the scoring matrix to reach a given position: a diagonal move from (i-1, j-1) to (i, j) with no gap penalties, or a move from another position within the same row or column, which incurs a gap penalty based on the gap size. For two sequences, a = a_1, a_2, ..., a_n, and b = b_1, b_2, ..., b_n, where $S_{i,j}$ = S (a_1, a_2, ..., a_i, b_1, b_2, ..., b_j) is the score at position i in the sequence a and position j in the sequence b, s(a_i, b_j) is the score for aligning the characters at positions i and j, W_x is the penalty for a gap of length x in the sequence a and W_y is the penalty for a gap of length y in the sequence b.

$$S_{ij} = \max \begin{cases} S_{i-1,j-1} + s(a_i, b_j), \\ \max\limits_{x \geq 1} \left(S_{i-x,j} - w_x \right), \\ \max\limits_{y \geq 1} \left(S_{i,j-y} - w_y \right) \end{cases}$$

Once all positions in the matrix ($S_{i,j}$) are populated, the optimal alignment score will be determined by the highest scoring position found in the final row and column (for a global alignment).

To find the optimal alignment of sequences in the scoring matrix, a second matrix, known as the traceback matrix, is employed, as illustrated in Figure 1.12. This matrix records the positions

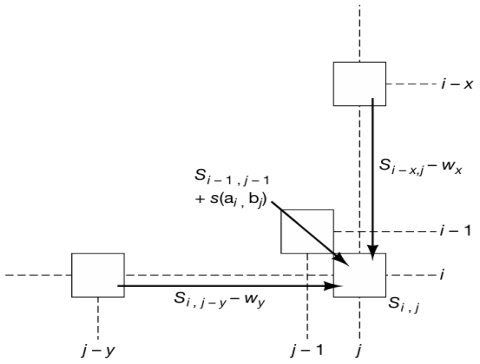


Figure 1.11: Formal description of the dynamic programming algorithm.

in the scoring matrix that led to the highest overall score [1]. These alignments are considered global alignments because they span the entire length of the sequences.

	gap	a1	a2	a3	a4
gap	0	1 gap	2 gaps	3 gaps	4 gaps
b1	1 gap	s11	s21 B	s31	s41
b2	2 gaps	s12	s22	s32	s42
b3	3 gaps	s13	s23 A	s33	s43
b4	4 gaps	s14	s24	s34	s44

Alignment A:

a1	a2	a3	a4
b1	b2	b3	b4

Alignment B:

a1	a2	a3	a4	–
b1	–	b2	b3	b4

Figure 1.12: Traceback Matrix and Corresponding Alignments for different paths

Aligning two sequences a_1 a_2 a_3 a_4 and b_1 b_2 b_3 b_4 using the dynamic programming algorithm is shown in Figure 1.12. The steps followed to align these sequences are as follows;

1. The sequences are placed across the top and down the left side of a matrix, similar to how it is done in dot matrix analysis. However, an additional row and column labeled "gap" are included, allowing for the possibility of starting the alignment with a gap of any length in either sequence. The gap rows are populated with penalty scores corresponding to gaps of increasing lengths, as specified. The upper-right cell is filled with a zero, indicating no gaps in either of the sequences.

2. Maximum possible values are computed for all cells below and to the right of the top row and left column, considering all potential gap lengths or no gaps, following the steps outlined below. The scores for individual matches, such as a_1 - b_1, a_1 - b_2, etc., are derived from a symbol comparison matrix (scoring matrix). To calculate the value

for a specific matrix position, trial values are computed for all possible moves into that position allowed by the algorithm [1]. These allowed moves come from any position above or to the left of the current cell, either in the same row or column, or from the upper-left diagonal position. A diagonal move attempts to align sequence characters without introducing a gap, so no gap penalty is incurred in this case. However, moves from the row or column above will introduce gaps, resulting in one or more gap penalties.

- s_{11} represents the score for an a_1 - b_1 match, which is added to the value of 0 in the upper left position.Based on the alignment algorithm, it can also be reached via vertical or horizontal paths. However, these routes typically yield lower scores since they begin with a gap penalty and accumulate additional penalties as they proceed. In contrast, the diagonal path is often more favorable, as it avoids the extra cost of introducing consecutive gaps.
- To calculate the trial value for s_{12}, we consider the possible moves into this position: Trial 1 involves adding the score for the a_1 - b_2 match to the value of s_{11} (the previous diagonal position) and subtracting a penalty for a gap of size 1. This trial assumes that the alignment for this particular position will align the character a_1 with b_2, introducing a gap in sequence B at position 2.
- The value of s_{22} is determined by evaluating several possible paths leading to it; namely from s_{11}, s_{21}, and s_{12}, as well as from the top row and leftmost column. Among the options, s_{22} takes the maximum score, which may involve adding the match score of a_2 and b_2 to s_{11}, or subtracting a gap penalty from s_{21}. Although other paths originating from further above or to the left may be considered, they are less likely to yield a better score due to the accumulation of multiple gap penalties [1].

3. As the highest scores for each matrix position are determined, the paths that lead to these optimal scores are also recorded. These paths, which represent extending the alignment by adding another matching pair (with or without gaps), are stored in a separate matrix known as the trace-back matrix. For instance, if the best score for reaching position s_{21} comes from a move from s_{11}, this information will be reflected in the corresponding section of the matrix.

4. The paths in the trace-back matrix are followed to construct the alignment. In the given example, the highest-scoring position in the sequence comparison matrix, s_{44}, is located, and the trace-back arrows are followed back to the beginning, forming the alignment path. There may be multiple possible alignments if several paths lead to the highest-scoring position. For instance, s_{43} might also be a high-scoring position, resulting in a different alignment, called trace-back alignment B. This alignment could involve a gap opposite a_2 and another gap opposite b_4, which does not match any symbol. Scoring end gaps is optional depending on the alignment program used.

5. In addition to the main alignment path, which begins from the highest-scoring matrix position, there can be other potential alignments starting from different high-scoring positions. Each of these alignments may also have multiple possible paths through the scoring matrix, with each path corresponding to a distinct alignment [1].

Dynamic programming is also used for multiple sequence alignment; however, it is generally practical only for a limited number of sequences. This limitation arises because the computational complexity increases rapidly with the number of sequences, making large-scale alignments computationally intensive and less feasible. The drawback of this method is that it is computationally very intensive, and as the length of the sequences to be aligned increases, the resources to be used, i.e. time and space (memory) increases exponentially. The resulting alignments are influenced by the selected scoring system. The Global Alignment is performed using dynamic programming method using the Needleman-Wunsch Algorithm. For Local Alignment, a slight modification in the Needleman-Wunsch Algorithm is done, resulting in the Smith-Waterman Algorithm, the major differences being that the scoring matrix must include negative values for mismatch, and when a scoring matrix value becomes negative, it is set to zero in Smith-Waterman. The major limitation of the dynamic programming approach is that the time and space complexity is $\mathcal{O}(\boldsymbol{m \cdot n})$, where m and n are the lengths of the respective sequences to be aligned.

1.5.3 Developments to Optimize Time and Space

To overcome the drawback of the Needleman-Wunsch algorithm, several methods of sequence alignment such as the Hirschberg's algorithm exist which is a modification of the dynamic programming Needleman-Wunsch algorithm, Banded Dynamic Programming which limits DP computation to a diagonal band of the matrix around the main diagonal, and FASTLSA which stands for Fast Linear Search Alignment were developed.

- The Hirschberg's algorithm takes $\mathcal{O}(\boldsymbol{m \cdot n})$ **time** to perform sequence alignment, but requires only $\mathcal{O}(\boldsymbol{min(m \cdot n)})$ **space**. Instead of storing the entire string of characters from the two sequences and the scoring matrix in memory, the Hirschberg's algorithm iteratively divides the sequences by two, then performs the Needleman-Wunsch algorithm on this subset of characters and does this repetitively. This is based on divide-and-conquer over the DP table. This approach saves memory required to do the computation, because instead of storing the full DP table, we store only the current and previous rows (or columns) during computation.

- The Banded Dynamic Programming is effective when sequences are expected to be similar, i.e., with small edit distances under the assumption that the optimal alignment path lies close to the main diagonal of the DP matrix. In many biological applications, the sequences being aligned are similar (e.g., homologous DNA or proteins), so the optimal alignment path usually deviates only slightly from the diagonal. We restrict the DP computation to a band of width $2k$+1 centered on the diagonal, reducing the time complexity to $\mathcal{O}(\boldsymbol{k \cdot \mathbf{min}(m,n)})$.

- The FASTLSA algorithm is a modification of the Hirschberg′s algorithm, which performs the iterative Needleman-Wunsch algorithm by dividing the sequences into k-tuples or characters at a time. It is a variant designed to optimize both speed and memory usage in global sequence alignment, particularly by modifying Hirschberg's algorithm to work with k-tuples instead of single characters so that it reduces the number of alignment

steps, potentially by a factor of k. This reduces the alignment matrix size from $m \times n$ to approximately $\frac{m}{k} * \frac{n}{k}$, thus reducing the time complexity to $\mathcal{O}\left(\frac{mn}{k^2}\right)$.

1.5.4 The Wavefront Alignment Algorithm

The Wavefront Alignment Algorithm (WFA) is an exact gap-affine method that speeds up the alignment by leveraging homologous regions between sequences. Unlike traditional dynamic programming, which has a quadratic time complexity, WFA operates in $\mathcal{O}(n \cdot s)$ time, where n is the read length and s is the alignment score, while requiring $\mathcal{O}(s^2)$ memory. Because the alignment score s is usually much less than the sequence length, the WFA algorithm operates much faster than other methods when aligning short reads. This wavefront alignment technique incrementally calculates partial alignments with progressively higher scores until the optimal alignment is found. Several studies and tests have been conducted on the performance of the wavefront algorithm, and established results show that this algorithm outperforms other well-established methods, while requiring less memory to obtain the optimal sequence alignment.

1.6 Product Survey

Sequence alignment tools are essential for identifying similarities, evolutionary relationships, and functional annotations between DNA, RNA, or protein sequences. The tools in Table 1.2 below represent widely used software packages and algorithms in bioinformatics. Sequence alignment algorithms have evolved significantly to balance accuracy, speed, and memory usage.

- The National Center for Biotechnology Information (NCBI), a US government agency under the National Institutes of Health (NIH), was established in 1988 and serves as a central resource for biological data, computational tools, and scientific literature related to molecular biology, genomics, and bio-informatics. NCBI provides free and open access to most of its data. Tools such as E-utilities allow users to script queries to NCBI databases, which is widely used in automation and large-scale analyses. This site includes a massive database of biomedical literature, scientific papers, journals, and reviews. There is a BLAST tool that is used to compare nucleotide or protein sequences against NCBI databases. A Genome Workbench which is used for visualization of genomic data and PubChem to search for chemical molecules and their bio-activities.
- The RNA Bioinformatics Group at the University of Freiburg is known for developing accessible web services for RNA structure prediction, alignment, and analysis (e.g., RNAfold, RNAalifold, LocARNA). Their teaching platform hosts several algorithm simulators to promote hands-on learning in computational biology. The tools are designed to visually demonstrate the Needleman-Wunsch algorithm, which is a dynamic programming method for global sequence alignment or Smith-Waterman algorithm for local sequence alignment.This tool is frequently used in bioinformatics and computational biology courses because users can enter two sequences (DNA, RNA, or protein) to align and customize match, mismatch, and gap penalty scores. Then, users can see how the dynamic programming matrix is filled and the optimal alignment path through the matrix.

Common Sequence File Formats: In sequence alignment and bioinformatics, various file formats are used to represent sequence data, each serving different purposes such as the storage of raw sequences, quality scores, annotations or alignment results. Each format has specific software support and is tailored for different steps in the bioinformatics pipeline, from raw sequencing to alignment, variant calling, and annotation.

- The **FASTA format** is a simple and widely used text-based format to represent nucleotide or protein sequences. It is used in bioinformatics tools and databases (such as BLAST, FASTA, EMBOSS, and GenBank). Each FASTA entry consists of, Header line (starts with >): contains a sequence identifier and optional description and sequence lines which contain the actual DNA, RNA, or protein sequence.
  Example:

  ```
  > seq1 Homo sapiens beta-globin gene
  GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATTTTGGCTG
  ```

  FASTA format supports multiple entries in a single file. Common file extensions are .fasta, .fa, .fna (nucleotides), .faa (proteins).

- The **FASTQ format** is a text-based format widely used in bioinformatics to store both raw sequence data and quality scores from high-throughput sequencing technologies (e.g., Illumina, Oxford Nanopore). Each sequence entry in a FASTQ file consists of 4 lines: Line 1 begins with @ followed by a sequence identifier, Line 2 contains the raw sequence (nucleotides: A, T, C, G), Line 3 begins with + (optional repetition of identifier) and Line 4 has ASCII-encoded quality scores (same length as the sequence). Example:

  ```
  @SEQ_ID
  GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATTTTGGCTG
  +
  !''*(((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>CCCCCC65
  ```

- **EMBL and GenBank** are two widely used flat file formats for representing annotated nucleotide or protein sequences in bioinformatics databases. While they serve similar purposes and have analogous structure, their syntax differs slightly due to being maintained by separate organizations: EMBL (European Molecular Biology Laboratory) format is maintained by EMBL-EBI (Europe) [3] and GenBank format is maintained by NCBI (USA). The key sections in both formats are; “Header” which contains ID/accession, organism name, date, sequence length, “Features” which has annotated genes, CDS, promoters, etc. and “Sequence” which contains the raw DNA/protein sequence. Table 1.1 shows the differences between EMBL and GenBank formats

```
Example of GenBank Format :
LOCUS       SCU49845     5028 bp    DNA  PLN   21-JUN-1999
DEFINITION  Saccharomyces cerevisiae TCP1-beta gene.
ACCESSION   U49845
FEATURES             Location/Qualifiers
     source          1..5028
                     /organism="Saccharomyces cerevisiae"
     CDS             <1..206
                     /product="TCP1-beta"
ORIGIN
        1 atgaaaac ttttctga tctgactg tggtgctt tggatttt
//
```

```
Example of EMBL Format :
ID   X56734; SV 1; linear; mRNA; STD; PLN; 1859 BP.
XX
DE   Saccharomyces cerevisiae TCP1-beta gene.
XX
FT   CDS             1..206
FT                   /product="TCP1-beta"
XX
SQ   Sequence 1859 BP; 578 A; 435 C; 462 G; 384 T; 0 other;
     atgaaaac ttttctga tctgactg tggtgctt tggatttt
//
```

Feature	EMBL Format	GenBank Format
Source	European Bioinformatics Institute (EBI)	National Center for Biotechnology Information (NCBI)
Header Start	ID (Sequence identifier and metadata)	LOCUS (Sequence identifier and length)
Sequence Data	Begins with SQ	Begins with ORIGIN
Features	Listed with FT	Listed with FEATURES
Keywords	DE (Description)	DEFINITION
Annotations	Provides detailed annotations	Provides similar annotations

Table 1.1: Differences Between EMBL and GenBank Formats

Software	Description	Alignment	Year	Format	Time Complexity	Space Complexity
Genome Magician	Ultra-fast local DNA sequence searches and alignments for next-generation sequencing (NGS) data.	Local	2020	FASTA, FASTQ	$\mathcal{O}(log\ n)$	$\mathcal{O}(n)$
SWIFOLD	Smith-Waterman Acceleration on Intel's FPGA with OpenCL for Long DNA Sequences.	Local	2017-2018	FASTA	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(m \cdot n)$
CUDAlign	DNA sequence alignment of unrestricted size in single or multiple GPUs.	Local, SemiGlobal, Global	2011-2015	FASTA	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(m+n)$
G-PAS (GPU-PairAlign)	GPU-based dynamic programming with backtracking	Local, SemiGlobal, Global	2010	FASTA	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(m \cdot n)$
K-align	fast and accurate multiple sequence alignment (MSA) algorithm known for its speed and scalability	Global	2005	FASTA	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2 + m \cdot n)$
Geneious	Commercial platform with various alignment algorithms like ClustalO, MUSCLE, MAFFT, and LASTZ	Local, SemiGlobal, Global	2005	FASTA, FASTQ, GenBank	$\mathcal{O}(m \cdot n)$ for Pairwise and $\mathcal{O}(m^2 \cdot n)$ for multiple sequences	$\mathcal{O}(n)$ for Pairwise and $\mathcal{O}(n^2)$ for multiple sequences
MUMmer (Maximal Unique Matches)	Uses suffix trees or suffix arrays to find maximal exact matches	Local, SemiGlobal, Global	1999	FASTA	$\mathcal{O}(n)$	$\mathcal{O}(n)$
EMBOSS Needle	Needleman-Wunsch based DP algorithm for global alignment.	Global	1970	FASTA	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(m \cdot n)$

Table 1.2: Softwares used for Sequence Alignment and their features

1.7 Market Survey

Bioinformaticians use sequence alignment to find similarities between protein or DNA sequences, which helps them understand how these molecules function and evolve. With the growing amount of genomic data, there is a need for faster alignment tools. They want a product that can handle both DNA and proteins, let them customize scoring, and align sequences with databases like GenBank. They are interested in using hardware like FPGA for faster processing and lower power use but stress the importance of easy updates. To encourage more use, they suggest making FPGA accelerators accessible remotely and scheduling queries efficiently to save time.

1.7.1 Parameters to be Measured, Monitored, and Controlled

1.7.1.1 Input Parameters

- **DNA Sequence Length (base pairs):** Specify the minimum and maximum supported lengths for each DNA sequence in a pair. This will depend on the memory available on the Alveo U50. Consider offering options for different sequence lengths to cater to a wider range of applications (e.g., whole genome vs. targeted regions).
- **Number of sequence pairs to be aligned:** Define the number of sequence pairs the accelerator can handle concurrently. This depends on the Alveo U50's processing resources. We might offer options for different batch sizes or implement dynamic batching based on sequence lengths.
- **Configuration settings for the BWFA-MEM2 algorithm:** The BWFA-MEM2 algorithm allows for configuration settings, we include them as input parameters. This could involve parameters such as scoring matrices, gap penalties, or optimization options.

1.7.1.2 Monitored Parameters

- **Resource Utilization:** Monitor the usage of various resources on the Alveo U50, including memory usage, logic utilization, Internal processing temperature of the Alveo U50, Power consumption of the accelerator.
- **Memory Utilization:** Track memory usage to ensure efficient allocation and avoid overflows.
- **Logic Utilization:** Monitor the utilization of logic elements to identify potential bottlenecks or underutilized resources.
- **Power Consumption:** Track the power consumption of the accelerator to assess its efficiency and potential thermal management needs. Internal Processing Temperature: Monitor the temperature of the Alveo U50 to ensure it stays within safe operating limits.

1.7.1.3 Controlled Parameters

- **Clock Frequency:** The clock frequency determines the operating speed of the accelerator. Implement a control mechanism to adjust the clock frequency based on workload or power consumption requirements.
- **Operating Temperature:** The cooling system of the Alveo U50 should be controlled to maintain a suitable operating temperature based on the monitored temperature readings.

1.7.2 Target Market

- **Bioinformatics Research Labs:** These labs perform large-scale DNA sequencing and analysis, potentially benefiting from faster sequence alignment tools.
- **Genome Sequencing Centers:** Large centers generating massive amounts of sequencing data could leverage our accelerator for improved processing speeds.
- **Pharmaceutical and Biotechnology Companies:** These companies rely on DNA analysis for drug discovery and development, and faster sequence alignment could accelerate their research cycles.
- **Personalized Medicine Providers:** As personalized medicine gains traction, faster DNA analysis tools might be valuable for tailoring treatments.

1.8 User Survey

Ultimately, we want researchers and bioinformatics professionals to adopt and use our accelerator. A well-designed user experience with clear interfaces, intuitive controls, and minimal technical barriers will increase user adoption. The Comprehensive documentation and informative error messages help users avoid mistakes during installation, configuration, and job submission. This reduces frustration and wasted effort, leading to more efficient workflows. By incorporating user feedback mechanisms into our documentation and support channels, we can gain valuable insights into user experiences and challenges. This information is crucial for future improvements and the development of new features that better address user needs.

Why do people look for other options when there are lot of software available to do sequence alignments?

The number of genomes/protein sequences that gets added to the existing pool increases exponentially each year. Having said this fact, the time taken for matching a query sequence with the whole set of reference sequences available in database also increases exponentially. Hence, there is always an urge to improve the speed of sequence alignment algorithms. Also, there are many algorithms which on slight modification can be made to run in parallel and thus increasing the speed multi-fold. Thus, hardware implementation comes as a promising option.

What do they feel about the proposed product?

Implementation of a sequence alignment algorithm in reconfigurable hardware, such as a Field Programmable Gate Array (FPGA), is a promising option to increase the computation speed because of its nature to exploit multiple levels of parallelism. Also, since the hardware is directly customized for sequence alignment, power consumption is significantly reduced. On the other side, it should be made sure that any modification or improvement in the algorithm in the future should be updated in FPGA with little to no effort, which is generally the case when software is used.

What could one do to promote/improve the usage of this product everywhere?

The accelerator designed for pairwise sequence alignment using FPGA can be linked to a network, so that users from anywhere can access the product remotely. Hence, if multiple queries are received to perform sequence alignment, it has to be scheduled such that the total time taken is minimised. Any options for also parallelising the queries by utilising the resources effectively should be considered.

1.9 Wish Specification

1.9.1 Software Specifications

- Sequences to be aligned - DNA sequences/Nucleotides
- Alignment type - Global
- Scoring system - Custom or Standard
- Input format of the sequences - String of two sequences to be aligned (FASTA).
- Output file format - String of aligned sequences (FASTA).
- Software interface - Command Line Interface (CLI) and Graphical User Interface (GUI).

1.9.2 Hardware Specifications

- Alveo U50 Data Center Accelerator Card
- Host Server running Linux

Chapter 2

Study

2.1 Introduction

In Bio-informatics, Sequence Alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural or evolutionary relationships between the sequences. There are many types of sequence alignment, one of them is Pairwise sequence alignment. Pairwise sequence alignment is very important component in the field of the bio-informatics. As the number of sequences in the genome database is increasing exponentially each year, efficient genome sequencing algorithms is an important area which requires lot of innovation. In this study report, we discuss the current generation algorithms which are used for pairwise sequence alignment so that they can be further accelerated by implementing them in re-configurable hardware like Field Programmable Gate Array (FPGA).

When deciding to perform a sequence alignment, it's crucial to consider the goal of the analysis. Is the aim to explore whether two proteins have similar domains or structures, to determine if they belong to the same family with a shared biological function, or to investigate a common ancestral relationship? The specific objective will guide the approach to the analysis. Several decisions must be made during the process, including selecting the appropriate program, choosing between global or local alignment, determining the type of scoring matrix, and setting the gap penalties [1]. Various algorithms have been created for pairwise alignment, each tailored to different needs, such as optimizing for speed, accuracy, or sensitivity.

Among the foundational algorithms, Needleman-Wunsch performs global alignment by comparing sequences end-to-end using dynamic programming, while Smith-Waterman focuses on local alignment, identifying the most similar subsequences. Heuristic methods like FASTA and BLAST provide faster alternatives for large database searches by identifying short matching words and extending them, trading some accuracy for significant speed gains. The Wavefront Alignment Algorithm (WFA) has emerged as a breakthrough approach enabling sublinear time and space complexity in many practical cases. Building on this, the Bidirectional WFA (BiWFA) improves memory efficiency by performing alignment from both ends and merging in the middle, making it exceptionally suitable for ultra-long sequences and embedded systems. Together, these tools form the core of modern sequence comparison strategies, balancing trade-

offs between speed, accuracy, and scalability.

2.2 Representation of Indels

In a Dot Matrix scheme, a match between two nucleotides in the genome or nucleic acid sequences is shown as a dot, while a mismatch is left blank. This creates a diagonal pattern of dots when there is a string of matched bases. Our goal is to find the best path through this matrix, starting from the top left corner and ending at the bottom right. This path should have the most dots which corresponds to matches and avoid gaps which are represented by horizontal or vertical movements. When one sequence is gapped relative to another, a deletion in one sequence is symmetric with an insertion in the other. For example, a deletion in Sequence A can be seen as an insertion in Sequence B. These Insertions and Deletions are referred to together as *Indels*. Figure 2.1 gives a clear idea on Insertions and Deletions.

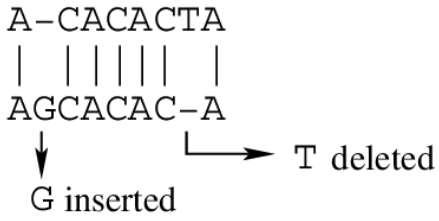


Figure 2.1: Insertions and Deletions for Sequence Alignment.

In this report, we will consider Insertion and Deletion with respect to the sequence along x-axis or the horizontal axis. So, Insertion means a horizontal path insertion or insertion in the x-axis sequence and Deletion means a vertical path insertion or insertion in the y-axis sequence.

2.2.1 Gap Linear vs Gap Affine Algorithms

Gapped and ungapped are the two scoring systems for sequence alignment. The ungapped scoring system considers only matches and mismatches/substitutions. It does not give accurate insights about relation between the given sequences. This system is rarely used, since it does not take all the biological mutations that happen during evolution into account. Insertion or deletion of genomes in sequences can be modeled only with the help of gapped scoring systems. Gapped scoring system can either be Gap Linear or Gap Affine. In nature, entire substrings are often deleted or inserted as a whole, rather than individual nucleotides being added or removed one by one [13]. A gap in an alignment refers to a continuous sequence of spaces in one of the rows. Since the insertion or deletion of substrings is a common evolutionary process, penalizing a gap of length x with a penalty of σx can be excessively harsh. Many real-world alignment algorithms use a more flexible gap penalty scheme, where the cost of a gap spanning x positions grows more slowly than simply adding up the penalties for x separate insertions or deletions.

Let σ be the penalty for one gap. Then the penalty for x continuous gap is σx. Whereas in gap affine system, separate scores are used for gap opening and gap extension. Let the penalty

for gap opening be δ and the penalty for extension be σ which is generally less than δ then penalty for x sized gaps is P(x) = $\delta + \sigma x$. Gap affine scoring system is the best fit for biological sequence alignment because insertion/deletion of genomes can more probably occur in clusters than in unit size. Thus, gap affine system which considers match, mismatch, gap opening and gap extension scores captures all the major phenomena that result in the sequence modification. The pairwise alignment problem is solved using some variation of the Needleman-Wunsch algorithm for Gap linear penalties or the Smith-Waterman algorithm for Gap affine penalties.

2.2.2 Global Sequence Alignment

This is the idea of being able to look at two sequences and figure out what is the most optimal way to align these two sequences to find out how they are similar to each other. If we take the example, "TREE" and "REED" between these two words there are lot of different ways that we can align these words. There are two different ways as shown in Fig 2.2. We can calculate the distance score using;

$$d(a,b) = \begin{cases} 1; if\, a = b \\ \mu; if\, a \neq b \\ \delta; if (a = \text{`}-\text{'}) or (b = \text{`}-\text{'}) \end{cases}$$

Figure 2.2: Different Alignments between TREE and REED

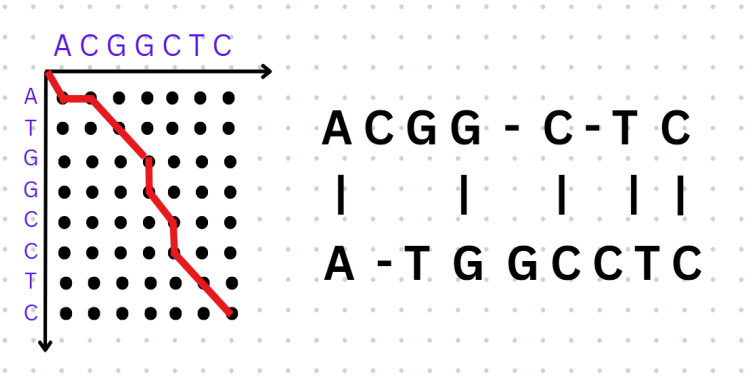


Figure 2.3: Example Alignment Matrix

In this example, we are only looking at two of the possible many sequences. So, to write many possible sequences we are using scoring matrix or alignment matrix. We line up two sequences in a large table to explore all the different possibilities. Let us take a look at the two sequences as shown in Fig 2.3. We can put the first sequences on vertical axis and second sequence on

horizontal axis and then we have dots which signifies all the possible sites. And then by drawing any possible path in here, we can figure out what that alignment would be. If we calculate what the possible score would be for any of these possibilities and find the maximum score then, we would be able to translate this to the mathematical scores and be able to find the optimal alignment by relating the score to the path that took through it to get that score. This seems to be a daunting task to calculate scores for various paths throughout this entire matrix and that is where the idea of dynamic programming comes into play.

2.3 Needleman-Wunsch Algorithm

The Needleman-Wunsch algorithm is a classical dynamic programming method used for global sequence alignment. It aligns two biological sequences (DNA, RNA, or protein) from start to end, optimizing for the best possible overall match, including mismatches and gaps (insertions or deletions). We consider the scoring of matches, mismatches, and indels in the alignment [13]. Instead of selecting a specific scoring matrix and then addressing a redefined alignment problem, we will frame a general global alignment problem where the scoring matrix is provided as input. When mismatches are penalized by some constant μ, indels are penalized by some other constant σ and matches are rewarded with +1, the resulting score is ;
No.of matches + μ * No.of Mismatches + σ * No.of Indels. The corresponding recurrence can be written as;

$$S_{i,j} = \max \begin{cases} S_{i-1,j} + \sigma \\ S_{i,j-1} + \sigma \\ S_{i-1,j-1} + \mu, if a_i \neq b_j \\ S_{i-1,j-1} + 1, if a_i = b_j \end{cases}$$

2.3.1 Example of Needleman-Wunsch Algorithm

Let us consider two sequences, Sequence-A and Sequence-B with lengths *m* and *n* respectively and the scoring scheme used are +1 for matches, -1 for mismatch and gap penalties.

1. **Initialization:** First, we need to create a matrix S of size (m + 1) x (n + 1). Here, S[i][j] represents the best score for aligning first i characters of A with first j characters of B. Then, we need to fill the first row and column with gap penalties that is;
   S[0][0] = 0, S[0][j] = j * gap_penalty and S[i][0] = i * gap_penalty

2. **Fill the Matrix:** We need to fill the Matrix using the recurrence relation as shown above in each cell to compute scores. We store the backtracking pointers as we calculate the dynamic programming table. These pointers indicate the path taken to reach each cell in the table, allowing us to reconstruct the alignment once the table is fully computed.

3. **Traceback for Alignment:** The optimal alignment is determined by beginning at the sequence positions with the highest score and tracing backward through the positions

that led to this maximum value. We start from S[m][n], and move diagonally if match or mismatch was chosen, move up if a gap was added to B (insertion in horizontal axis) and move left if a gap was added to A(insertion in vertical axis). We continue this and build two aligned sequences by adding matched characters or gaps until we reach S[0][0].

The pseudo code is shown in Listing 2.1 and the results of Needleman-Wunsch Algorithm for the sequences "GATTACA" and "GCATGCU" are shown below.

```
Score Matrix:
     0    -1    -2    -3    -4    -5    -6    -7
    -1     1     0    -1    -2    -3    -4    -5
    -2     0     0     1     0    -1    -2    -3
    -3    -1    -1     0     2     1     0    -1
    -4    -2    -2    -1     1     1     0    -1
    -5    -3    -3    -1     0     0     0    -1
    -6    -4    -2    -2    -1    -1     1     0
    -7    -5    -3    -1    -2    -2     0     0

Aligned Sequences:
    G-ATTACA
    GCA-TGCU
```

Listing 2.1: Pseudo Code of Needleman-Wunsch Algorithm

```
// Fill the first row and column with gap penalties
    for i = 1:len1 + 1
        scoreMatrix(i, 1) = (i - 1) * gap;
        tracebackMatrix(i, 1) = 2; // Up
    end
    for j = 1:len2 + 1
        scoreMatrix(1, j) = (j - 1) * gap;
        tracebackMatrix(1, j) = 3; // Left
    end
    tracebackMatrix(1, 1) = 0; // No direction at the origin
    // Fill the rest of the score matrix
    for i = 2:len1 + 1
        for j = 2:len2 + 1
            matchScore = scoreMatrix(i - 1, j - 1) + (seq1(i - 1) ==
                seq2(j - 1)) * match + (seq1(i - 1) ~= seq2(j - 1))
               * mismatch;
            gapSeq1 = scoreMatrix(i - 1, j) + gap;
            gapSeq2 = scoreMatrix(i, j - 1) + gap;
            [scoreMatrix(i, j), tracebackMatrix(i, j)] = max([
               matchScore, gapSeq1, gapSeq2]);
        end
    end
```

2.4 Smith-Waterman Algorithm

Smith-Waterman Algorithm is pretty much similar to the Needleman-Wunsch Algorithm. The main difference is that all the negative values we get during the matrix preparation becomes zero in the Smith-Waterman. In 1981, Temple Smith and Michael Waterman observed that the most biologically significant areas in DNA and protein sequences were the sub-regions that aligned well, while the less-related regions were less important [1]. To detect these key aligned segments, they introduced an important adaptation of the Needleman-Wunsch algorithm called the Smith-Waterman algorithm, which is tailored specifically for local sequence alignment.

From a biological perspective, irrelevant diagonal paths from the source to the sink may yield a higher score than the biologically relevant alignment, as mismatches are typically penalized less than indels [13]. When meaningful similarities exist in certain regions of DNA fragments but not in others, biologists aim to maximize the alignment score to capture the biologically significant matches of s(a_i ... a_i' , b_j ... b_j') over all sub-strings a_i ... a_i' of a and b_j ... b_j' of b. This is referred to as the local alignment problem, where the alignment does not necessarily span the entire length of the sequences, unlike the global alignment problem. The global alignment problem involves finding the longest path between the vertices (0, 0) and (m, n) in the matrix, while the local alignment problem focuses on identifying the longest path among various paths between arbitrary vertices (i, j) and (i', j') within the matrix [13].

Instead of finding the longest path from every vertex (i, j) to every other vertex (i', j') the longest local alignment problem can be reduced to finding the longest path from source (0, 0) to every other vertex. A small difference in the recurrence reflects the transformation of this;

$$S_{i,j} = \max \begin{cases} 0 \\ S_{i-1,j} + \sigma \\ S_{i,j-1} + \sigma \\ S_{i-1,j-1} + \mu, if a_i \neq b_j \\ S_{i-1,j-1} + 1, if a_i = b_j \end{cases}$$

The largest value of $S_{i,j}$ across the entire matrix represents the score of the best local alignment between sequences a and b. In global alignment, we simply focus on the last cell of the matrix, S[m][n]. In contrast, optimal local alignment identifies only the longest path within the matrix. Additionally, multiple local alignments may be biologically significant, and various methods have been developed to find the k-best non-overlapping local alignments . These methods are particularly effective when comparing multi-domain proteins that share similar segments but may have undergone rearrangements or shuffling between proteins [13].

2.5 Gotoh Algorithm

The Gotoh Algorithm is an optimization of the Needleman-Wunsch and Smith-Waterman algorithms for global and local sequence alignment, respectively. It introduces affine gap penalties, which model biological gaps (insertions or deletions) more realistically than linear penalties.

The Gotoh algorithm offers a faster approach that takes only $\mathcal{O}(\boldsymbol{M} \cdot \boldsymbol{N})$ time. This penalty format works well for most sequencing applications. The algorithm uses three matrices, they are **D**, **P**, and **Q**. The **D** matrix holds the highest score found so far for aligning the sequences up to that point. The **P** matrix keeps track of the best score possible if there are insertions of varying length in one sequence (insertions in horizontal axis). Similarly, the **Q** matrix stores the best score considering deletions of varying length in one sequence (insertions in vertical axis).

This algorithm computes the optimal alignment of two sequences when using an affine gap scoring. Here, the scoring of a long consecutive gap is favored over a collection of small gaps with the same combined length. This incorporates the assumption that a single large insertion or deletion event is biologically more likely to happen, compared to many small insertions or deletions. Fewer gaps even if they are long should be preferred over many short gaps. Gotoh's dynamic programming approach tabulates optimal sub-solutions in matrices M, I and D.

2.5.1 Explanation of Gotoh Algorithm

When mismatches incur a penalty of a constant μ, gap openings are penalized by a constant α, gap extensions by a constant β, and matches are given a reward of +1, the overall alignment score can be expressed as:
No.of matches + μ * No.of Mismatches + α * No.of Gap Openings + β * No.of Gap Extensions. The corresponding recurrence can be written as;

$$P_{i,j} = \max \begin{cases} P_{i-1,j} + \beta \\ Q_{i-1,j} + \alpha + \beta \end{cases}$$

$$Q_{i,j} = \max \begin{cases} Q_{i,j-1} + \beta \\ P_{i,j-1} + \alpha + \beta \end{cases}$$

$$D_{i,j} = \max \begin{cases} P_{i,j} \\ Q_{i,j} \\ D_{i-1,j-1} + \mu, if a_i \neq b_j \\ D_{i-1,j-1} + 1, if a_i = b_j \end{cases}$$

Let **A** = $a_1\ a_2\ a_3\ ...\ a_m$ and **B** = $b_1\ b_2\ b_3\ ...\ b_n$ be the two sequences to be aligned. The best possible score for any entry in the matrices **D, P** and **Q** is to choose the minimum value as shown in equation 2.1. In the beginning of the computation, we set $D_{i,0} = P_{i,0}$ = α + i*β and $D_{0,j} = Q_{0,j}$ = α + j*β where $0 \leq$ i $\leq$ m, $0 \leq$ j $\leq$ n for global alignment. Otherwise, we set zero penalty for initial gaps. $D_{i,0} = P_{i,0} = 0$ and $D_{0,j} = Q_{0,j} = 0$ where $0 \leq$ i $\leq$ m, $0 \leq$ j $\leq$ n for aligning locally similar sub-sequences.

2.5.2 Example of Gotoh Algorithm

Sequence A = "CCGA" and Sequence B = "CG" are taken as examples for global alignment using gap-affine Gotoh algorithm.

Scoring system used is {Match, Mismatch, Gap opening, Gap extension} = {1,-2,-3,-1}. The three matrices in Figure 2.4 are filled using recurrence relations provided in the section 2.5.1. The back tracing path to track the optimal alignment that led to the maximum score is shown in blue color. There can be multiple optimal alignment path one such alignment obtained is:

CCGA
_ C G _

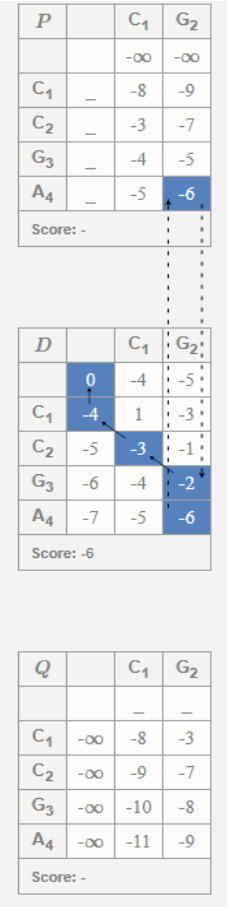


Figure 2.4: Alignment of Sequences using Gotoh Algorithm

2.6 Myers Algorithm

Traditional algorithms for finding the longest common subsequence (LCS) have a constant time complexity of $\mathcal{O}(\boldsymbol{M \cdot N})$ which means they take the same amount of time to compare highly similar and completely unrelated sequences. This inefficiency is addressed by the Myers algorithm. Myers algorithm exploits sequence similarity to reach a time complexity of $\mathcal{O}(\boldsymbol{M \cdot ND})$, where N is the total length of the two sequences and D denotes the minimum number of edit steps required to convert one sequence into the other. The algorithm runs more quickly when the sequences are more similar, resulting in a smaller D. Statistically, D tends to be small in sequencing tasks, making the Myers algorithm performance significantly better with increasing data size compared to other LCS algorithms.

2.6.1 Edit Graph

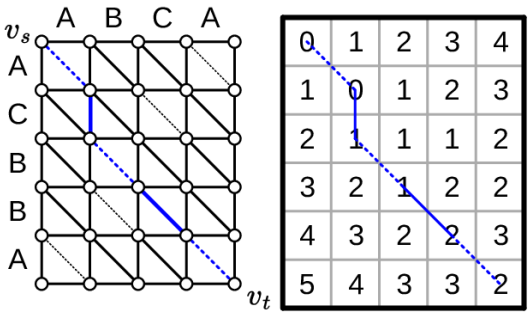


Figure 2.5: Edit Graph corresponding to the Alignment of strings “ACBBA” and “ABCA”

Let $A = a_1, a_2, ..., a_N$ and $B = b_1, b_2 ..., b_M$ be the two sequences to be aligned. We place the sequence A along the x-axis and sequence B along y-axis. The rectangle that covers the plane from (0, 0) to (N, M) is called Edit Graph, as shown in Figure 2.5. The vertices of edit graph are covered by horizontal, vertical and diagonal edges. A horizontal edge which connects (x-1, y) to (x, y) denotes an insertion in sequence A and the vertical edge that connects (x, y-1) to (x, y) denotes a deletion in sequence A. The point (x, y) is called a match point when $a_x = b_y$ and a diagonal edge is drawn from (x-1, y-1) to (x, y).

The goal is to find a path from (0, 0) to (N, M) with the longest common subsequence. It means the path should have maximum number (L) of match points or maximum number of diagonal edges (L). The worst case path to travel from (0, 0) to (N, M) is to go all the way down from (0, 0) to (0, N) and then travel right to reach (N, M). Let D be the number of horizontal and vertical edges in the path. This case will have D = N + M. We need to find an algorithm to maximize the number of diagonal edges (L). From the above relation, it is safe to say that the maximizing L is same as minimizing D.

2.6.2 Explanation of Myers Algorithm

We have to find the maximum value of L or minimum value of D which reaches the vertex (N, M). In other words, we aim to find the minimum D path on diagonal K = N-M, such that the point (N, M) is the furthest reaching point of D path on diagonal K = N-M. Thus we increase D from 0 to its worst value (N + M) and then compute furthest reaching D paths on all possible diagonals for that value of D. Then, we check whether any of the D path ends at vertex (N, M). If not, we increment the D value and recompute until we reach the point *(N, M)*.

To find the furthest reaching D path endpoint in diagonal k, we compare the furthest reaching (D-1) path endpoint in diagonal (k-1) and (k+1). The path with farthest endpoint among the two is selected and is extended by a horizontal edge, if (k-1) diagonal is selected or with a vertical edge, if (k+1) diagonal is selected, thus reaching D path on diagonal k. Then, the D path is extended on the diagonal k, if match points exist thus reaching the farthest endpoint of D path on diagonal k. The coordinate is then stored in a matrix for subsequent D path calculations.

2.7 Wavefront Alignment Algorithm

The Wavefront Alignment Algorithm (WFA) is an exact gap affine pairwise alignment algorithm that progressively computes partial alignments of increasing score until the optimal solution or alignment is obtained. WFA computes the optimal gap affine alignment in $\mathcal{O}(ns)$ time, which depends on the sequence length n and the optimal alignment score s. This allows scaling with sequence length provided that the error rate between the sequences remains moderate. The space complexity of the WFA algorithm is $\mathcal{O}(s^2)$. The WFA algorithm introduces a novel technique that calculates only a subset of the Dynamic Programming (DP) matrix cells needed to determine the optimal alignment [12]. By doing so, it performs exact pairwise sequence alignment between a query and each candidate in the database, producing results identical to the affine-gap Gotoh algorithm. Therefore, the WFA algorithm can effectively replace Gotoh's method to boost performance. Additionally, WFA scales more efficiently with longer sequences, delivering speed improvements of 10 to 100 times compared to traditional approaches.

2.7.1 Explanation of WFA Algorithm

Let the query $q = q_0, q_1 ..., q_{n-1}$ and the text $t = t_0, t_1, ..., t_{m-1}$ be strings of length n and m respectively. The pairwise global alignment problem involves finding the alignment from position (0, 0) to (n, m) that minimizes the penalty score, considering matches, substitutions, and gaps. As previously mentioned, this is typically addressed using variants of the Gotoh algorithm. The recurrence relations for the dynamic programming matrix used by Gotoh's method are outlined in section 2.5.1. Most implementations apply strictly positive penalties for mismatches (x), gap openings (o), and gap extensions (e), where all are greater than zero. Furthermore, the match score is set to zero, which helps leverage sequence similarities and speeds up the alignment process. This simplification enables a more efficient and streamlined approach to gap-affine alignment.

The wavefront vectors $P_{i,j}$, $Q_{i,j}$ and $D_{i,j}$ keeps track of alignments ending in an insertion, deletion, or match/mismatch separately. Beginning with $D_{0,0} = 0$, the WFA algorithm calculates wavefront vectors by applying two key operations, *extend()* and *compute()*. These operations are explained below:

- The extend() operator increments each offset in the wavefront vector based on the count of consecutive matching characters between the sequences. Since the match penalty is zero (a=0), the score along the diagonal stays unchanged, meaning that matches do not reduce the alignment score.
- After extending the offsets of the wavefront vector, the compute() operator calculates the subsequent wavefront vectors using the recurrence relations described in section 2.5.1, based on the penalty parameters x, o and e defined by the gap-affine model. The WFA algorithm alternates between *extend()* and *compute()* operations until it processes the entirety of both sequences. Once it reaches the optimal alignment score $D_{N,M}$, the algorithm backtracks through the wavefront vectors all the way to the starting wavefront $D_{0,0} = 0$.

2.7.2 Example of Calculating Wavefront Vectors using WFA Algorithm

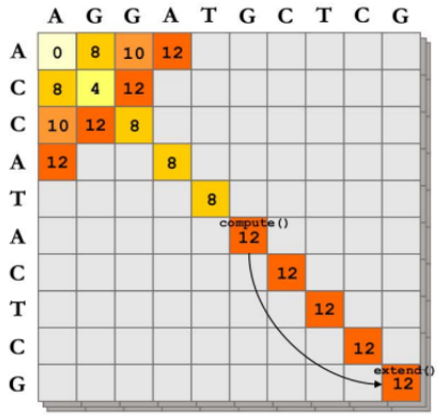


Figure 2.6: Alignment of two sequences using WFA.

Figure 2.6 illustrates an example of aligning two sequences with the WFA algorithm, using penalty values (x,o,e)=(4,6,2). The related cells in the dynamic programming matrix are filled according to the classical Gotoh algorithm.The values of the elements in the matrix is obtained using the Gotoh DP algorithm explained in section 2.5. We know that the wavefront vectors $P_{i,j}$, $Q_{i,j}$ and $D_{i,j}$ denote the point in the DP matrix. Initially, vector $D_{0,0}$ is the first element of the matrix and is chosen to be 0. Note that from Figure 2.6, the set of values that the DP matrix takes are 0, 4, 8, 10 and 12, which means that the score **s** must also lie in this range. These calculations can be verified using the WFA recursive relations from equations in section 2.5.1.

2.8 Bi-directional Wavefront Alignment Algorithm

The Bi-directional Wavefront Alignment (BWFA) algorithm builds upon the strengths of the traditional WFA algorithm and addresses its memory limitations. BWFA tackles the memory bottleneck of WFA by performing the alignment simultaneously in both directions. Imagine two wavefronts, one starting at the beginning and another at the end of the sequences. The computation of wavefronts in the reverse direction also require the reversal of both the sequences that are to be aligned. The computation of wavefronts is done both in forward and reverse directions until they meet. The coordinates at which the wavefronts collide is called the junction or breakpoint. This junction roughly divides the optimal alignment score in half. The same procedure is then applied recursively on both the sides of the junction to find further junctions as illustrated in Figure 2.7. It is shown that the BWFA algorithm reduces the space complexity to $\mathcal{O}(s)$ compared to $\mathcal{O}(s^2)$ of the WFA algorithm. The WFA algorithm paved the way for optimal alignment methods that can efficiently scale to long sequences. However, its $\mathcal{O}(s^2)$memory usage soon became a bottleneck when dealing with very long or highly noisy sequences.

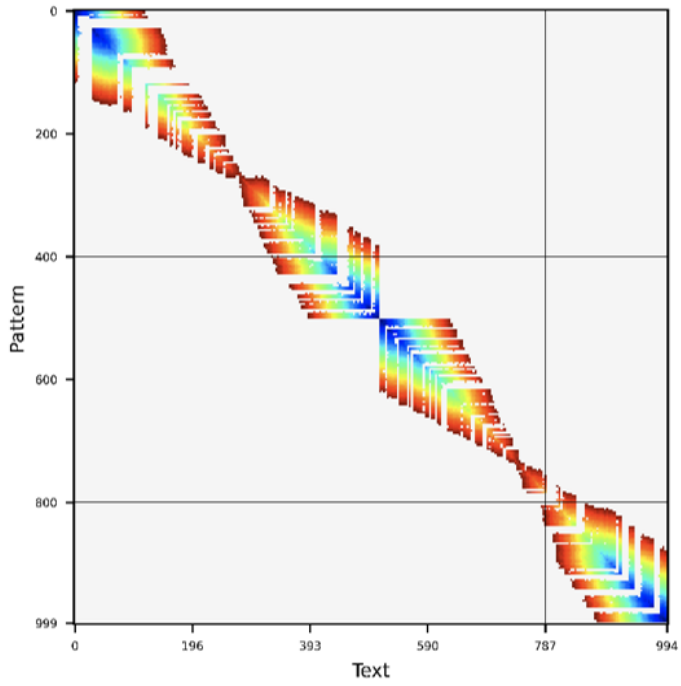


Figure 2.7: Computed cells in the BWFA and their junctions.

2.9 High Level Synthesis

If we look at the processors being released last couple of years, we will notice that many of them have custom accelerators on chip. By embracing customization we can achieve high performance while consuming small amount of area and power and that is why all major vendors are eager to add custom accelerators to their product. One promising way to reduce the design effort is to use the High-Level Synthesis tools. It allows us to design in High Level Languages such as C or OpenCL and the tool automatically determines the micro-architecture and generates low level designs. So, it drastically reduces the design effort and leads to faster time to market. HLS reduced the programmer effort in building custom accelerators. Nvidia was able to simplify their code by 5x. HLS decreased time by 50% and overall development time, including verification by 40%.

High-Level Synthesis (HLS) is a transformative technology that converts high-level behavioral descriptions of hardware into Register-Transfer Level (RTL) models, serving as a pivotal component within the Electronic System Level (ESL) design flow. HLS leverages untimed functional descriptions, written in high-level programming languages such as C, C++, or SystemC, that focus on the functionality of the hardware without specifying the timing details of operations to hide all the traditional hardware design complexities from designers. The untimed functional descriptions can be augmented with interface specifications, detailing how the hardware block will interact with the rest of the system. These interface definitions, play a critical role in the synthesis process, influencing decisions made during the conversion from high-level descriptions to RTL. HLS tools extract inherent parallelism from the high-level descriptions, meticulously scheduling operations, allocating resources, and optimizing resource sharing. This ensures that the design achieves the necessary performance metrics while main-

taining a compact area footprint.

Commercial HLS tools vary significantly in their capabilities, including the range of input languages they accept, the degree of automation they offer, the quality of the synthesized hardware, and their integration with other ESL design tools. This diversity allows designers to select the most appropriate tool for their specific needs, thereby enhancing productivity and design efficiency. For this project we are using Xilinx Vitis HLS tool, it takes application code and transform it into design targeted for accelerator. Vitis provides various development tools to emulate, analyze and debug the design. Vitis HLS generates RTL code from the high-level language design. Output of HLS (RTL files) can be synthesized into FPGA bitstream for on-board execution.

2.9.1 Xilinx Vitis HLS

It is a part of the Xilinx Vitis unified software platform, which integrates software development and hardware acceleration. The main goal of this software tool is converting a hardware design description into final FPGA configuration. Several languages including VHDL/Verilog, SystemC, C/C++ and OpenCL can be used to describe a hardware design. The main idea behind the HLS is using software based languages to develop hardware modules automatically or just with a minor modification to the original C-function. Not only this approach facilitates the design process, but it also supports providing and reusing libraries. HLS provides a relatively easy way of describing the architecture that meets our needs, we use compiler directive and tell HLS to generate which type of design we want (more resources and more area or less resources and less area). Here are some of the key features of Vitis HLS:

- **Language Support:** Supports C, C++, and SystemC and OpenCL.
- **Optimization:** Offers advanced optimization techniques for performance, area, and power, including loop pipelining, loop unrolling, and dataflow optimization using #pragma directives.
- **Seamless Integration:** Seamless integration with the Vitis unified software platform and Vivado for implementation and IP packaging.
- **Custom Hardware Acceleration:** Target custom accelerators for compute-intensive applications (AI, image processing, finance, etc.).
- **Interface Synthesis:** Provides automatic synthesis of various interfaces, such as AXI4, for easy integration with other hardware components.
- **Verification and Debugging:** Includes comprehensive tools for debugging and verification, such as C/RTL co-simulation and waveform viewers.
- **Design Space Exploration:** Allows for rapid exploration of multiple design architectures and performance trade-offs without modifying the source code.

- HLS can automatically analyze and exploit concurrency in an algorithm, insert registers in critical paths and achieve desired clock frequency. HLS can generate control logic that directs the datapath and map data onto storage elements, computation onto logic elements.

The restrictions in HLS exist because the synthesized hardware must be deterministic, static, and time-bounded, unlike general-purpose software. Here are the key coding limitations of High-Level Synthesis (HLS) that designers should consider:

- **No Dynamic Memory Allocation:** Functions like malloc, calloc, or new are not synthesizable. All memory allocation must be static or predetermined at compile time.
- **No System Calls:** Standard OS-dependent features (e.g., printf, fopen, exit) are not supported for synthesis (though some like printf are available in simulation).
- **No Recursive Function Calls:** Recursion leads to indeterminate stack size and is not synthesizable. Loops or iterative equivalents must be used instead.
- **Limited Pointer Usage:** Pointers are allowed but heavily restricted: No pointer arithmetic (like ptr++, ptr += n), No dynamically changing base addresses and No function pointers or arrays of pointers.
- **Longer Iteration Times for RTL Verification:** Though HLS speeds up initial design, debugging and verification of the generated RTL can be time-consuming. Debugging the RTL generated from high-level C/C++ code can be difficult due to the abstraction gap.
- **Limited Low-Level Control:** HLS abstracts hardware behavior, making it harder to finely control timing, placement, and critical path optimizations compared to hand-written RTL.

2.9.2 HLS Design Flow

1. **HLS Component Development Flow:** The HLS component workflow, illustrated in Figure 2.8, offers a suite of tools to simulate, evaluate, implement, and fine-tune C/C++ code for deployment on programmable logic, aiming to achieve high throughput and low latency. A key aspect of this flow is the insertion of appropriate pragmas to define function interfaces and to enable loop and function pipelining. This process forms the core of the HLS flow [23]. Here are the steps for the development of the HLS component from a C++ function:
   (a) **Design:** The first step is the design capture in which the design description is verified and possible errors are reported. The capture process mainly focuses on the linguistic syntax and semantics of the hardware modules, connections among them and their ports.
   (b) **C-Simulation:** After writing the code, simulation is an optional but essential step. Simulation is one of the main debugging techniques that designers spend most of their time to verify the functionality of the C/C++ code with the C/C++ test bench.

(c) **Code Analysis:** This is the primary step before applying synthesis optimization techniques, Data and control dependencies, timing and hierarchy analysis are the important tasks in this step. This step analyzes the performance, parallelism, and legality of the C/C++ code which helps the next step to apply synthesis optimizations more efficiently.

(d) **C-Synthesis:** Synthesis is the most important part. Generally, two levels of synthesis are available; High-Level Synthesis which translates C/C++ or SystemC description of a design into its equivalent RTL using the v++ compiler and Logic Synthesis which converts the RTL description into a netlist of logic gates.

(e) **C/RTL Co-Simulation:** There are different levels of simulation provided by Vitis HLS such as high-level C Simulation, low-level HDL Simulation and hardware-software Co-Simulation. The high-level C-Simulation can be used to verify functionality of a design, the low-level HDL simulation can be used for functional correctness as well as cycle accurate verification. The Co-Simulation integrates the software and the hardware prts in order to bring the cycle accurate verification into the high-level synthesis design flow by verifying the RTL code generated using the C/C++ test bench.

(f) **Implementation and Package:** The implementation step uses the FPGA library and building blocks to implement the netlist generated by the synthesis step and performs all the necessary tasks to place and route the netlist onto the device resources. Then, bitstream file is generated which can be used to program an FPGA.

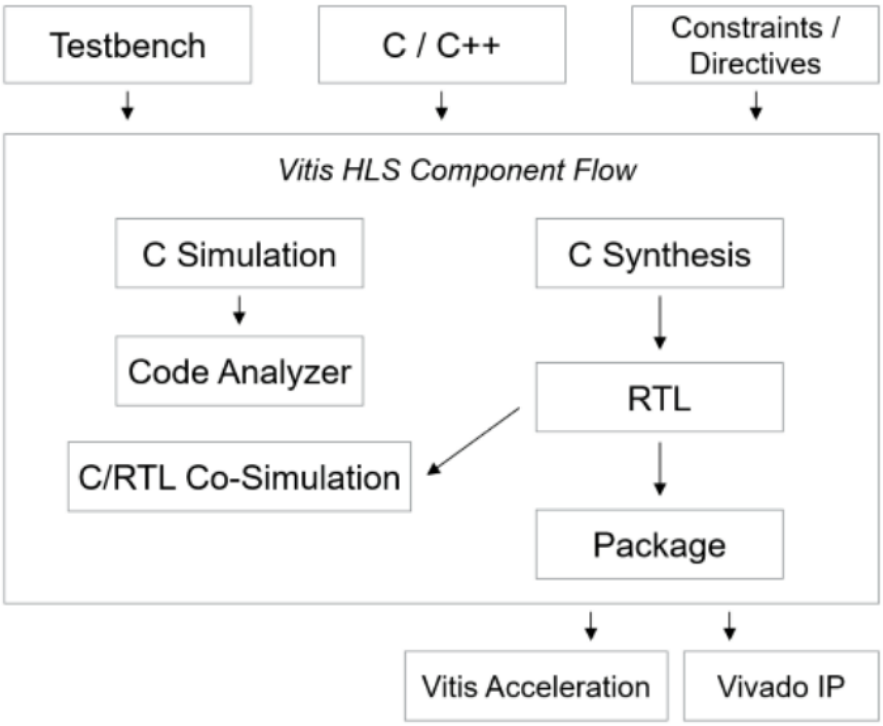


Figure 2.8: HLS Component Development Flow

2.9.3 C/C++ Constructs to RTL Mapping

The table 2.1 shows how the constructs are mapped into the RTL Hardware components:

C/C++ constructs	HW Components
Functions	Modules
Arguments	I/O Ports
Operators	Functional Units
Scalars	Reg/Wires
Arrays	Memory
Control flows	Control logic

Table 2.1: Mapping C/C++ constructs to HW Components

2.9.4 Non Synthesizable Constructs of C/C++

Not all constructs of C/C++ are synthesizable and usage of such constructs may cause issues during the design process. Here are few of them:

- **System Calls:** System calls, which involve performing operations on the operating system where the C/C++ application runs, cannot be synthesized into hardware. These calls must be re- moved before synthesis. Examples of system calls include 'printf()' and 'fprintf()'.
- **Use of Dynamic Memory:** Operating systems handle dynamic memory allocation using resources that are managed and released during runtime. For successful hardware synthesis, the design must be completely self-contained with all necessary resources explicitly defined. Functions like 'malloc()', 'alloc()', and 'free()' are examples of dynamic memory usage that must be avoided.
- **Pointer Limitations:** HLS imposes several restrictions on the use of pointers. Pointer casting between native C/C++ types is not supported, although casting among native types is permitted. Pointers can be used for arrays as long as they point to scalars or arrays of scalars, but arrays of pointers cannot point to other arrays of pointers. Additionally, function pointers and pointers to pointers are not supported.
- **Recursive Functions:** Defining recursive functions is problematic in HLS. Moreover, Vitis HLS does not sup- port tail recursion, which involves a limited number of function calls. As a result, recursion should be avoided in designs intended for hardware synthesis.

2.10 Installation and configuration of Alveo U50

This section describes the features of the Alveo U50 card and its installation process to make it work alongside with Xilinx Vitis Software.

2.10.1 Card Features

The Alveo U50 data center accelerator card supports PCIe Gen3 x16 and is also compatible with Gen4 x8. Built on Xilinx's 16nm UltraScale+ technology, it includes 8 GB of HBM memory to deliver high-performance and flexible acceleration for memory-intensive and compute-heavy tasks such as analytics, database operations, and machine learning inference [**?**]. Some of the Alveo U50 accelerator card features are shown in Figure 2.9.

Card Component	Alveo U50
FPGA	UltraScale+™ XCU50 FPGA
HBM	8 GB - two 4 gigabyte (GB) HBM memory stacks
	Split into 32 256 MB channels
Network Interface	1x QSFP28
	Supporting 100 GbE, 40 GbE, or 4x10/25 GbE
PCIe	16-lane PCI Express
	PCIe Integrated Endpoint block connectivity
	Gen1, 2, or 3 up to x16, Gen4 x8
	Single or dual Gen4 x8
I2C Bus	✓
Status LEDs	✓
Power Management	Power management with system management bus (SMBus) voltage, current, and temperature monitoring
Electrical Design Power	75W PCIe slot functional with PCIe slot power only
Configuration Options	1 gigabit (Gb) Quad Serial Peripheral Interface (SPI) flash memory
	UltraScale+ device configurable over USB/JTAG and Quad SPI configuration flash memory
UART	UART access through the maintenance connector

Figure 2.9: Alveo U50 Card Features

2.10.2 Card installation and validation

Alveo U50 supports PCIe X16, so it has to be placed in a proper PCIe slot on the System. For using Alveo U50 card we first need to install the deployment software.

The deployment includes the following software components:

- **Xilinx runtime (XRT):** XRT supplies the necessary libraries and drivers for running applications on Alveo accelerator cards. It also includes command-line tools like xbutil

and xbmgmt. You can access help for these utilities at any time by using the *–help* flag.

- **Deployment platform:** The deployment platform includes the essential base firmware required to execute pre-compiled applications but does not support application development or compilation. To build new applications, the development software must be installed. Although it's possible to install the development tools on a system with an Alveo card, this is not required for simply running existing applications.

  The card is then flashed with the base image and installation is validated using the commands "xbmgmt program", "xbmgmt examine" and "xbutil validate". To validate the installation an example project is created in the vitis tool and the software emulation, hardware emulation and hardware steps were performed in order to generate the bitstream and program the device. The test was successful and the card was successfully brought up for use.

2.11 Target Specifications

2.11.1 Functionality

- Alignment of DNA sequences of varying lengths based on Alveo U50 memory.
- Significant speedup in throughput (number of sequence pairs aligned per second) compared to a CPU-based implementation on a standard benchmark dataset.
- Efficient utilization of resources on the Alveo U50 to avoid bottlenecks.

2.11.2 Hardware Platform

- The design will be implemented on Alveo U50 FPGA datacenter card.
- HLS (High-Level Synthesis) tools from Xilinx are used to design the accelerator in a C/C++ language.

2.11.3 Input and Output Formats

- **Command-Line Interface (CLI)**: The Input to the algorithm is provided as two sequences, which are a string of characters representing nucleotide bases.
- The output is obtained as two string of nucleotide bases which are aligned. Both the compact and full CIGAR formats are generated, where CIGAR stands for Concise Idiosyncratic Gapped Alignment Report. It provides a streamlined representation of sequence alignments and is commonly used in the Sequence Alignment/Map (SAM) file format to describe how reads align to a reference sequence [2].

- A CIGAR string consists of pairs in the format <integer> <operation>, such as 76H130M. Here, each operation is denoted by a single uppercase letter and represents a specific type of alignment column, such as a match, insertion, or deletion. The accompanying integer indicates how many consecutive times that operation occurs. These operations are further detailed in the table below, with each one corresponding to a distinct alignment behavior.
- An example of storing an alignment in compact and full CIGAR format is; for example consider two sequences "AAA _ CGT" and "ACATCG _" which needs to be aligned. From Table 2.2, in full CIGAR format, the alignment will be stored as MXMIMMD, since the first character in the two sequences correspond to a match, second character corresponds to a mismatch, and so on. This full CIGAR format is stored in the CPU after unpacking compact CIGAR from the FPGA.
- In compact CIGAR format, the information regarding matching characters is omitted and only the differences between the two sequences is stored in order to save memory. Hence, in compact CIGAR format, the alignment will be stored as XID.

Operation	**Description**
M	A match refers to an alignment column that includes a pair of bases, one from each sequence. These bases can either be the same (indicating an exact match) or different (indicating a mismatch).
D	A deletion represents a gap in the target (reference) sequence, meaning that one or more bases are present in the query sequence but missing from the target at that position.
I	An insertion indicates a gap in the query sequence, meaning that one or more bases are present in the target (reference) sequence but absent from the query at that specific position.
S	This refers to a portion of the query sequence that is excluded from the alignment but still retained in the read. *S* (Soft clipping) typically marks unaligned bases at the beginning or end of a read, indicating that the full read is present but only a portion aligns to the reference sequence.
H	This refers to a portion of the query sequence that is excluded from the alignment and also removed from the reported read. *H* (Hard clipping) means only the aligned portion of the query sequence is retained in the alignment record, while the clipped segments are omitted entirely.
=	An alignment column showing a two identical letters.
X	An alignment column showing a mismatch.

Table 2.2: Description of operators in CIGAR format.

- **Graphical User Interface (GUI)**: For users with moderate expertise, providing a user-friendly interface allow users to upload DNA sequence data files (e.g., FASTA format) through a drag-and-drop or file selection option. We can use charts or graphs to display resource utilization (memory, logic) during processing.

2.12 Summary

The various processes of comparing two biological sequences (DNA, RNA, or protein) to identify regions of similarity or difference are explained in this section. These algorithms are fundamental in bioinformatics for tasks like gene identification and functional analysis. BWFA efficiently calculates the alignment score and path by expanding from both ends of the sequences simultaneously. Future work will be dedicated to realizing the BWFA algorithm on software and hardware, and to explore the possibilities of modifications in the existing algorithms for achieving further enhancements in speed and reduction in complexity. This could involve algorithmic optimizations or tailoring the approach to specific sequence characteristics.

Chapter 3

Design

3.1 Introduction

This chapter explains in detail about the working of Bidirectional Wavefront Algorithm and the concepts of Vitis HLS implementation to design our Alveo U50 based accelerator. The aim is to speed up the junction finding section of the BWFA algorithm in the Alveo U50 and aligning the sequences. The number of reads and parameterised junction finding functions were chosen to increase parallelism of finding junctions.

3.2 Longest Common Subsequence

An alignment between two sequences, v and w, is represented as a two-row matrix where the top row contains the characters of v in order, and the bottom row contains the characters of w in order. Spaces (or gap symbols) can be inserted into either row to allow alignment, with the restriction that two spaces cannot be placed in the same column. Figure 3.1 illustrates an example alignment of the sequences "ATGTTATA" and "ATCGTCC". This alignment models a possible way in which sequence v could have transformed into sequence w. The positions where characters from both sequences match define a common subsequence—a sequence that appears in both v and w in the same order, though not necessarily in contiguous positions. In the alignment shown, "ATGT" appears as a common subsequence of both "ATGTTATA" and "ATCGTCC". Finding an alignment that maximizes the number of matches between the two sequences directly corresponds to identifying their longest common subsequence (LCS). It is important to note that there may be multiple different LCS for a given pair of strings.

Figure 3.1: Alignment of Sequences

At each stage of the alignment process, we decide whether to remove one symbol from either sequence or one symbol from both sequences, starting from the left of the remaining unaligned portions. If we remove one symbol from each sequence simultaneously, we align them together in the current alignment. If we remove a symbol from only one sequence, it is aligned with a gap (space) in the other sequence. In the alignment visualization, different types of symbol pairings are color-coded as the matched symbols are highlighted in red and contribute a score of 1 whereas, mismatched symbols are shown in purple and symbols aligned with gaps are shown in blue if they come from the first sequence, and in green if they come from the second.

3.3 The Manhattan Tourist Problem

Imagine a tourist exploring Midtown Manhattan who wants to visit as many attractions as possible while traveling from the corner of 59^{th} Street and 8^{th} Avenue to the corner of 42^{nd} Street and 3^{rd} Avenue, as illustrated in Figure 3.2(a). Due to limited time, the tourist can only move south ($\downarrow$) or east ($\rightarrow$) at each intersection. There are numerous possible routes across the map, but no single path passes by every attraction. The goal is to determine a valid route that allows the tourist to see the maximum number of sights. This challenge is known as the Manhattan Tourist Problem [18].

We can represent the layout of Manhattan as a directed graph, known as the Manhattan Graph, where each intersection is modeled as a node, and each city block between intersections is a directed edge that reflects the allowed direction of travel—either south ($\downarrow$) or east ($\rightarrow$), as shown in Figure 3.2(b). Every directed edge is assigned a weight corresponding to the number of attractions located along that block. The journey begins at the source node (marked in blue) at coordinates (0, 0), and ends at the sink node (marked in red) at coordinates (n, m). The total number of attractions visited along any given path is determined by summing the weights of the edges in that path. Thus, solving the Manhattan Tourist Problem involves identifying the path from the source to the sink that has the maximum total weight—also known as the longest path in the Manhattan Graph. Attempting to solve this problem using brute force is not feasible due to the enormous number of possible paths. A more intuitive greedy approach would involve choosing, at each intersection, the direction (south or east) that immediately offers more attractions in the next block. However, this strategy may lead to suboptimal results, as it doesnot account for better opportunities that may lie further along alternative paths.

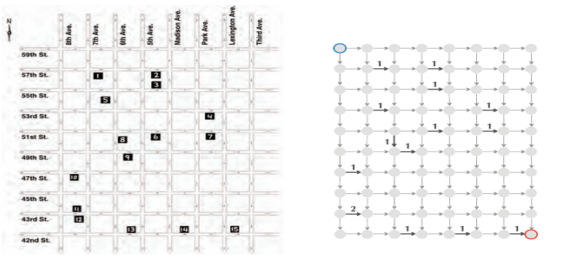


Figure 3.2: Map of Midtown Manhattan

Manhattan Tourist Problem:
Find a longest path in a rectangular city.

Input: A weighted $n \times m$ rectangular grid with $n+1$ rows and $m+1$ columns.
Output: A longest path from source $(0,0)$ to sink (n,m) in the grid.

Figure 3.3: Description of Manhattan Tourist Problem

3.3.1 Relation between Sequence Alignment and Manhattan Problem

In Figure 3.4, two arrays of integers are added alongside the alignment of the sequences "ATGTTATA" and "ATCGTCC". The first array, [0 1 2 2 3 4 5 6 7 8], indicates how many characters from "ATGTTATA" have been used up to each position in the alignment. Similarly, the second array, [0 1 2 3 4 5 5 6 6 7], tracks how many characters from "ATCGTCC" have been consumed at each alignment position. Additionally, a third array is included to represent the type of operation at each column of the alignment using a diagonal arrow (↘) indicating a match or mismatch between characters from both sequences. A right arrow (→) which signifies an insertion, where a gap is added in the second sequence and a down arrow (↓) indicates a deletion, where a gap is added in the first sequence.

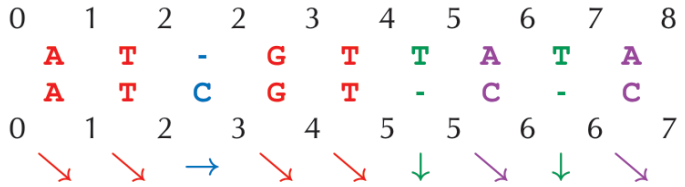


Figure 3.4: An alignment of "ATGTTATA" and "ATCGTCC".

In addition to the horizontal and vertical edges in the Manhattan Problem, Figure 3.5 introduces diagonal edges that connect the points (i, j) to (i + 1, j + 1). The Directed Acyclic Graph (DAG) depicted in Figure 3.5 is called the alignment graph of the strings v and w, denoted as Alignment(v,w). A path from the source node to the sink node within this graph is referred to as an alignment path. Each possible alignment of v and w can be interpreted as a set of steps that constructs a specific alignment path within Alignment(v,w). For example, the path (0, 0) ↘ (1, 1) ↘ (2, 2) → (2, 3) ↘ (3, 4) ↘ (4, 5) ↓ (5, 5) ↘ (6, 6) ↓ (7, 6) ↘ (8, 7) shown in Figure 3.5(a) represents the alignment between "ATGTTATA" and "ATCGTCC" from Figure 3.4. Another example of a random path in the alignment graph is shown in Figure 3.5(b).

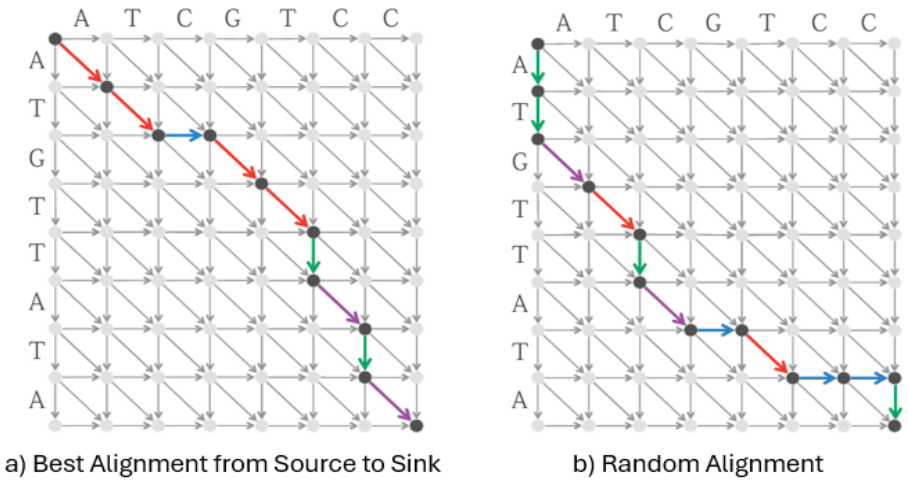


Figure 3.5: Directed Acyclic Graph (DAG)

Identifying the longest common subsequence (LCS) between two strings is essentially the same as finding an alignment that includes the greatest number of matching characters. In Figure 3.6, all diagonal edges in Alignment("ATGTTATA", "ATCGTCC") that represent character matches have been highlighted. If we assign a weight of 1 to each of these matching diagonal edges and a weight of 0 to all other edges (insertions and deletions), then solving the LCS problem becomes equivalent to finding the longest path through this weighted directed acyclic graph (DAG). To tackle this, we must develop an algorithm capable of finding the longest path in a DAG. This is where dynamic programming comes in a powerful method widely used to efficiently solve complex problems across many areas of science and engineering.

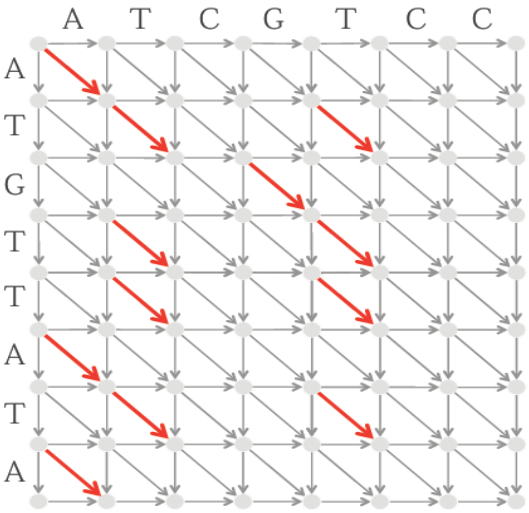


Figure 3.6: Alignment Graph with edges

The pseudo code below calculates the length of the longest path to a given node (i,j) within a grid, based on the key insight from the Manhattan Tourist Problem: a node (i,j) can only

be reached either by moving south (↓) from (i-1,j) or east (→) from (i,j-1). To determine the longest path from the starting point at (0,0) to the destination at (n,m), we compute the longest path lengths from the source to every intermediate node (i,j), gradually expanding outward from the origin. Although it may initially appear that this approach increases the workload—requiring us to solve n x m subproblems rather than just one, dynamic programming provides a smart solution. It ensures that each subproblem is solved only once, avoiding the inefficiency of repeated calculations and thus saving significant computational effort.

From now on, we will represent the length of the longest path from the source (0,0) to any node (i,j) as $S_{i,j}$. Calculating the values of $S_{0,j}$ for $0 \leq j \leq m$ is straightforward because the only way to reach any node along the top row is by continuously moving to the right (→), leaving no alternative paths. Therefore, $S_{0,j}$ is simply the sum of the weights of the first j horizontal edges starting from the origin. Likewise, for nodes along the leftmost column, $S_{i,0}$ equals the sum of the weights of the first i vertical edges from the starting point. For all other nodes where both $i > 0$ and $j > 0$ the only valid ways to reach (i,j) are either by moving downward from (i-1,j) or moving rightward from (i,j-1) [18]. Thus, $S_{i,j}$ can be computed as the maximum of two values:

$$S_{i,j} = \max \begin{cases} S_{i-1,j} + \text{weight of vertical edge from } (i-1,j) \text{ to } (i,j) \\ S_{i,j-1} + \text{weight of horizontal edge from } (i,j-1) \text{ to } (i,j) \end{cases}$$

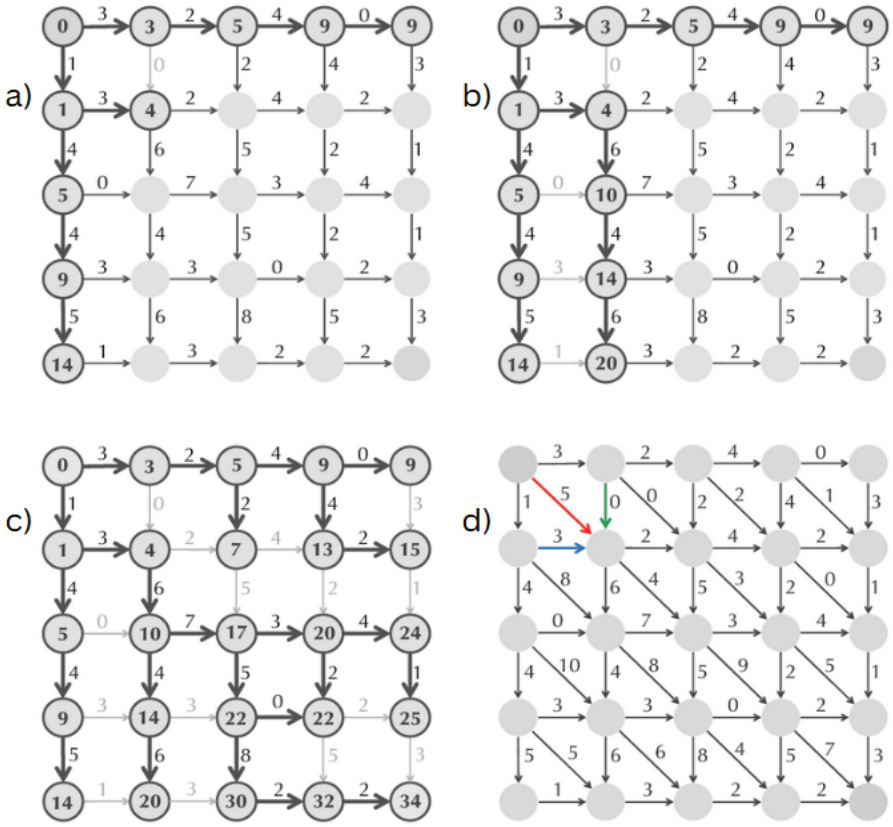


Figure 3.7: Calculation of Maximum Length Path to go from Source to Sink.

In Figure 3.7 (a) we can observe that computing $S_{1,1}$ uses the horizontal edge → from (1,0),

which is highlighted. Node(1,1) has three predecessors((0,0), (0,1),and(1,0)) that are used in the computation of $S_{1,1}$. In Figure 3.7 (b) the computation of all values of $S_{i,1}$ in column 1 can be observed. In Figure 3.7 (c) the graph has calculated all scores of $S_{i,j}$. In Figure 3.7 (d) we can see a graph with diagonal edges constructed for an imaginary city.

```
MANHATTANTOURIST(n, m, Down, Right)
    s_{0,0} ← 0
    for i ← 1 to n
        s_{i,0} ← s_{i-1,0} + down_{i,0}
    for j ← 1 to m
        s_{0,j} ← s_{0,j-1} + right_{0,j}
    for i ← 1 to n
        for j ← 1 to m
            s_{i,j} ← max{s_{i-1,j} + down_{i,j}, s_{i,j-1} + right_{i,j}}
    return s_{n,m}
```

Figure 3.8: Pseudo Code for Manhattan Tourist Problem with Down and Right Edges.

The pseudocode is shown Figure 3.8 where, $down_{i,j}$ and $right_{i,j}$ are the respective weights of the vertical and horizontal edges entering node (i, j). We denote the matrices holding ($down_{i,j}$) and ($right_{i,j}$) as Down and Right, respectively [18].

3.3.2 Sequence Alignment as a graph

After understanding how dynamic programming addressed the Manhattan Tourist Problem, we can modify the Manhattan Tourist approach to handle alignment graphs with diagonal edges. In a Directed Acyclic Graph (DAG), let S_b represent the length of the longest path from the source to node b. A node a is considered a predecessor of node b if there is a directed edge from a to b in the DAG. It is important to note that the indegree of a node corresponds to the number of its predecessors [18]. The **score** S_b of node b with indegree k is computed as a maximum of k terms:

Let S_b denote the length of the longest path from the source to node b. A node a is called a predecessor of node b if there exists a directed edge from a to b in the DAG. The indegree of node b equals the number of its predecessors [18]. The **score** S_b for a node b with indegree k is calculated as the maximum of k values:

$$S_b = \max_{\text{all predecessors } a \text{ of node } b} \{S_a + \text{weight of edge from } a \text{ to } b\}.$$

For instance, in the graph presented in Figure 3.7 (d), node (1,1) has three predecessors. We can reach (1,1) by moving right from (1,0), down from (0,1), or diagonally from (0,0). Assuming that we have already calculated the values for $S_{0,0}$, $S_{0,1}$, and $S_{1,0}$, we can now determine $S_{1,1}$ by selecting the maximum value from these three options.

$$S_{1,1} = \max \begin{cases} S_{0,1} + \text{edge weight} \downarrow \text{ which connect } (0,1) \text{ to } (1,1) = 3 \\ S_{1,0} + \text{edge weight} \rightarrow \text{ which connect } (1,0) \text{ to } (1,1) = 4 \\ S_{0,0} + \text{edge weight} \searrow \text{ which connect } (0,0) \text{ to } (1,1) = 5 \end{cases}$$

To calculate the score at any node (i,j) in the alignment graph, the following recurrence relation is applied:

$$S_{i,j} = \max \begin{cases} S_{i-1,j} + \text{ edge weight} \downarrow \text{ which connects } (i-1,j) \text{ to } (i,j) \\ S_{i,j-1} + \text{ edge weight} \rightarrow \text{ which connects } (i,j-1) \text{ to } (i,j) \\ S_{i-1,j-1} + \text{ edge weight} \searrow \text{ which connects } (i-1,j-1) \text{ to } (i,j) \end{cases}$$

To determine the longest path in a general Directed Acyclic Graph (DAG), the first step is to establish a topological ordering—an arrangement where each node appears only after all of its predecessors. In such an ordering, any directed edge (a_i, a_j) ensures that node a_i comes before node a_j, i.e., i < j. The Manhattan Tourist problem works effectively for finding the longest path in a grid because the structure of the algorithm naturally respects this topological order. By processing nodes sequentially—either by rows or columns—the algorithm aligns with the inherent dependency constraints of a DAG embedded in a grid layout.

After establishing a topological order, the longest path from the source to the sink can be computed by traversing the DAG nodes in that specific order. Because each edge contributes to only a single recurrence relation, the overall time complexity for finding the longest path is directly proportional to the number of edges in the DAG.

3.4 Space-Efficient Sequence Alignment

To illustrate the concept of fitting alignments, consider the task of aligning a large NRP synthetase protein from Bacillus brevis (20,000 amino acids long) with a much shorter A-domain from Streptomyces roseosporus (600 amino acids long). Performing such an alignment directly on a computer can be impractical due to the high memory demands required to store the dynamic programming (DP) matrix. The computational time needed for aligning two sequences of lengths n and m is proportional to the number of edges in the alignment graph, which is $\mathcal{O}(n \cdot m)$. Similarly, the memory requirement is also $\mathcal{O}(n \cdot m)$, primarily due to the need to retain backtracking information [18]. However, it is possible to reduce the memory usage to $\mathcal{O}(n)$by sacrificing some speed, effectively doubling the runtime while maintaining the same overall time complexity of $\mathcal{O}(n \cdot m)$.

A **divide-and-conquer** strategy is effective when a complex problem can be solved by breaking it down into smaller, more manageable subproblems. This approach involves two main stages: in the divide phase, the original problem is split into smaller instances, and each is solved independently; in the conquer phase, these individual solutions are combined to form a solution to the original problem [18]. When the goal is to calculate only the alignment score—without

constructing the full alignment—the memory usage can be significantly reduced. In this case, we only need to store values for two columns of the alignment graph at a time, leading to a space complexity of $\mathcal{O}(n)$.

This optimization is based on the insight that, to compute the alignment scores in column j, we only need the values from the previous column, *j-1*. As a result, any alignment scores from columns earlier than *j-1* become unnecessary and can be discarded during the computation of column *j*. This concept is visually represented in Figure 3.9.

Finding the longest path typically involves maintaining a full matrix of backtracking pointers, which leads to a space complexity that grows quadratically. However, the space-efficient approach avoids storing these pointers by trading memory usage for a modest increase in computation time.

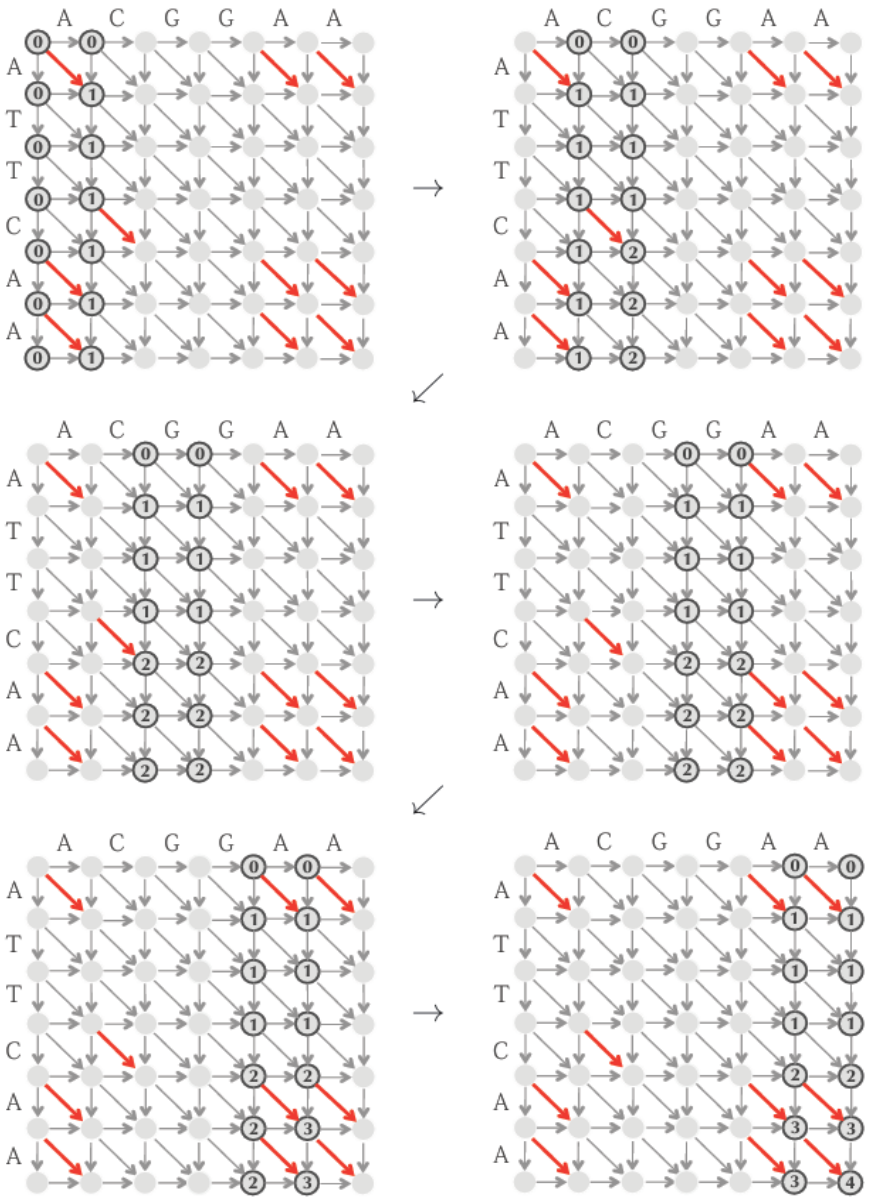


Figure 3.9: Computing an alignment score using only two columns

3.4.1 Middle Node of the Alignment

Given strings $v = v_1, v_2, ..., v_n$ and $w = w_1, w_2, ..., w_m$, define middle = $\lfloor m / 2 \rfloor$. The middle column of *AlignmentGraph*(v, w) is the column containing all nodes (i, middle) for $0 \leq i \leq n$. Any longest path from the source to the sink in the alignment graph is guaranteed to pass through the middle column at some point. Our initial goal is to identify this crossing point—called the middle node—using only $\mathcal{O}(n)$ memory. Keep in mind that various longest paths might intersect the middle column at different nodes, and a single path could contain multiple such middle nodes [18].

In Figure 3.10 (a), the midpoint is calculated as 3 ($\lfloor m/2 \rfloor$), and the alignment path intersects the middle column at the unique middle node (4, 3), which holds the highest score in that column namely, 4. Let *FromSource(i)* represent the length of the longest path from the starting point (0, 0) to the point (i, middle), and let *ToSink(i)* represent the length of the longest path from (i, middle) to the endpoint (n, m). Certainly,

$$Length(i) = FromSource(i) + ToSink(i)$$

and so we need to compute *FromSource(i)* and *ToSink(i)* for each i.

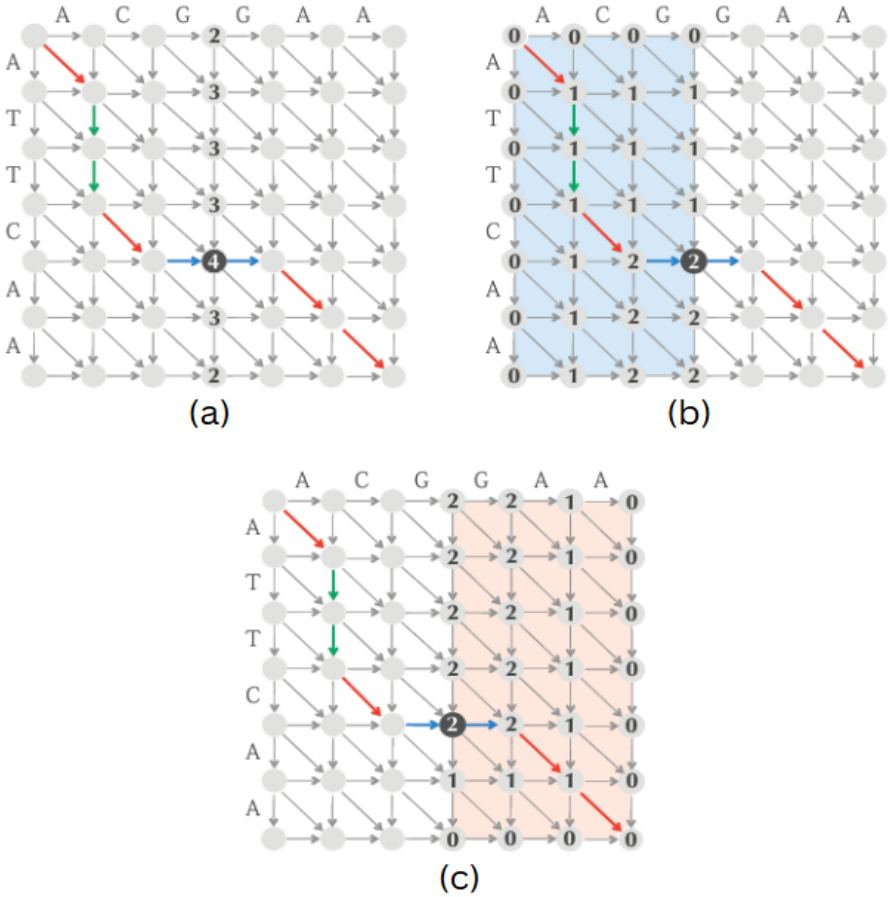


Figure 3.10: Finding Middle Node based on two scores i.e., *FromSource(i)* and *ToSink(i)*

Figure 3.10 (a) illustrates the alignment graph for the sequences "ATTCAA" and "ACGGAA", with the optimal alignment path highlighted. Each node in the middle column, identified by coordinates (i, middle), displays a value *Length(i)*, calculated using the previously described method. The node with the highest *Length(i)* value is selected as the middle node—though multiple nodes may share this distinction. In this figure, the chosen middle node is marked in black [18].

In Figure 3.10 (b), the values of *FromSource(i)* for all i can be computed using $\mathcal{O}$(n) space and $\mathcal{O}(\frac{n \cdot m}{2})$ time. Specifically, each *FromSource(i)* corresponds to the score $S_{i,middle}$, which can be obtained using a linear-space dynamic programming approach [18]. To compute these scores, the algorithm performs a forward traversal from the 0th column to the middle column of the alignment graph, effectively covering half the graph's structure. As a result, the overall computation touches approximately half of the graph's edges, leading to a time complexity proportional to half the graph's area, or n.m / 2.

In Figure 3.10 (c), the values of *ToSink(i)* for all i can also be computed using $\mathcal{O}$(n) space and $\mathcal{O}(\frac{n \cdot m}{2})$ time. This step involves determining the longest paths from the sink node (n,m) back to (i,middle), which is equivalent to reversing the direction of all edges in the alignment graph and treating the sink as the new source. Instead of explicitly reversing the graph, we can reverse the input sequences $v = v_1, v_2, ..., v_n$ and $w = w_1, w_2, ..., w_m$, and then compute the scores $S_{n\text{-}i,m\text{-}\,middle}$using the reversed sequences. This makes computing *ToSink(i)* conceptually and computationally similar to *FromSource(i)*, and the process again requires only linear space and a time complexity proportional to half the alignment graph's area, or n.m / 2.

In total, we can calculate all values Length(i) = FromSource(i) + ToSink(i) using linear space, with a runtime proportional to $\frac{n \cdot m}{2} + \frac{n \cdot m}{2} = n \cdot m$, which corresponds to the total area of the alignment graph. It may seem that we've spent a considerable amount of time just to identify a single node in the alignment path. One might think that this approach is ineffective, having consumed $\mathcal{O}(n \cdot m)$ time (the entire area of the graph) while yielding minimal information [18].

3.4.2 The Fast and Memory-Efficient Alignment Algorithm

After identifying a middle node, we can immediately determine two regions that the longest path must traverse, one on each side of the node. As illustrated in Figure 3.11, one region includes all the nodes above and to the left of the middle node, while the other covers all the nodes below and to the right. Consequently, the combined area of these two regions is half the total area of the alignment graph [18].

We can now break down the task of finding the longest path from (0,0) to (n,m) into two subproblems: determining the longest path from (0,0) to the middle node, and finding the longest path from the middle node to (n,m). The next step is to identify the two middle nodes within the smaller regions, a process that takes time proportional to the sum of the areas of these regions, which is $\frac{n \cdot m}{2}$, as shown in Figure 3.11 (b). At this point, we have already identified three nodes of the optimal path. In the following iteration, we will apply the divide-and-conquer approach again to locate four middle nodes within the even smaller blue regions, which collectively have an area of $\frac{n \cdot m}{4}$, as shown in Figure 3.12. This brings us much closer to reconstructing the entire optimal alignment path!

In the Fig 3.11 (a), A middle node creates two highlighted rectangles, showing that any optimal path passing through this node must travel within these regions. As a result, we can disregard the other parts of the alignment graph when searching for the optimal alignment path.

In the Fig 3.11 (b), After identifying the middle nodes (represented as two additional black circles) within the previously defined rectangles, we can continue determining the optimal alignment. Once the middle edge is found, we can again identify two smaller rectangles that the longest path must pass through. These new rectangles occupy less than half of the alignment graph's total area, as shown in Figure 3.11, offering a more efficient approach..

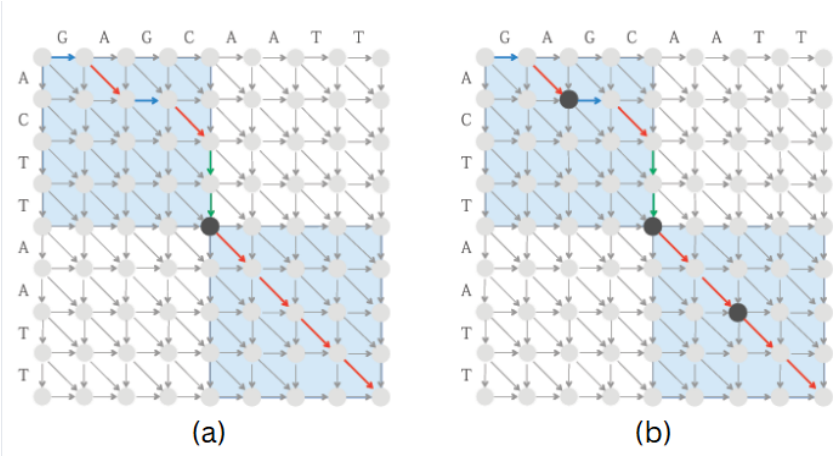


Figure 3.11: Divide and Conquer Approach to find the Optimal Alignment

In general, at each stage before the final one, the number of middle nodes we find doubles, while the time required to find each set of middle nodes is halved. By continuing this process, we can identify the middle nodes for all rectangles and ultimately construct the complete alignment in a total time of,

$$n \cdot m + \frac{n \cdot m}{2} + \frac{n \cdot m}{4} + \cdots < 2 \cdot n \cdot m = \mathcal{O}(n \cdot m).$$

Thus, we have arrived at a linear-time alignment algorithm that requires only linear space [18].

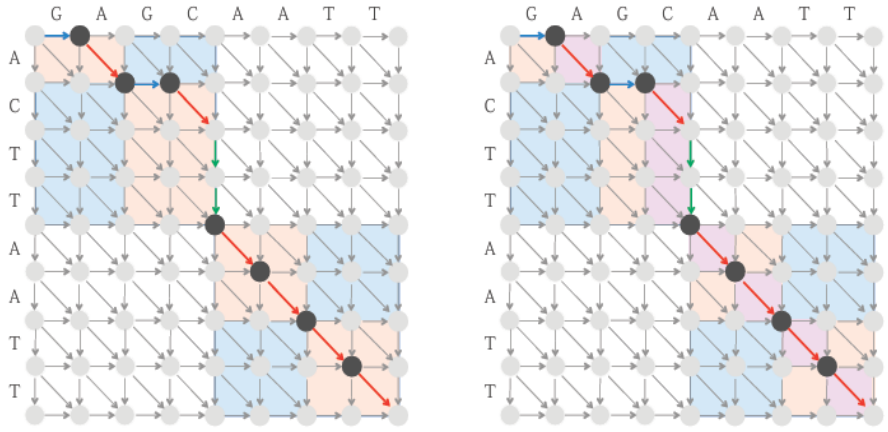


Figure 3.12: Finding middle nodes within previously identified blue rectangles.

Figure 3.13 presents the pseudocode for the *LinearSpaceAlignment* algorithm, which recursively determines the longest path in the alignment graph for a given substring v_{top+1} through v_{bottom} of string v, and a substring w_{left+1} through w_{right} of string w. The algorithm makes use of the function *MiddleNode(top, bottom, left, right)* to identify the coordinate i of the midpoint node (i, j) within the alignment graph for the specified substrings. It also calls *MiddleEdge(top, bottom, left, right)*, which determines whether the middle edge is horizontal, vertical, or diagonal, denoted respectively by "$\rightarrow$ ", "$\downarrow$ ", or "$\searrow$ ". To construct the full alignment between v and w, the algorithm is initiated with *LinearSpaceAlignment(0, n, 0, m)*. When *left = right*, it implies alignment of an empty substring of w with a segment of v, which can be straightforwardly computed as a sequence of vertical gap penalties [18].

```
LINEARSPACEALIGNMENT(top, bottom, left, right)
    if left = right
        return alignment formed by bottom − top vertical edges
    if top = bottom
        return alignment formed by right − left horizontal edges
    middle ← ⌊(left + right) / 2⌋
    midNode ← MIDDLENODE(top, bottom, left, right)
    midEdge ← MIDDLEEDGE(top, bottom, left, right)
    LINEARSPACEALIGNMENT(top, midNode, left, middle)
    output midEdge
    if midEdge = "→" or midEdge = "↘"
        middle ← middle + 1
    if midEdge = "↓" or midEdge = "↘"
        midNode ← midNode + 1
    LINEARSPACEALIGNMENT(midNode, bottom, middle, right)
```

Figure 3.13: Pseudo Code for Linear Space Alignment

3.5 Vitis HLS Implementation

This section covers the key aspects of implementing any C/C++ code using Vitis HLS, which includes clock frequency settings, reset configuration, and the use of pragmas for optimization. Software designed for CPUs and that written for FPGAs differ fundamentally. Creating code that runs efficiently and portably on both platforms typically involves trade-offs that reduce overall performance [23]. A software which is well optimized for a CPU cannot be used directly for an FPGA, at least not before it is thoroughly checked and all the non-synthesizable parts of the code are replaced with the synthesizable counterparts. The Software written for CPU runs sequentially one step at a time, whereas the software written for FPGA's will exploit the parallelism and run more efficiently on the FPGA.

3.5.1 Clock and Reset Specifications

- **Clock Frequency Specification**
  In the Vitis HLS implementation, the clock period is a critical parameter that determines the timing characteristics of the synthesized hardware. The clock period is specified in nanoseconds (ns) or MHz when the HLS components are created. The default clock period and default clock uncertainty settings can be set in Vitis HLS tool.

- **Reset Configuration**
  The reset configuration in Vitis HLS allows for precise control over how the hardware can be initialized and reset. The reset settings include the following:

  - **Polarity of the Reset:** The ability to specify whether the reset signal is active high or active low.
  - **Synchronous or Asynchronous Reset:** Determining if the reset signal is synchronized with the clock or operates independently.
  - **Registers to be Reset:** Identifying the registers that must be initialized when the reset signal is activated to guarantee that the hardware begins operation in a defined and predictable state.

3.5.2 Why Pragmas are used

In Vitis HLS, pragmas serve as crucial directives or hints that empower the synthesis tool to translate high-level C++ code into efficient RTL hardware descriptions. These directives play a pivotal role in several aspects of design optimization:

- **Optimization:** Pragmas enable the application of techniques that enhance the overall efficiency and performance of the hardware design.
- **Latency Reduction:** Specific pragmas can minimize the number of clock cycles needed to execute operations, thereby reducing latency and improving responsiveness.
- **Throughput Performance:** By leveraging appropriate pragmas, we can maximize data processing rates, enhancing the overall throughput of the design.
- **Area and Resource Usage:** Pragmas provide mechanisms to manage and reduce the utilization of FPGA resources, ensuring that the design remains within the specified constraints of the target device.

By effectively utilizing these pragmas, the design can be tailored to meet specific performance and resource goals, ensuring an efficient and high-performance implementation of the intended algorithm.

3.5.3 Different Pragmas Available

For optimization and to guide the tool in synthesizing and implementing hardware designs efficiently, pragmas are used wherever necessary. A detailed description of each pragma that can be used in the synthesis of the hardware is given below.

1. **Pragma HLS Interface:** The interface pragma in Vivado HLS is used to specify the interface properties of the function. It informs the tool about the interaction between the function and the rest of the system, including how data is transferred in and out of the function. The interface pragma or directive is only supported for use on the top-level function, and cannot be used for sub-functions of the HLS component. The syntax for this pragma is as follows:

   ```
   #pragma HLS interface mode = <mode> port = <name>
   ```

   (a) The mode = <mode> attribute within the pragma defines how the port operates, influencing its behavior in the RTL (Register Transfer Level) implementation. These modes encompass:

   - **Port-level protocols:**
     - **ap_none:** Establishes a straightforward data port devoid of additional control signals.
     - **ap_vld:** Integrates a valid signal, signaling when data is ready to read or write.
     - **ap_ack:** Includes an acknowledge signal to confirm successful data read or write operations.
     - **ap_hs:** Incorporates both valid and acknowledge signals, facilitating a bidirectional handshake to manage data validity and acknowledgment.
     - **ap_ovld:** Applies to output ports, featuring a valid signal to indicate readiness for data transmission, useful for output arguments or half of bidirectional interfaces when used with ap_none.
     - **ap_memory:** Configures array arguments akin to a standard RAM interface, with separate ports in Vivado IP Integrator.
     - **bram:** Similar to ap_memory, designed for Vivado IP Integrator with a single port for memory interfaces.
     - **ap_fifo:** Utilizes a standard FIFO (First-In-First-Out) interface with separate data, empty, and full ports, suitable for read or write operations but not bidirectional data flow.
   - **Block-level control protocols:**
     - **ap_ctrl_chain:** Implements control ports for block-level operations, facilitating design initiation, continuation, and signaling of idle, done, and ready states.
     - **ap_ctrl_hs:** Similar to ap_ctrl_chain but excludes the continue operation, useful for simpler control requirements.

- **ap_ctrl_none:** Omits block-level I/O protocols, potentially limiting verification in C/RTL co-simulation.

(b) The attribute port = <name> designates the specific function argument or return value to which the interface pragma is applied. Block-level I/O protocols may also be assigned to the function's return port.

2. **Pragma HLS inline:** The inline pragma in Vitis HLS controls whether a function should be merged into its caller during synthesis, thereby optimizing performance or resource usage. When a function is inlined, its code is incorporated directly into the calling function, eliminating the overhead of function call and return. This approach can lead to more efficient hardware implementation by reducing latency and optimizing for speed. This pragma must be placed in the scope of the function to which inlining options must be applied. The syntax for this pragma is as follows:

```
#pragma HLS inline <recursive | off>
```

Here are the different modes of the INLINE pragma and their implications:

- **INLINE:** This pragma directive, when applied without arguments, instructs the compiler to inline the function into any calling functions. It promotes optimization by consolidating the function's logic directly into its caller.
- **INLINE off:** Specifying INLINE off disables inlining for the function it is applied to. This mode is useful when specific functions should retain their standalone hierarchy rather than being merged into their callers. Disabling inlining preserves function boundaries and can be essential for maintaining modular design practices or when explicit hierarchical organization is preferred.
- **INLINE recursive:** When INLINE recursive is used, it applies the pragma recursively within the function body. This means that any nested functions or blocks within the function will also be inlined where applicable. This approach allows for deeper optimization across multiple levels of function hierarchy.

By default, Vitis HLS determines whether to inline functions based on the hardware size implications of the synthesized design. Small functions are typically inlined for efficiency, while larger functions may remain as separate entities to manage resource utilization effectively. When using the pipeline pragma to introduce pipelining into functions, it is crucial to note that pipelining is dependent on inlining. If a function is not inlined, pipelining that function may not be possible or effective. Therefore, when intending to pipeline functions, it is essential to ensure that inlining is enabled or appropriately managed using INLINE off if pipelining is explicitly desired.

3. **Pragma HLS pipeline:** The pipeline pragma in Vitis HLS is used to optimize and enhance the performance of loops or functions by reducing the initiation interval (II), thereby allowing concurrent execution of operations. This technique is crucial for achieving higher clock speeds and maximizing throughput in hardware designs. The pipeline pragma is placed within the body of a function or loop in the C/C++ source code to specify how operations should be pipelined during synthesis. The syntax is as follows:

```
#pragma HLS pipeline II = <int> off style = <value>
```

- **Initiation Interval (II):** The II = <int> option specifies the desired initiation interval for the pipeline. The II determines how frequently the pipeline processes new inputs. For instance, an II of 1 means that the pipeline processes a new input every clock cycle, optimizing for maximum throughput. However, the actual II achieved may vary depending on data dependencies and resource constraints.
- **Disabling Pipelining:** The off keyword can be used to disable pipelining for specific loops or functions. This is useful when pipelining is undesired for a particular segment of code.
- **Pipeline Styles (style = <value>):**
  - **stp (Stall Pipeline):** This is the default pipeline style used by Vitis HLS. It runs only when input data is available and stalls otherwise. It is suitable when flushable pipelines are not required, ensuring straightforward operation without performance or deadlock issues.
  - **flp (Flushable Pipeline):** This style defines a pipeline that can be flushed, meaning it consumes more resources and may have a larger II. Flushable pipelines are beneficial when there is a need to maintain continuous operation despite occasional data unavailability, at the cost of increased resource utilization.
  - **frp (Free-running, Flushable Pipeline):** This type operates continuously, even without input data, which can enhance timing and boost performance by preventing deadlocks. However, this continuous clocking of pipeline registers may lead to increased power consumption.

4. **Pragma HLS Array Partition:** The array_partition pragma in Vitis HLS is used to partition an array into smaller arrays or individual elements to improve parallel access and performance. By using this pragma, one can reduce the access bottleneck and increase throughput, which is especially beneficial for performance critical components. This pragma partitions the arrays, enabling their synthesis into flip-flops rather than block RAM. Syntax for this pragma is as follows:

```
#pragma HLS array_partition variable= <name> type = <type>
    factor = <int> dim = <int> off = true
```

- **(variable = <name>):** A required argument that specifies the array variable to be partitioned.
- **(factor = <int>):** Defines how many smaller arrays will be generated.
- **(type = <type>):** Vitis HLS supports three types of array partitioning:
  (a) **Cyclic Partitioning:** Cyclic partitioning distributes the elements of an array across smaller arrays in a round-robin fashion. For instance, with a factor of 3:
    - Element 0 goes to the first array.
    - Element 1 goes to the second array.
    - Element 2 goes to the third array.

– Element 3 cycles back to the first array.

Usage: Useful for applications where data access patterns are irregular and benefit from cyclical distribution of elements.

(b) **Block Partitioning:** Block partitioning divides the original array into smaller subarrays made up of consecutive elements.
Usage: Suitable for large arrays where complete partitioning is not feasible due to resource constraints.

(c) **Complete Partitioning:** Complete partitioning splits an array into individual elements, each of which can be accessed independently. This is the default <type>.
Usage: Best for small arrays where you need maximum parallelism.

- **(dim = <int>):** Indicates the specific dimension of a multi-dimensional array that should be partitioned. This is defined by an integer ranging from 0 to N–1, where N represents the total number of dimensions in the array.
  - When a value of 0 is specified, every dimension of the multi-dimensional array is partitioned according to the defined partition type and factor options.
  - Specifying any non-zero value partitions only that particular dimension. For instance, if you specify 1, only the first dimension of the array will be partitioned.
- **(off=true):** Disables the array_partition feature for the specified variable.

5. **Pragma HLS unroll:** In Vitis HLS (High-Level Synthesis), the loop unroll pragma is used to instruct the synthesis tool to unroll loops in the generated hardware design. Loop unrolling is a technique where the iterations of a loop are executed concurrently, reducing loop overhead and potentially improving performance by increasing parallelism. By unrolling, each iteration of the loop is executed in parallel, enabling simultaneous. Syntax for this pragma is as follows:

```
#pragma HLS unroll factor = <N>
```

where factor = <N> specifies how many iterations of the loop should be unrolled. For example, factor=2 would unroll the loop twice, effectively executing two loop iterations in parallel. If factor is not specified, the loop is fully unrolled. With unroll pragma, multiple instances are synthesized and the loop will be executed concurrently. Without using the unroll pragma in HLS, a single instance of the hardware is synthesized and loops will execute iteratively impacting the number of clock cycles utilized for execution.

Considerations:

- **Resource Usage:** Complete loop unrolling can significantly increase resource utilization in the generated hardware. It's important to balance between performance gains and resource constraints.
- **Impact on Timing:** Unrolling loops can potentially improve timing by reducing loop overhead. However, excessive unrolling can lead to longer critical paths, impacting timing negatively.

3.6 Limitations of HLS Design

While HLS offers many optimization features and benefits, it also has some inherent limitations. The limitations which can affect the efficiency and performance of the final hardware design are:

1. **Performance Optimization Complexity:** Achieving optimal performance requires a deep understanding of both the high-level code and the underlying FPGA architecture. Effective use of pragmas and directives is necessary, which can be complex and time-consuming.

2. **Accuracy of Estimations:** Performance and resource usage estimations provided by Vitis HLS might not always be accurate, leading to discrepancies between expected and actual results.

3. **Operator Limitations:** Vitis HLS does not provide a replicate operator or a bitwise negation operator. While these specific operators are missing, other bitwise operators are available as inbuilt functions.

4. **Impact of clock period on number of pipeline stages:** The number of pipeline stages is determined by the clock period specified, limiting flexibility in complex pipelined designs.

   Despite these limitations, Vitis HLS remains a valuable tool for accelerating FPGA design by enabling higher-level abstractions and faster iterations. However, it often requires complementary knowledge of FPGA design and careful optimization to achieve the best results.

Chapter 4

Engineering

4.1 Introduction

This chapter explains in detail about the hardware setup that was used to obtain the alignment of two DNA sequences. Explanations regarding Host Code, Kernel Code and interface that was used for communication between CPU and FPGA, commands used in the command line interface, FPGA-CPU design interface along with the flow of data are included in this chapter.

4.2 What is Hardware Acceleration and Why it Matters

Hardware acceleration involves using specialized hardware components to execute specific computational tasks more efficiently than a general-purpose CPU. This technique boosts performance by offloading certain functions—such as graphics rendering, data processing, or cryptographic operations—to dedicated units like GPUs, FPGAs, or ASICs. In typical computing environments, the CPU handles most tasks by default. However, relying solely on the CPU—especially for compute-intensive or repetitive tasks—can lead to inefficiencies. Hardware acceleration addresses this by distributing the workload to more suitable components. This is particularly beneficial in systems where the CPU may not be powerful enough or when optimal performance and reduced latency are required [24]. In the context of FPGAs, hardware acceleration means using the reconfigurable logic of the FPGA to implement custom data paths or algorithms directly in hardware, instead of running them as software on a CPU as shown in Figure 4.1.

4.2.1 Why we need to enable it?

- Acceleration enables much smoother browsing and media playback.
- When hardware acceleration works as intended, it can significantly boost performance, making applications run faster and smoother. Enabling hardware acceleration lets us fully tap into that hardware's capabilities across all compatible applications.

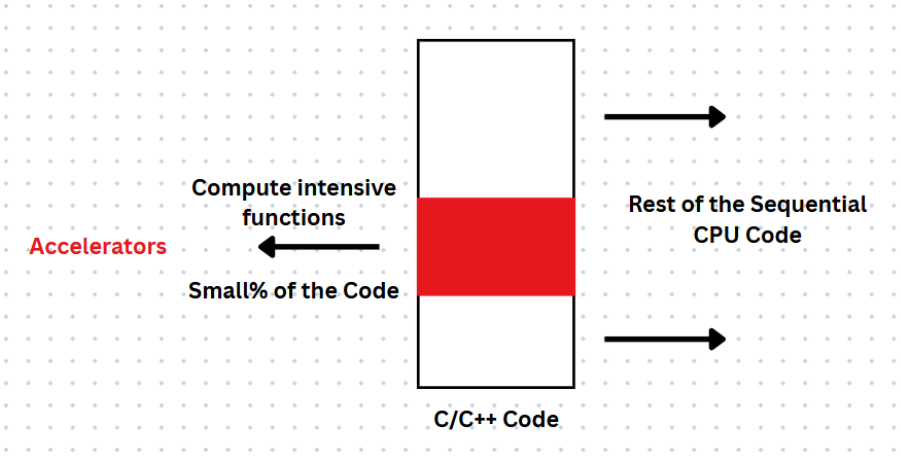


Figure 4.1: Representation of Hardware Acceleration

4.2.2 Why we may need to disable it?

- Unfortunately hardware acceleration sometimes does not works as smoothly as it should.
- If the CPU is powerful enough but the other hardware components are not, enabling hardware acceleration might not improve performance and may need to be turned off.
- Hardware components are prone to overheating and other damages. Intensive use through hardware acceleration may cause these problems.
- A common reason to disable hardware acceleration is when the software isn't effectively utilizing the hardware or runs less reliably compared to running solely on the CPU.

4.2.3 CPU vs FPGA

- A CPU (Central Processing Unit) is a general purpose processor that handles a wide range of tasks in a computer system including running applications, managing the operating system and performing mathematical calculations. CPU are designed to be versatile and can handle many type of tasks, but they are not optimized for any specific type of workload.
- An FPGA is a reconfigurable hardware platform designed to handle complex and parallel workloads by creating custom logic circuits tailored to specific tasks. Unlike CPUs, which follow a fixed instruction set, FPGAs allow developers to design dedicated hardware pipelines that process data in true parallelism. This makes FPGAs especially efficient for workloads like signal processing, machine learning inference, and real-time data streaming. For operations like matrix multiplication, FPGAs can be configured to execute multiple stages in parallel using pipelined and spatial architectures, offering low-latency and high-throughput performance.
- An FPGA is a hardware device that excels at handling specialized computations by allowing the creation of custom data paths and parallel execution units. This contrasts with

a CPU, which is optimized for general-purpose computations and sequential instruction execution. CPUs are the standard processors used in most electronic devices for a wide range of tasks. While an FPGA can significantly outperform a CPU for certain workloads, such as real-time processing or highly parallel tasks, this advantage depends on the nature of the computation. FPGAs are especially well-suited for computations that can be heavily parallelized or require low latency, as their architecture allows for the direct hardware implementation of algorithms rather than executing them as software instructions.

- Parallel computing involves dividing a larger computation into smaller, independent tasks that can be executed at the same time. These tasks are then combined or synchronized to produce the final outcome of the original computation. However, the process of configuring and integrating an FPGA with a system can introduce overhead. If the computational task is relatively simple or infrequent, the time and effort spent on programming and interfacing with the FPGA may outweigh the performance benefits.

4.3 Hardware-Software Partitioning

Assume we have an application to accelerate using FPGA. First, we should find parts of the application that can benefit from the FPGA structure to run faster on FPGA than on the CPU. This process is called **software-hardware partitioning**. Assume an application consisting of four parts. For example, part-3 is a compute intensive task and is slow on CPU, so it is a good idea to implement that on FPGA, which can run much faster. In this way, we improve the entire performance. Traditionally, FPGAs are very good at performing compute intensive tasks, and CPUs are good at memory intensive tasks. However, with recent advances in FPGAs and proposals of a high bandwidth memory connection between FPGA and DDR memories, FPGA can show better performance in some memory intensive tasks as well.

4.3.1 Profiling

Profiling an application and finding the compute-intensive parts is one of the most important steps in improving performance. Application profiling is a practical technique to find bottlenecks. Profiling is a form of dynamic program analysis that involves adding instrumentation to the program's source code or binary executable using a specialized tool known as a profiler. There are several profiling tools available that we can use in the process of partitioning hardware and software. Valgrind, Gperf tools from Google and Gprof the classical gcc profiling tool are some tools used for profiling. Gperf tool is a CPU profiler that software developers use in Google. It's main goal is to measure and compare the execution time of different functions in a software application running on a Linux system. Main goal of this tool is to measure and compare the execution time of different functions in a software application. There are three parts to use it;

1. **Linking:** To include the CPU profiler in our executable, add the flag *-lprofiler* during the linking stage of the build process.

2. **Running:** To enable the profiler, set the environment variable CPUPROFILE to the desired filename where the profiling data will be saved.

3. **Analyzing:** Google's *-pprof* is used to analyze profiling data generated by the profiler.

4.3.2 Profiling our BWFA C++ Code

The step-by-step commands to use gprof to profile our C or C++ program are shown in Figure 4.2 and details of each command is given below:

- *cat bwfa.cpp*: Concatenate and display file contents.
- *g++ -o0 -pg bwfa.cpp*: Compiling the file bwfa.cpp with the flags; **-o0** No optimzation and ensures the profiler gets accurate line or function data. It is great for debugging and profiling. **-pg** Enables profiling with gprof and adds instrumentation for generating a gmon.out file when the program runs.
- *./a.out*: This generates gmon.out file.
- *gprof a.out gmon.out*: Asking gprof to analyze the profiling data gmon.out that was generated when we ran ga.out and displays the performance report.
- *sudo apt install python3-pip* and then *pip3 install gprof2dot*.
- *export PATH="$ HOME/.local/bin:$ PATH"* and
- *echo 'export PATH = "$ HOME /.local/bin:$ PATH"' >> ~/.bashrc* and
- *source ~/.bashrc* are used to add the executable in our $ PATH.
- *gprof a.out gmon.out —gprof2dot— dot -Tpng -o out.png*: This generates the profiling report (gprof a.out gmon.out), converts that into a graph (gprof2dot) and renders it as a png image using Graphviz (dot -Tpng) and then saves the image (-o out.png).
- *gprof a.out gmon.out —gprof2dot -s -W— dot -Gpdi=200 -Tpng -o out.png*: Converts it to dotgraph using edge weights like time or call counts for function calls (gprof2dot -W) and sorts functions by time spent by showing the biggest bottlenecks at the top (-s) and renders a high resolution image (-Gpdi=200).
- *sudo apt install valgrind*
- *valgrind –tool=callgrind ./a.out*: Runs our program under callgrind, a cache and call-graph profiler. It collects data about function calls and CPU usage (but not memory leaks, for that we need to use –tool=memcheck). The output is stored in a file like callgrind.out.<pid>
- *gprof2dot -f callgrind –strip callgrind.out* —dot -Tpng -o out.png*: Tells grpof2dot to parse callgrind format data (-f callgrind) by using all matching callgrind output files in the location (callgrind.out*) and removes long path prefixes and namespaces to clean up function names (–strip).

```
rclab@rclab-System-Product-Name: ~
rclab@rclab-System-Product-Name:~$ g++ -O0 -pg ga.cpp
rclab@rclab-System-Product-Name:~$ ./a.out
Aligned Sequences:
CCGTGG-CCCCGGGC-G-CCTTCCTG--G----G--C--C-T-GAAGG-CG-CT--GCG--ATGG-CTGGAC-CTGTCCCACATGGA
-A-TGGT-----G-CTGTC-T-CCTGCCGACAAGACCAACGTC-AAGGCCGCCTGGG-GTAA-GGTC-GG-CGC-G--C-AC---G-
rclab@rclab-System-Product-Name:~$ gprof a.out gmon.out > profile.txt
rclab@rclab-System-Product-Name:~$ gprof a.out gmon.out | gprof2dot | dot -Tpng -o output.png
rclab@rclab-System-Product-Name:~$ gprof a.out gmon.out | gprof2dot -w  | dot -Tpng -o output.png
rclab@rclab-System-Product-Name:~$ gprof a.out gmon.out | gprof2dot -w  | dot -Gdpi=200 -Tpng -o output.png
rclab@rclab-System-Product-Name:~$ gprof a.out gmon.out | gprof2dot -s -w  | dot -Gdpi=200 -Tpng -o output.png
rclab@rclab-System-Product-Name:~$ valgrind --tool=callgrind ./a.out
==9415== Callgrind, a call-graph generating cache profiler
==9415== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==9415== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==9415== Command: ./a.out
==9415==
==9415== For interactive control, run 'callgrind_control -h'.
Aligned Sequences:
CCGTGG-CCCCGGGC-G-CCTTCCTG--G----G--C--C-T-GAAGG-CG-CT--GCG--ATGG-CTGGAC-CTGTCCCACATGGA
-A-TGGT-----G-CTGTC-T-CCTGCCGACAAGACCAACGTC-AAGGCCGCCTGGG-GTAA-GGTC-GG-CGC-G--C-AC---G-
==9415==
==9415== Events    : Ir
==9415== Collected : 13295737
==9415==
==9415== I   refs:      13,295,737
rclab@rclab-System-Product-Name:~$ gprof2dot -f callgrind callgrind.out.* | dot -Tpng -o output1.png
rclab@rclab-System-Product-Name:~$ gprof2dot -f callgrind --strip callgrind.out.* | dot -Tpng -o output1.png
rclab@rclab-System-Product-Name:~$ gprof2dot -f callgrind callgrind.out.* | \sed -E '
  s/std::[a-zA-Z0-9_:<>]*::/g;   # Remove std:: and nested namespaces
  s/[a-zA-Z0-9_:]+<[^>]*>//g;    # Remove full template arguments
  s/\([^\)]*\)//g;               # Remove argument lists
  s/::/g;                        # Remove remaining scope resolution
' | dot -Gdpi=200 -Tpng -o output2.png
rclab@rclab-System-Product-Name:~$
```

Figure 4.2: Terminal Commands used for profiling the C++ Code

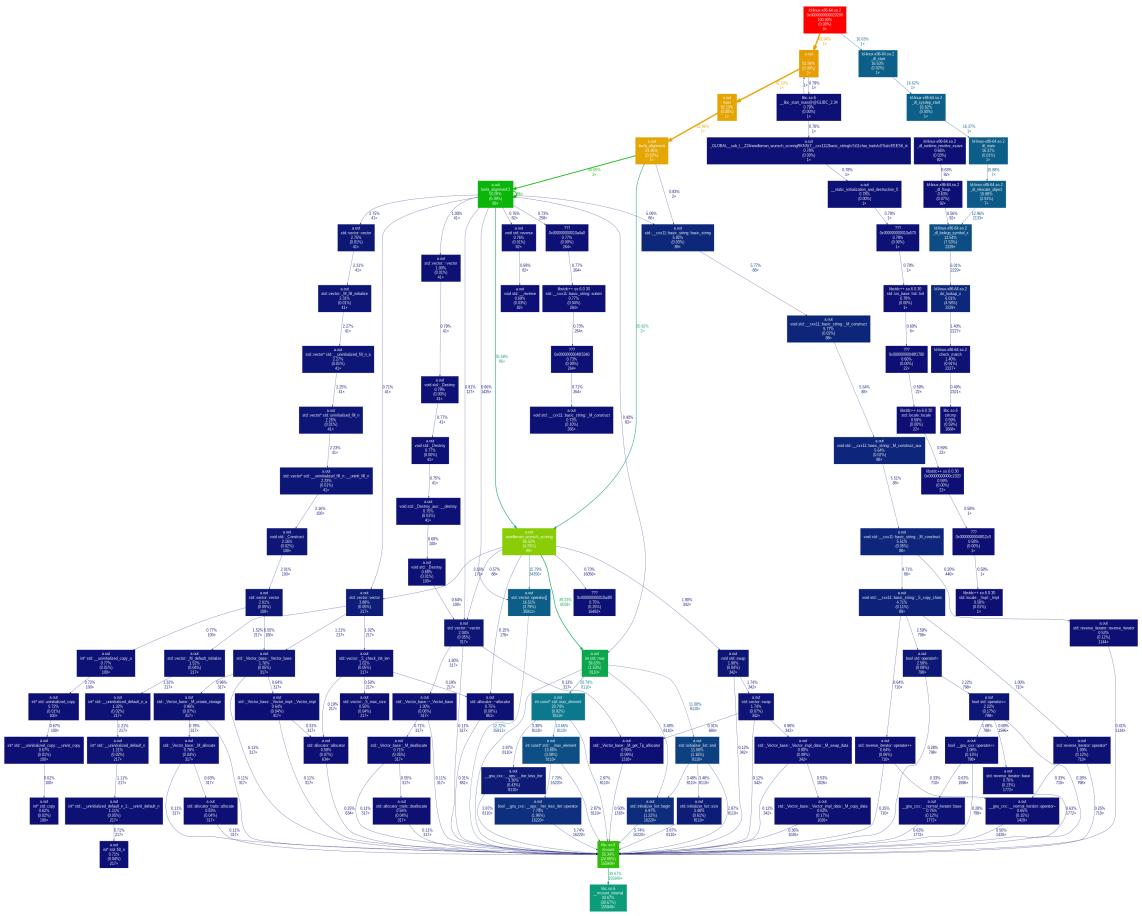


Figure 4.3: Output of Profiling Tool for BWFA C++ Code

4.3.3 Analyzing the output of Profiler

We have got a gprof2dot + Callgrind call graph, rendered via Graphviz as shown in Figure 4.3. To interpret this profiler output for performance debugging, we look into the part of call graph where *bwfa_ alignment'2* is analyzed as shown in Figure 4.4 :

- **50.05%** of the total time (cummulative time including child calls) was spent inside *bwfa_alignment'2* function.
- **0.39%** is the self time (time that is actually spent inside *bwfa_alignment* itself).
- **88x** means this function was called 88 times.
- The edges from *bwfa_ alignment'2* represents the amount of runtime accounted for and the number of calls. Example, the edge with 2.75% and 41x represents that this edge accounted for 2.75% of runtime across 41 calls.

Parent_ function - *bwfa_alignment:* It only runs once, but triggers a lot of downstream work (most notably 50.05% inside *bwfa_alignment'2*) and the arrow labeled 2× pointing from this function to *bwfa_alignment'2* means, if *bwfa_alignment'2* were optimized to take zero time, *bwfa_alignment* could run up to 2× faster.

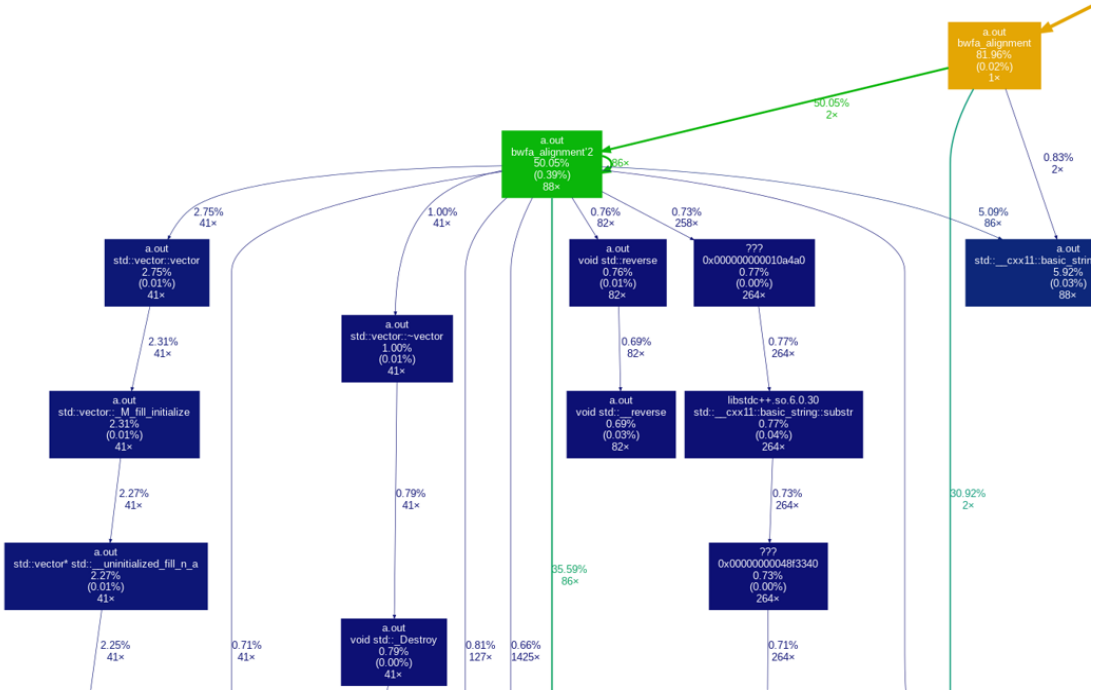


Figure 4.4: Analyzing the part of profiler output

4.4 Function Acceleration on FPGA with Vitis

Firstly, we need a computer with a 64-bit Linux operating system, such as Ubuntu 20.04 LTS or CentOS 7 to run the Xilinx Vitis toolset. The system should have a quad-core Intel i7 (8th generation or newer) or an AMD Ryzen 7 processor, along with a minimum of 16 GB of RAM, although 32 GB is recommended for handling large-scale designs and AI workloads. This computer will be used for design description and synthesis process. The Vitis software installed on the computer can be used to emulate our design without using the actual FPGA hardware. The Emulation can be a software emulation or a hardware emulation. We can also generate the FPGA bitstream to program the actual FPGA board and evaluate the design on the real hardware. Vitis technology targets FPGA hardware such as Alveo data center accelerator cards, Zync Ultrascale and Zync 7000 SoC based embedded system platforms.

- A hardware platform is required to represent the actual target FPGA. This platform is a software and hardware package that describes the target hardware and operating system running on the embedded system. We can create this package for our board, but we use one of the platforms created by xilinx or FPGA board vendors.
- Vitis also includes the Xilinx Runtime (XRT) system, which offers a suite of APIs enabling the host application to communicate with the target platform. XRT manages data transfers and interactions between the host program and the accelerated kernels.
- The Vitis core development kit offers a comprehensive software development toolchain, including compilers and cross-compilers for building host programs and kernels. It also includes analysis tools for profiling and evaluating application performance, along with debugging utilities to help identify and resolve issues within the application.
- Vitis accelerated libraries deliver high-performance FPGA acceleration with minimal modifications to existing code, eliminating the need to completely re-implement algorithms to leverage Xilinx adaptive computing [23]. These libraries offer optimized implementations for commonly used operations in math, statistics, linear algebra, and DSP, as well as for domain-specific applications such as computer vision and image processing.

4.4.1 Program Structure

A general application consists of two parts:

1. The C/C++ top function or host program that runs on the available processor and a couple of accelerators or kernels that run on FPGA. We use the HLS C language to describe the kernels. HLS C language is the C, C++ language and a couple of pragmas and coding styles. The host is responsible for managing the execution of the kernel. The host code is compiled with gcc or g++ and an executable file is generated.
2. The kernel source code runs on accelerator hardware uses high-level synthesis and converts the C/C++ code into the equivalent RTL code. The host program will use vitis runtime system (XRT) APIs to communicate with the kernels as shown in Figure 4.5.

3. Then the vitis project itself combines the FPGA bitstream and executable host program into the target platform.

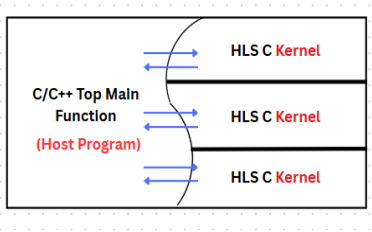


Figure 4.5: General Form of program structure

4.4.2 Vitis Design Flow

We can use the Vitis development environment to describe both the host program and the kernel. The Xilinx Vivado toolset is then used to integrate or link the generated hardware into the underlying hardware platform and generate the FPGA bitstream as shown in Figure 4.6. There are three different building targets;

1. To evaluate our application, we can first perform the **software emulation** process. This process checks the functionality of our application. During this process, vitis starts a Linux emulator, runs the application under the emulator, and reports the result messages in the console view. The primary purpose of software emulation is to verify the functional correctness of both the host application and the kernels. It focuses solely on functional execution, without accounting for timing delays or latency. As a result, it offers no insights into the performance of the accelerator. Software emulation is mainly used for refining algorithms, debugging, addressing functional problems, and allowing developers to rapidly test and enhance their code [23].

2. After software emulation, we can perform **hardware emulation**. The goal of hardware emulation is to check the cycle accuracy of our applications. During this process we can monitor the hardware signal waveforms. Hardware emulation performs an RTL-level simulation of the programmable logic design. Its purpose is to validate the functional correctness of the RTL code generated from the original C/C++ kernel code through synthesis. Hardware emulation is used to test the interaction between different kernels and host code. Using this emulation, we can get the initial performance estimates for the application. In hardware emulation, both compilation and execution take more time compared to software emulation; however, it offers a detailed, cycle-accurate representation of kernel behavior.

3. After finishing the hardware emulation, the **Hardware** step builds a target and compiles the project to create the FPGA bitstream. Compiling a Vitis project for generating the FPGA bitstream is a lengthy process, so before starting, ensure that the design functionality is correct. During the emulation processes, we do not need to use the actual FPGA and these emulations are used for validation and debugging purposes. Compiling for an emulation target is significantly faster than compiling for real hardware.

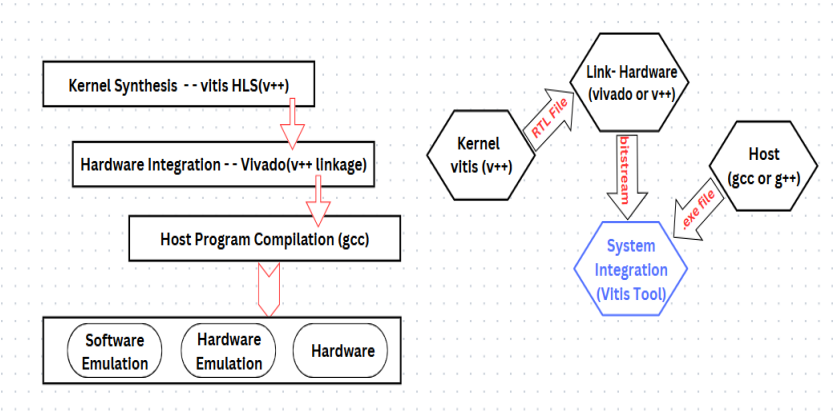


Figure 4.6: Overview of Vitis Design Flow

4.5 Host Code Structure

The kernel is located on the accelerator and uses the AXI protocol to communicate with the memory and other modules. The host program uses the Xilinx runtime system (XRT) to communicate with the kernels and the memory. Therefore, we need a bridge to connect these two bodies together. The host code implemented using the Xilinx Runtime (XRT) library is structured into several stages: setting up the environment, executing core commands, and post-processing the results. This section will provide an overview of each part and its purpose.

4.5.1 Setting up the Environment

The setup of the environment involves loading the required libraries, defining constants for DNA sequence alignment, and preparing the necessary functions for file handling and memory allocation. The key constants used are:

- `MAX_SEQ_LEN`: The maximum length of a sequence chunk.
- `ALIGN_LEN`: The length of aligned sequences after kernel computation.
- `TOTAL_KERNELS`: The number of kernels to be used in parallel for processing.

The XRT library is included to interact with the FPGA. The device and its properties are accessed and initialized in this section. This part of host code is shown in Listing 4.1.

Listing 4.1: Host Code - Setting up the Environment

```
#include <experimental/xrt_device.h>
#include <experimental/xrt_kernel.h>
#include <experimental/xrt_bo.h>

#define MAX_SEQ_LEN 128
#define ALIGN_LEN (MAX_SEQ_LEN * 2)
#define TOTAL_KERNELS 4
```

4.5.2 Core Commands

In the core section of the code, the DNA sequences are read, the FPGA device is initialized, and the sequence alignment is performed using parallel kernel execution. The device is loaded using the XRT API, and buffers are allocated to store the sequence data and aligned results.

- **Loading the FPGA Device:** The FPGA device is loaded using `xrt::device(0)`. The FPGA `xclbin` file is loaded into the device using `load_xclbin`.
- **Reading DNA Sequences:** Two DNA sequences are read from user-provided files, validated to ensure they are not empty, and their lengths are determined.
- **Buffer Allocation:** Memory buffers are created for sequence data and aligned sequences, allowing efficient data transfer between the host and the FPGA.

This part of host code which loads FPGA device and allocates buffers is shown in Listing 4.2.

Listing 4.2: Host Code - Core Commands (Loading FPGA Device and Buffer Allocation)

```
std::cout << "Connecting to FPGA device...\n";
auto device = xrt::device(0);
std::cout << "Device Name: " << device.get_info<xrt::info::device::
   name>() << "\n";
auto uuid = device.load_xclbin(xclbin_file);

// Allocate buffers
auto bo_seq1 = xrt::bo(device, MAX_SEQ_LEN, kernel.group_id(0));
auto bo_seq2 = xrt::bo(device, MAX_SEQ_LEN, kernel.group_id(1));
auto bo_aligned1=xrt::bo(device,ALIGN_LEN, kernel.group_id(7));
auto bo_aligned2=xrt::bo(device,ALIGN_LEN, kernel.group_id(8));

char* ptr_seq1 = bo_seq1.map<char*>();
char* ptr_seq2 = bo_seq2.map<char*>();
char* ptr_aligned1 = bo_aligned1.map<char*>();
char* ptr_aligned2 = bo_aligned2.map<char*>();

std::memset(ptr_seq1, 0, MAX_SEQ_LEN);
std::memset(ptr_seq2, 0, MAX_SEQ_LEN);
std::memset(ptr_aligned1, 0, ALIGN_LEN);
std::memset(ptr_aligned2, 0, ALIGN_LEN);

std::memcpy(ptr_seq1, seq1.c_str(), len1);
std::memcpy(ptr_seq2, seq2.c_str(), len2);

bo_seq1.sync(XCL_BO_SYNC_BO_TO_DEVICE);
bo_seq2.sync(XCL_BO_SYNC_BO_TO_DEVICE);
```

- **Kernel Setup:** Multiple kernels are instantiated for DNA sequence alignment. The kernels are linked to memory buffers, allowing them to perform parallel sequence alignments.
- **Kernel Execution:** Each kernel processes a chunk of the sequence in parallel, with the sequences being divided into smaller parts to maximize parallelism.

This part of host code which sets up and starts execution of the kernel is shown in Listing 4.3.

Listing 4.3: Host Code - Core Commands (Kernel Steup and Execution)

```
// Use CU-specific kernel name for parallel launch
std::string kernel_name = "bwfa_kernel:{bwfa_kernel_" + std::
   to_string(i + 1) + "}";
auto kernel = xrt::kernel(device, uuid, kernel_name);
int len_chunk1 = std::min(MAX_SEQ_LEN, (len1 > start1) ? (len1 -
   start1) : 0);
int len_chunk2 = std::min(MAX_SEQ_LEN, (len2 > start2) ? (len2 -
   start2) : 0);
runs.emplace_back(kernel(
    bos_seq1[i], bos_seq2[i],len_chunk1, len_chunk2, gap_penalty,
       gap_extend, match_score, mismatch_penalty, bos_aligned1[i],
       bos_aligned2[i]
));

// Wait for all kernels to finish processing for this iteration
std::cout << "Processing .......\n";
for (auto& run : runs) {
    run.wait();
}
```

4.5.3 Post-Processing

After the kernel execution, post-processing is performed, which involves:

- **Kernel Execution Timing:** The execution time of each kernel is recorded, and the total time taken by the kernels is computed.
- **Fetching Aligned Sequences:** Once the kernel execution is complete, the aligned sequences are fetched from the FPGA, and any trailing null characters are removed.
- **Displaying Results:** The final aligned sequences are displayed along with the total execution time and the kernel execution time.

This part of host code which fetches the aligned sequences from the kernels and displays the results is shown in Listing 4.4.

Listing 4.4: Host Code - Post-Processing(Fetching and displaying results)

```
// Retrieve the aligned results from all kernels
for (int i = 0; i < runs.size(); ++i) {
    bos_aligned1[i].sync(XCL_BO_SYNC_BO_FROM_DEVICE);
    bos_aligned2[i].sync(XCL_BO_SYNC_BO_FROM_DEVICE);

    char* p1 = bos_aligned1[i].map<char*>();
    char* p2 = bos_aligned2[i].map<char*>();

    final_aligned1 += trim_nulls(p1, ALIGN_LEN);
    final_aligned2 += trim_nulls(p2, ALIGN_LEN);
}
std::cout << " Aligned 1    : " << final_aligned1 << "\n";
std::cout << " Aligned 2    : " << final_aligned2 << "\n";
std::cout << "Execution Time: " << hardware_time << "msec\n";
```

4.5.4 Evolution of the Design: Enhancements and Optimizations

The DNA sequence alignment design evolved through multiple versions, with each new version introducing essential features, optimizations, and improvements in both functionality and user experience. Below is a detailed breakdown of each version and the improvements made at every step:

- **Version 1: Basic Design with Single Kernel for Sequence Alignment**

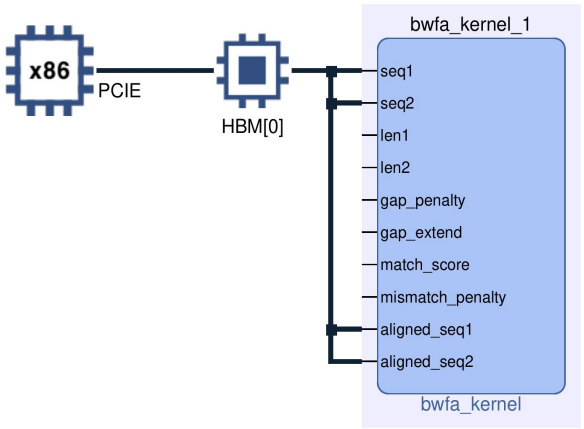


Figure 4.7: Version-1

  - In the initial version as shown in Figure 4.7, a single-kernel with a single High Bandwidth Memory (HBM) was used to perform DNA sequence alignment. While functional, the single-kernel design was not capable of handling large datasets, making the system unsuitable for real-time applications or longer sequences. This de-

sign was primarily focused on core functionality without much consideration for performance optimization or scalability or user experience.

- **Version 2: Enhanced User Experience with Single Kernel**
  - In this iteration, efforts were made to enhance the user experience, including better feedback and error handling. Users were provided with informative messages to help them understand the alignment process and diagnose potential issues.
  - Although the design still used a single kernel, these user-facing improvements set the foundation for future versions focused on both performance and usability.
- **Version 3: File-based Input Handling and Execution Timing**
  - This version introduced the capability to read DNA sequence data from text files using function shown in Listing 4.5, offering greater flexibility and scalability. It allowed users to load large sequence data from external files instead of hard coding sequences directly into the program.
  - A significant enhancement was the addition of execution timing functionality. The system now measured the time taken for different steps of the sequence alignment process, helping identify bottlenecks and providing valuable performance insights for future optimizations.

Listing 4.5: Function to read sequences from a file

```
bool read_sequence_from_file(const std::string& filename, std::
    string& sequence) {
     std::ifstream file(filename);
     if (!file.is_open()) {
         std::cerr << "Error: Could not open the file " <<
             filename << "\n";
         return false;
     }

     // Read the sequence from the file
     std::getline(file, sequence);

     // Close the file
     file.close();

     // Validate sequence
     if (sequence.empty()) {
         std::cerr << "Error: The sequence in file " << filename
             << " is empty.\n";
         return false;
     }

     return true;
}
```

- **Version 4: Parallel Processing with Multiple Kernels and Single HBM**
  - Recognizing the need for faster performance, this version implemented parallel processing by utilizing multiple kernels as shown in Figure 4.8. The design allowed concurrent processing of different segments of the DNA sequences, thereby improving throughput.
  - A single High Bandwidth Memory (HBM) shared among the kernels was introduced, ensuring efficient data transfer between the kernels. This enhanced the speed of DNA sequence alignment and laid the groundwork for handling larger data sets in future versions.

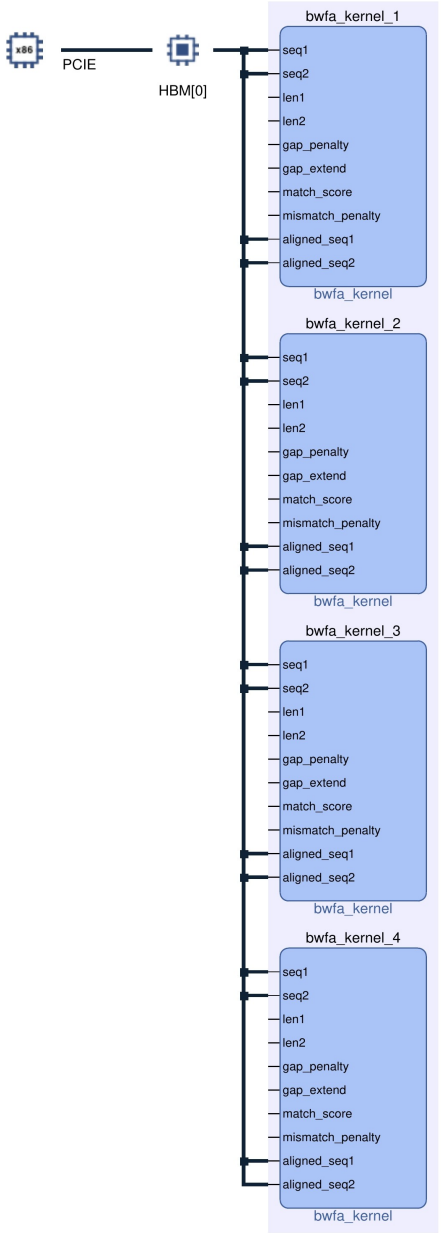


Figure 4.8: Version-4

- **Version 5: Multi-Kernel and Iterative Design for Any Length Sequence**
  - To support sequences of any length, the design was further optimized by adding multiple iterations. Each iteration worked on a distinct segment of the sequence, enabling it to scale better with sequences of varying lengths.
  - The parallel processing capabilities of multiple kernels combined with separate HBM allocations for each kernel as shown in Figure 4.9. This design allowed for faster and more flexible handling of DNA sequences, making the system capable of processing larger and more diverse datasets.

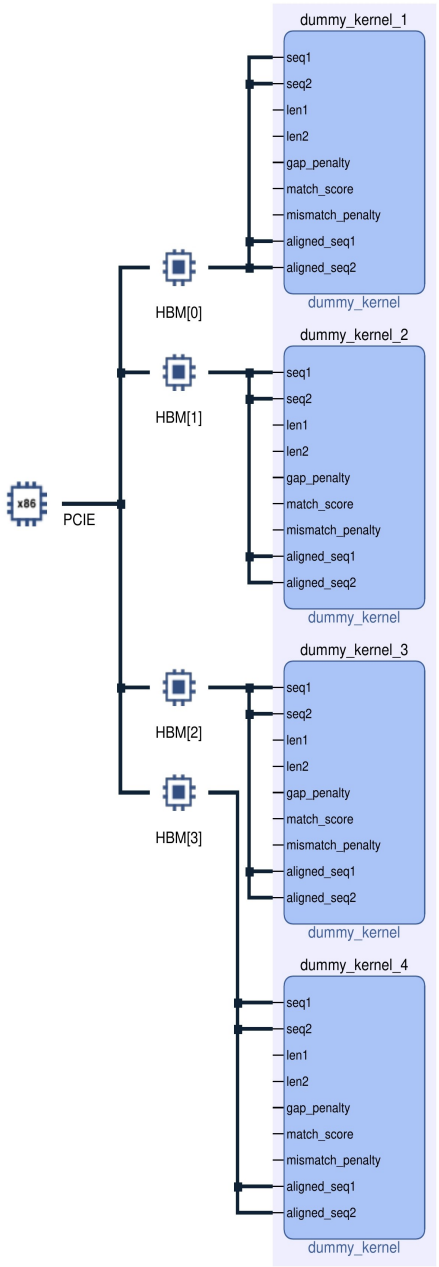


Figure 4.9: Version-5

- **Version 6: Detailed Performance Profiling and Memory Optimization**
  - This version introduced more granular performance profiling. It recorded the execution time for each individual kernel and identified performance bottlenecks at a kernel level.
  - Furthermore, separate HBM allocations were introduced for different inputs as shown in Figure 4.10, allowing for more efficient memory management. This not only optimized memory usage but also improved overall processing speed by minimizing memory contention. Assigning different memory ports to kernel arguments for parallel data access increases the performance.

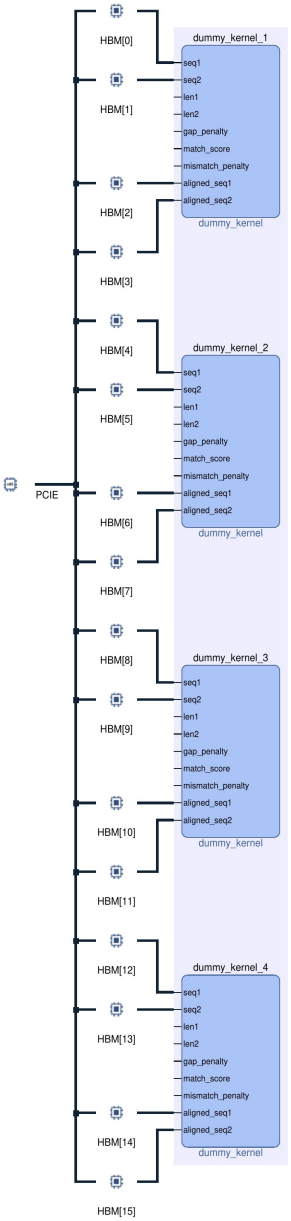


Figure 4.10: Version-6

- **Version 7: High-Performance Optimizations for Speed and Efficiency**
  - In this version optimizations were done through the settings of vitis that drastically improved the speed of DNA sequence alignment. The kernel design was further refined to reduce unnecessary computations, streamline memory access patterns, and fully exploit parallelism.
  - This version achieved significant reductions in execution time, making the design suitable for applications requiring fast DNA sequence alignment, such as real-time bioinformatics analysis.
- **Version 8: Static Memory Allocation for Faster Execution**
  - In Version 8, static memory allocation was implemented as shown in Listing 4.6 to optimize memory usage and reduce the overhead associated with dynamic memory allocation during kernel execution.
  - This modification led to a more efficient memory handling mechanism, further improving execution speed. By ensuring that memory resources were pre-allocated before the execution began, this version minimized latency during the sequence alignment process.

Listing 4.6: Static Buffer Allocation

```
// Allocate buffers statically
std::vector<xrt::bo> bos_seq1(TOTAL_KERNELS), bos_seq2(
   TOTAL_KERNELS), bos_aligned1(TOTAL_KERNELS), bos_aligned2
   (TOTAL_KERNELS);
std::vector<std::unique_ptr<char[]>> maps_seq1(TOTAL_KERNELS
   ), maps_seq2(TOTAL_KERNELS);
std::vector<xrt::kernel> kernels(TOTAL_KERNELS);

for (int i = 0; i < TOTAL_KERNELS; ++i) {
    std::string kernel_name = "bwfa_kernel:{bwfa_kernel_" +
       std::to_string(i + 1) + "}";
    kernels[i] = xrt::kernel(device, uuid, kernel_name);

    bos_seq1[i] = xrt::bo(device, MAX_SEQ_LEN, kernels[i].
       group_id(0));
    bos_seq2[i] = xrt::bo(device, MAX_SEQ_LEN, kernels[i].
       group_id(1));
    bos_aligned1[i] = xrt::bo(device, ALIGN_LEN, kernels[i].
       group_id(8));
    bos_aligned2[i] = xrt::bo(device, ALIGN_LEN, kernels[i].
       group_id(9));

    maps_seq1[i] = std::make_unique<char[]>(MAX_SEQ_LEN);
    maps_seq2[i] = std::make_unique<char[]>(MAX_SEQ_LEN);
}
```

- **Version 9: Full Integration with GUI for User-Friendly Experience and Real-Time Timings**
  - The final version of the design integrated the entire DNA sequence alignment process with a graphical user interface (GUI) shown in Figure 4.11 for any input sequence length, which provided users with a more intuitive experience.
  - In addition to improving usability, the GUI let the users to select the files via file system window and an option to log the alignment process in to a separate log file, it also has clear command window option and the edit command window option.
  - This version represented the culmination of all previous efforts, combining high-performance computing with user-friendly interaction.

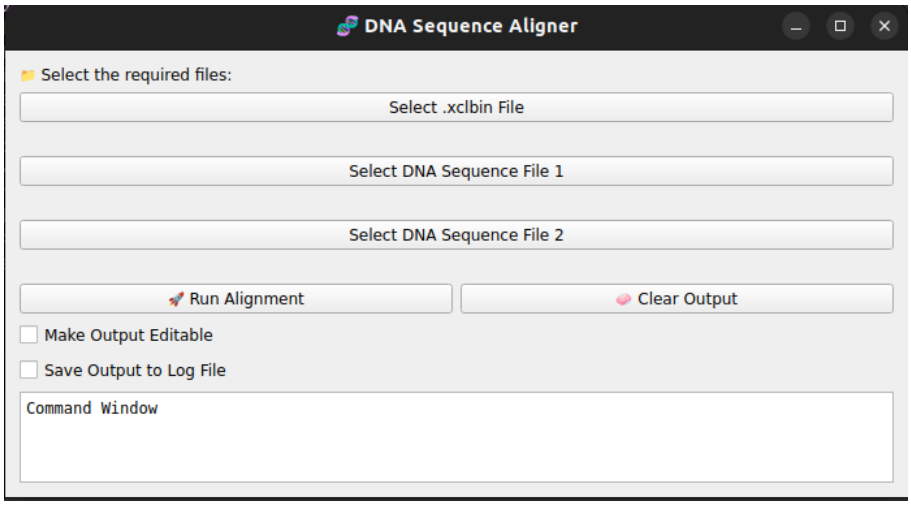


Figure 4.11: Version-9

4.6 PCIe Interface

The PCIe (Peripheral Component Interconnect Express) interface is a high-speed, serial computer expansion bus that establishes point-to-point connections between the CPU and peripheral devices. It utilizes a low-voltage, serial communication protocol that enables simultaneous data exchange via two unidirectional paths. Being a layered and packet-based protocol, PCIe is organized into three layers: the transaction layer, data link layer, and physical layer [28].

A PCIe link connects two ports and supports bidirectional data transmission, handling configuration, I/O operations, and memory transactions. In the context of FPGA systems, the FPGA typically acts as the PCIe endpoint. Physically, a PCIe link comprises one or more lanes, each formed by two differential signal pairs—one pair for transmitting and one for receiving. This makes each lane a full-duplex channel using four wires in total. Data is transferred as byte streams in both directions. Common PCIe configurations include 1, 4, 8, or 16 lanes, denoted as x1, x4, x8, and x16 respectively. For example, “x8” represents an eight-lane card or slot,

with “x16” being the largest size in common use. The PCIe topology with various components is shown in Figure 4.12 and the components of the PCIe topology are explained briefly below.

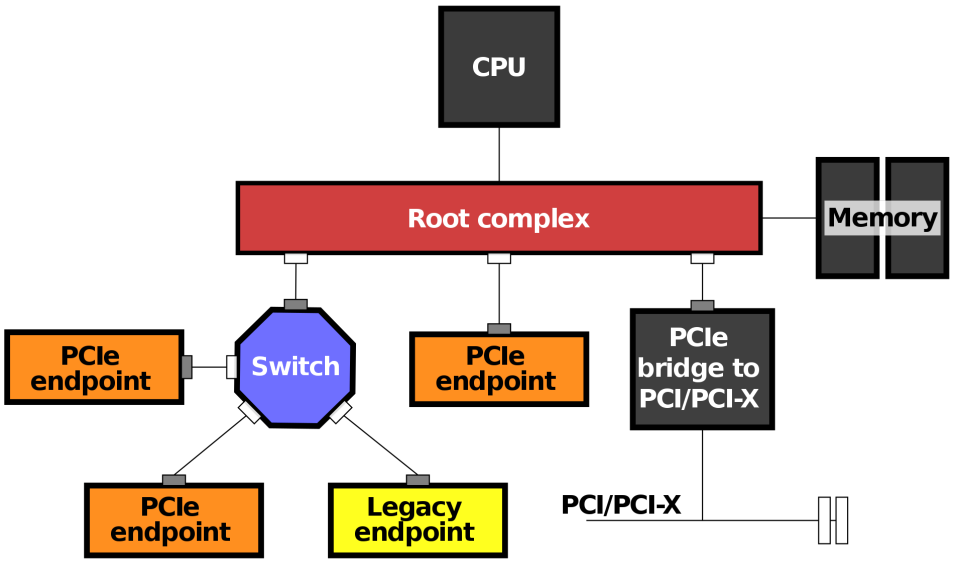


Figure 4.12: PCI Express topology [29].

- **Root complex:** It serves as the interface between the CPU and the memory subsystem with the PCIe switch fabric, which includes one or more PCIe devices [29]. Additionally, it initiates transaction requests on behalf of the CPU, which is connected via a local bus.
- **PCIe endpoints:** Endpoints are the peripheral devices that can request or complete a PCI Express transactions. They can communicate with another PCIe endpoint as well as the root complex. Through PCIe, they can gain direct access to the host CPU memory.
- **PCIe bridge / switch fabric:** These adapters enable PCI devices to interface with PCIe slots by converting protocols from the PCI standard to the PCIe standard. The PCIe bridge translates requests into point-to-point transfers over the appropriate lane within the PCIe interface [30].

Here the interface between Host and FPGA uses the Direct Memory Access (DMA) through PCIe. By using DMA, the FPGA, which is a PCIe endpoint in the topology can transfer the data without intervention of CPU at very high speeds. This serves as our interface protocol for tranfer of the chunked sequences which are sent to FPGA via DMA through PCIe.

Using a three layer protocol stack as shown in Figure 4.13, PCIe achieves reliable data transfers [30]. There are three layers involved in the transmission of a packet between two PCIe devices;

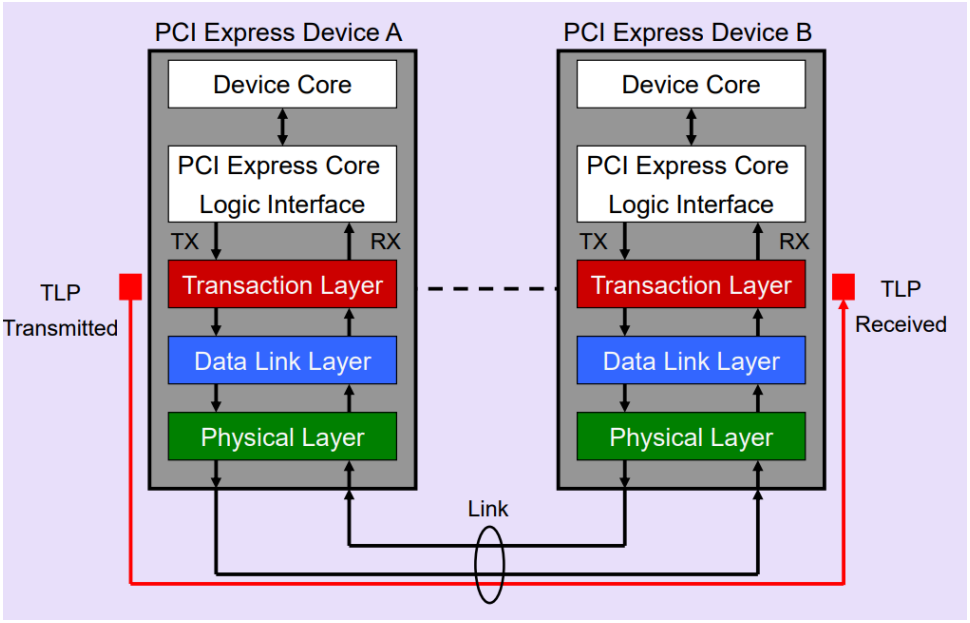


Figure 4.13: PCIe protocol stack

- **Physical layer:** It ensures reliable data transmission over the link by performing encoding and recovering the clock from the incoming data stream. Additionally, it uses a Cyclic Redundancy Check (CRC) to detect and correct any data errors that may occur during communication.
- **Datalink layer:** It verifies incoming packets for errors using retransmission mechanisms and oversees the acknowledgment process to ensure data integrity.
- **Transaction layer:** It receives 32-bit units known as double words from the master device, which are then formatted into packets containing both address and data information before being passed to the data link layer. These units, referred to as Transaction Layer Packets (TLPs), include a header and an optional payload that carries the actual data to be transmitted.

4.7 Kernel Code Structure

This section provides a detailed explanation of the BWFA kernel for sequence alignment with affine gap penalties. The kernel is optimized for FPGA hardware acceleration, utilizing memory buffers and pipelining to perform efficient computations for DNA sequence alignment.

4.7.1 Header Files

At the beginning of the kernel code, several essential header files are included for functionality and hardware interaction:

- `ap_int.h`: Provides fixed-width integer types, optimized for high-level synthesis (HLS).
- `hls_stream.h`: Facilitates streaming data between hardware modules, which is useful for efficient data transfer.
- `string.h`: Used for sequence manipulation, handling string operations.
- `stdio.h`: Provides I/O functionality for debugging and printing results.

This section allows the kernel to interact efficiently with hardware, manage memory, and handle sequence strings.

4.7.2 Needleman-Wunsch Scoring Function

The core function of the BWFA kernel is the Needleman-Wunsch scoring function, which calculates the dynamic programming matrix. It compares possible outcomes such as match, mismatch, insertion or deletion and then selects the best alignment score for each pair of characters in the sequences.

Listing 4.7 shows the code snippet of the scoring function:

Listing 4.7: Needleman-Wunsch Scoring Function

```
void needleman_wunsch_scoring(const char* seq1, const char* seq2,
    int* prev_D, int* prev_P, int* prev_Q, int* curr_D, int* curr_P,
    int* curr_Q) {
     // Initialize first row and column based on gap penalties
     for (int i = 0; i <= len1; i++) {
         prev_D[i] = gap_penalty * i;
     }
     for (int j = 0; j <= len2; j++) {
         prev_P[j] = gap_penalty * j;
     }

     // Matrix filling based on match, insert, and delete operations
     for (int i = 1; i <= len1; i++) {
         for (int j = 1; j <= len2; j++) {
             int match_score = (seq1[i-1] == seq2[j-1]) ?
                match_penalty : mismatch_penalty;
             curr_D[j] = max(prev_D[j-1] + match_score, max(prev_P[j
                -1] + gap_penalty, prev_Q[j-1] + gap_penalty));
             curr_P[j] = max(prev_D[j] + gap_penalty, prev_P[j] +
                match_score);
             curr_Q[j] = max(prev_D[j] + gap_penalty, prev_Q[j] +
                match_score);
         }
     }
}
```

This function initializes the dynamic programming table's first row and column based on gap penalties and then iterates through the sequences to fill the complete matrix with scores based on Needleman-Wunsch algorithm.

4.7.3 BWFA Alignment Function

The BWFA alignment function shown in Listing 4.8, splits the sequences into smaller segments and computes the alignment in parallel, reducing the overall time complexity. It performs recursive alignment until the sequences are small enough (reaching base case) to process efficiently.

Listing 4.8: BWFA Alignment Function

```
void bwfa_alignment(const char* seq1, const char* seq2, int* prev_D,
    int* prev_P, int* prev_Q, int* curr_D, int* curr_P, int* curr_Q)
    {
    if (len1 == 0 || len2 == 0) { // Base case for empty sequences
        return;
    }
    // Divide sequence into two parts and compute alignment
    int midpoint = len2 / 2;
    needleman_wunsch_scoring(seq1, seq2 + midpoint, prev_D, prev_P,
       prev_Q, curr_D, curr_P, curr_Q);
    char rev_seq1[MAX_SEQ_LEN], rev_seq2[MAX_SEQ_LEN];
    int i=0, j=0;
    for (i = 0; i < len1_sub; i++) rev_seq1[i] = seq1[sub.end1 - 1 -
        i];
    for (j = 0; j < mid; j++) rev_seq2[j] = seq2[sub.end2 - 1 - j];
    rev_seq1[i] = '\0';
    rev_seq2[j] = '\0';
    needleman_wunsch_scoring(rev_seq1, rev_seq2, len1_sub, len2_sub
       - mid, gap_penalty, gap_extend, match_score, mismatch_penalty
       , score2);
    int best_split = 0;
    int best_score = score1[0] + score2[len1_sub];

    for (int i = 1; i <= len1_sub; ++i) {
        int split_score = score1[i] + score2[len1_sub - i];
        if (split_score >= best_score) {
            best_score = split_score;
            best_split = i;
        }
    }
    stack[stack_ptr++] = {sub.start1 + best_split, sub.end1, sub.
       end2 - mid, sub.end2};
    stack[stack_ptr++] = {sub.start1, sub.start1 + best_split, sub.
       start2, sub.end2 - mid};
}
```

This function handles large sequences by recursively dividing them into smaller subproblems, optimizing the alignment process by handling each smaller problem in parallel. The complete working of this process is explained in Design Phase section.

4.8 Demo Setup

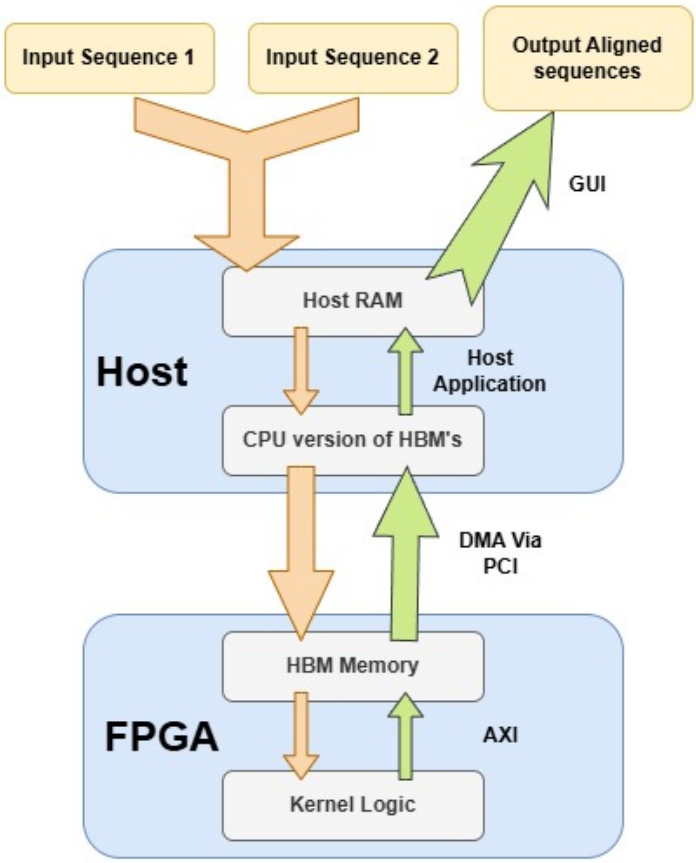


Figure 4.14: Demo Setup

Demo setup of the implemented design is shown in Figure 4.14. This system implements a hardware-accelerated DNA sequence alignment using Xilinx Alveo U50 FPGA card. The design is written and implemented using C++ and utilizes Xilinx Runtime (XRT) APIs to interface with the FPGA hardware. The compute kernels run in parallel across multiple high-bandwidth memory (HBM) channels to optimize throughput.

4.8.1 System Architecture

The top-level system consists of the following major components:

- **Host CPU**: Executes the C++ host application that collects and loads sequences in to RAM, then makes them in to chunks addressable by DMA through PCIe and keep it ready for dispatch to kernels on FPGA.
- **PCIe Interface**: Handles data transfer between the host and FPGA card using Direct Memory Access (DMA).
- **Xilinx Alveo U50 FPGA**: Hosts custom hardware kernels implemented in HLS (High-Level Synthesis). These kernels are compiled using the Vitis platform and programmed into the FPGA via the `xclbin` file.
- **HBM (High Bandwidth Memory)**: Each kernel uses a separate HBM pseudo-bank to ensure memory access isolation and parallelism. The buffers created are mapped to these HBM's and kernel can access them using AXI4 protocol.
- **Kernels**: Four parallel instances of the DNA alignment kernel execute concurrently on the FPGA. Each kernel reads input data from its HBM bank, performs alignment, and writes results back.

4.8.2 Data Flow Pipeline

The end-to-end data flow can be broken down into the following stages:

1. **Sequence Input (Host)**: The user enters paths to two DNA sequence files via CLI/GUI. These files are loaded into strings on the host side.
2. **Buffer Allocation (Host)**: Host allocates memory buffers using XRT APIs (`xrt::bo`). These buffers reside on the FPGA device, specifically within the assigned HBM pseudo-channels.
3. **Buffer Mapping and Copy (Host)**: Input sequences are split into chunks based on (`MAX_SEQ_LEN = 64`) and copied into the mapped host buffers. `memcpy` operations are used to fill these buffers from RAM before synchronization.
4. **DMA Transfer to FPGA (Host → HBM)**: Using `bo.sync(XCL_BO_SYNC_BO_TO_DEVICE)`, the host triggers DMA-based transfer of sequence data to the respective HBM banks.
5. **Kernel Execution (FPGA)**: Each kernel receives pointers to its assigned input/output buffers. It performs pairwise sequence alignment based on scoring parameters (match, mismatch, gap) and writes results into output buffers.
6. **DMA Transfer to Host (HBM → Host)**: Output buffers are synced back using `XCL_BO_SYNC_BO_FROM_DEVICE`, enabling host access to results.
7. **Result Aggregation and Display**: The host program collects all chunks of the aligned sequences and prints the final output. Execution times are also measured for performance profiling.

4.8.3 High Bandwidth Memory (HBM) Usage

Each kernel accesses a unique HBM pseudo-channel to avoid contention:

- Xilinx Alveo U50 offers up to 8GB HBM split into 32 pseudo-channels.
- Each pseudo-channel provides independent 256-bit wide access.
- In this demo, every kernel is assigned dedicated input/output buffers in separate pseudo-channels.
- This design choice ensures high parallel throughput and minimizes memory bottlenecks.

4.8.4 Host–FPGA Communication

The communication between the host and FPGA is managed by the Xilinx Runtime (XRT):

- **XCLBIN Loading**: The `xclbin` file contains hardware configuration and is loaded using `device.load_xclbin()`.
- **Buffer Objects (BO)**: Host allocates BOs which represent device-side memory.
- **PCIe DMA**: Data transfers are facilitated using PCIe Gen4 with DMA support.
- **Synchronous Operations**: `sync()` APIs manage data movement with explicit control.

4.8.5 Performance Profiling

The following timing metrics are recorded:

- **Total execution time**: Time from sequence loading to final output.
- **Kernel execution time**: Per-kernel time per iteration.
- **Maximum and cumulative kernel time**: For performance optimization and hotspot identification.

4.8.6 Design Extensibility

- Additional kernels can be added by extending `TOTAL_KERNELS` and defining more buffer sets.
- Scoring schemes can be modified dynamically via host input parameters.
- FPGA-side logic can be customized using either HLS or RTL kernel flows.

Chapter 5

Results

5.1 Simulations

The sequences used for the following simulation are of length 64, which is chosen just for an ease of capturing all the command window and report.

5.1.1 Simulation of designed BWFA C++ Code

The C++ code has been saved as `allignCode.cpp`, it is compiled using g++ compiler to generate the `allignCode.o` file which when executed using `./allignCode` will run in command window performing the sequence alignment and display the aligned sequences as shown in Figure 5.1.

```
rclab@rclab-PowerEdge-T550:~/Downloads$ g++ -std=c++17 allignCode.cpp -o allignCode
rclab@rclab-PowerEdge-T550:~/Downloads$ ./allignCode
Aligned Sequences:
CCGTGGCCCCGGGCGCCTTC---CTGGGCC----TGAAGG-CGC-TGCGATGGCT--GGACCTGTCCCACATGGA
A--TGGTGCTGT-CTCCTGCCGACAAGACCAACGTCAAGGCCGCCTG----GGGTAAGGTC--GGCGCGCACG--
Length of Sequences: 64
Time taken: 3.19126 ms
```

Figure 5.1: C++ Code Simulation

In software, there is no need of `MAX_SEQ_LEN` field because the computation is done dynamically and the memory requirements are met based on the input while the programming is running. Even for the length of input sequences as 64, the CPU only design takes *3.2msec* but for the CPU + FPGA design the time taken is shown in Figure 5.2 which is *1.2msec*. Clearly implemented design performed well for these inputs.

5.1.2 Hardware Simulation Through Command Line Interface (CLI)

The Hardware designed for sequence alignment has run on terminal and the aligned sequences along with the various input parameters and steps in hardware execution is shown in Figure 5.2.

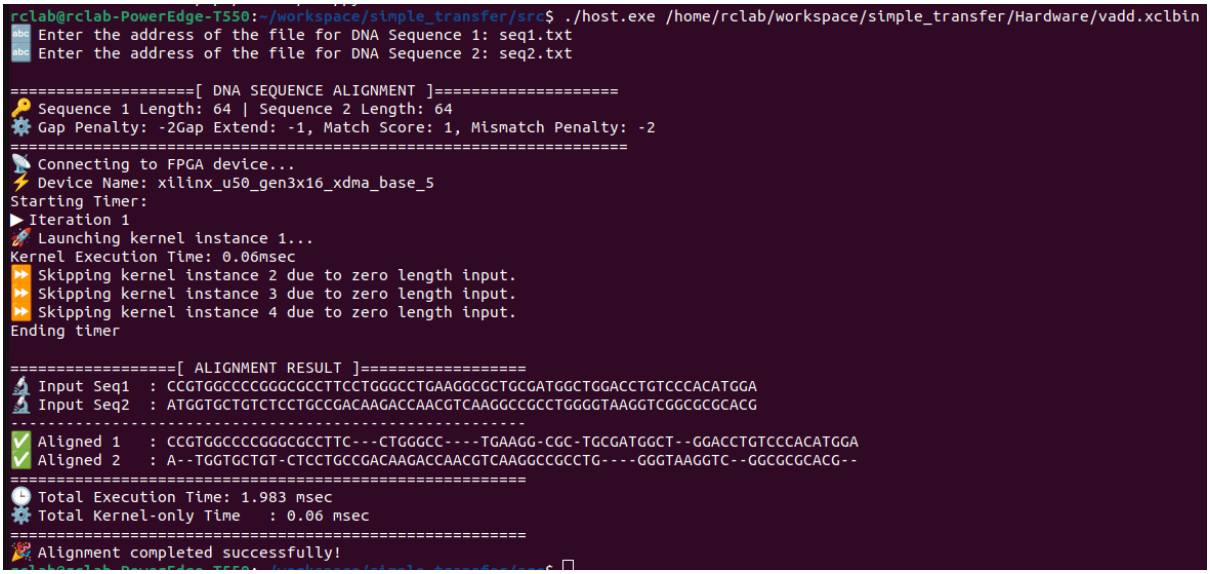


Figure 5.2: Hardware Simulation through CLI

5.1.3 Hardware Simulation Through Graphical User Interface (GUI)

The Hardware designed for sequence alignment has been run using the GUI designed and results are shown in Figure 5.3.

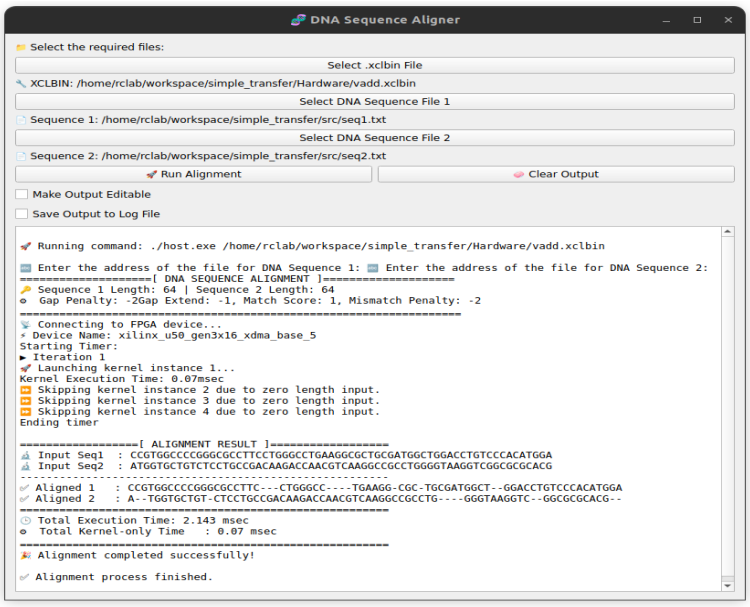


Figure 5.3: Hardware Simulation through GUI.

5.2 Timing Report

Table 5.1 shows the Target and Estimated frequencies of various kernels used in the design. Figure 5.4 shows the Timing summary report generated by the vitis tool after the synthesis of implemented design.

Compute Unit	Kernel Name	Module Name	Target Frequency (MHz)	Estimated Frequency (MHz)
bwfa_kernel_1	bwfa_kernel	bwfa_alignment	300.300	411.015
bwfa_kernel_1	bwfa_kernel	bwfa_kernel	300.300	411.015
bwfa_kernel_2	bwfa_kernel	bwfa_alignment	300.300	411.015
bwfa_kernel_2	bwfa_kernel	bwfa_kernel	300.300	411.015
bwfa_kernel_3	bwfa_kernel	bwfa_alignment	300.300	411.015
bwfa_kernel_3	bwfa_kernel	bwfa_kernel	300.300	411.015
bwfa_kernel_4	bwfa_kernel	bwfa_alignment	300.300	411.015
bwfa_kernel_4	bwfa_kernel	bwfa_kernel	300.300	411.015

Table 5.1: Target vs. Estimated Frequency of Kernels

The **target frequency** is the user-specified desired clock frequency for the hardware design we are synthesizing. It is set before synthesis and tells the HLS tool what timing constraints to aim for. The HLS tool will try to schedule operations and optimize pipelining to meet this frequency when synthesizing RTL from C/C++ code. The **estimated frequency** is the achievable frequency as calculated by the HLS synthesis tool, based on actual delays in the generated RTL and design logic. If the estimated frequency $\geq$ target frequency, our design is in accordance with timing. The Worst Negative Slack (WNS) is 0.019ns at an operating Time Period of 3.33ns. Therefore,the Maximum Operating Frequency can be calculated as:

$$f_{max} = \frac{1}{3.33ns - 0.019ns} = \frac{1}{3.311} = 302.023MHz. \tag{5.1}$$

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.019 ns	Worst Hold Slack (WHS):	0.009 ns	Worst Pulse Width Slack (WPWS):	0.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	715692	Total Number of Endpoints:	713575	Total Number of Endpoints:	284965

All user specified timing constraints are met.

Figure 5.4: Timing summary

5.3 Resource Utilization

Resource utilization refers to the amount of hardware resources (like logic elements, memory blocks, multipliers, etc.) that are consumed by our synthesized design on the target FPGA.

This is a critical metric for evaluating whether our hardware design fits within the available resources of the chosen FPGA device. Figure 5.5 shows the utilization per single kernel generated as part of system diagram by the Vitis tool and the resources used by various kernels of the implemented design are shown in Table 5.2. Post-synthesis utilization refers to the estimated usage of FPGA resources after High-Level Synthesis (HLS) has been performed, but before place and route (P&R) steps in Vivado. This gives us an approximation of how our C/C++ design will map onto the target FPGA fabric. The post synthesis utilization summary of the implemented design generated by the tool is shown in Figure 5.6. We use resource utilization analysis during early design to estimate the resource footprint before synthesis and post-synthesis utilization after running synthesis to see realistic estimates for area, timing, and performance before exporting to Vivado.

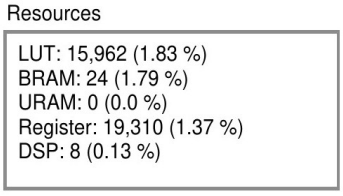
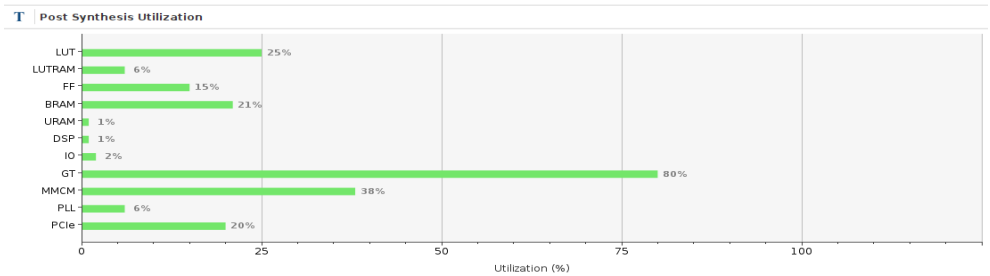


Figure 5.5: Resources Per Kernel.

Name	Kernel	LUT	Register	BRAM	URAM	DSP
bwfa_kernel_1	bwfa_kernel	15,962	19,310	24	0	8
bwfa_kernel_2	bwfa_kernel	15,962	19,310	24	0	8
bwfa_kernel_3	bwfa_kernel	15,962	19,310	24	0	8
bwfa_kernel_4	bwfa_kernel	15,962	19,310	24	0	8

Table 5.2: Resource Utilization Summary for Each Kernel Instance



Resource	Utilization	Available	Utilization %
LUT	219067	870720	25.16
LUTRAM	24779	402720	6.15
FF	266280	1741440	15.29
BRAM	283.50	1344	21.09
URAM	4	640	0.63
DSP	36	5952	0.60
IO	10	416	2.40
GT	16	20	80.00
MMCM	3	8	37.50
PLL	1	16	6.25
PCIe	1	5	20.00

Figure 5.6: Post Synthesis Utilization.

5.4 Performance Analysis

5.4.1 Length of input sequences *Vs* Speedup

Multiple simulations were performed with various input sequences of different lengths, Hardware and Software times were observed and hence calculated. The calculations were performed by changing the design factor `MAX_SEQ_LEN` of a base kernel as 64, 128 and 256 and the length of input sequences were varied from 8 to 65536. The results were plotted as shown in Figure 5.7. Where each data point corresponds to number of times hardware aided design is better than CPU only based design. From the Figure 5.7 we can understand that for the length of base kernel `MAX_SEQ_LEN` = 64, the performance is the best as compared to other base kernel lengths. So, in our design we have chosen length of base kernel as 64.

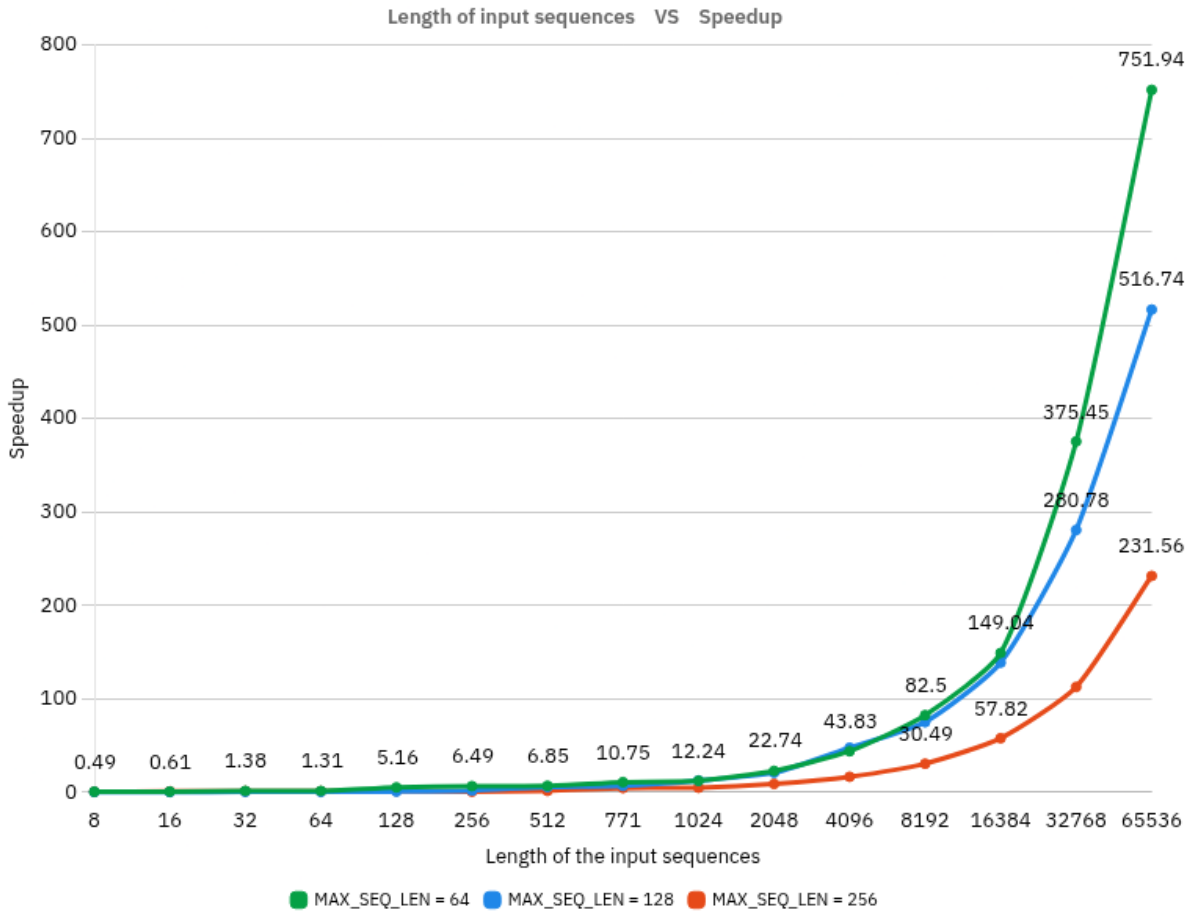


Figure 5.7: Length of input sequences *Vs* Speedup

5.4.2 CPU Only performance *Vs* CPU + FPGA performance

For `MAX_SEQ_LEN` = 64, both hardware and software times are measured for input sequences of various lengths and is shown in Figure 5.8. As the deviation between them is too high, we

can not get the detailed information in a linear plot, hence we used a logarithmic plot. Each data point in the plot corresponds to the time taken to align sequences in milliseconds. This plot shows how well the design is doing better than the CPU only design. In real scenarios the lengths of input sequences are often very high, 100 to 10,000 for medium read lengths and 10,000 to 10,00,000 for large read lengths. using CPU only design will take lot of time and also this keeps the CPU so busy that it can not address other intensive tasks especially in CPU's with low resources.

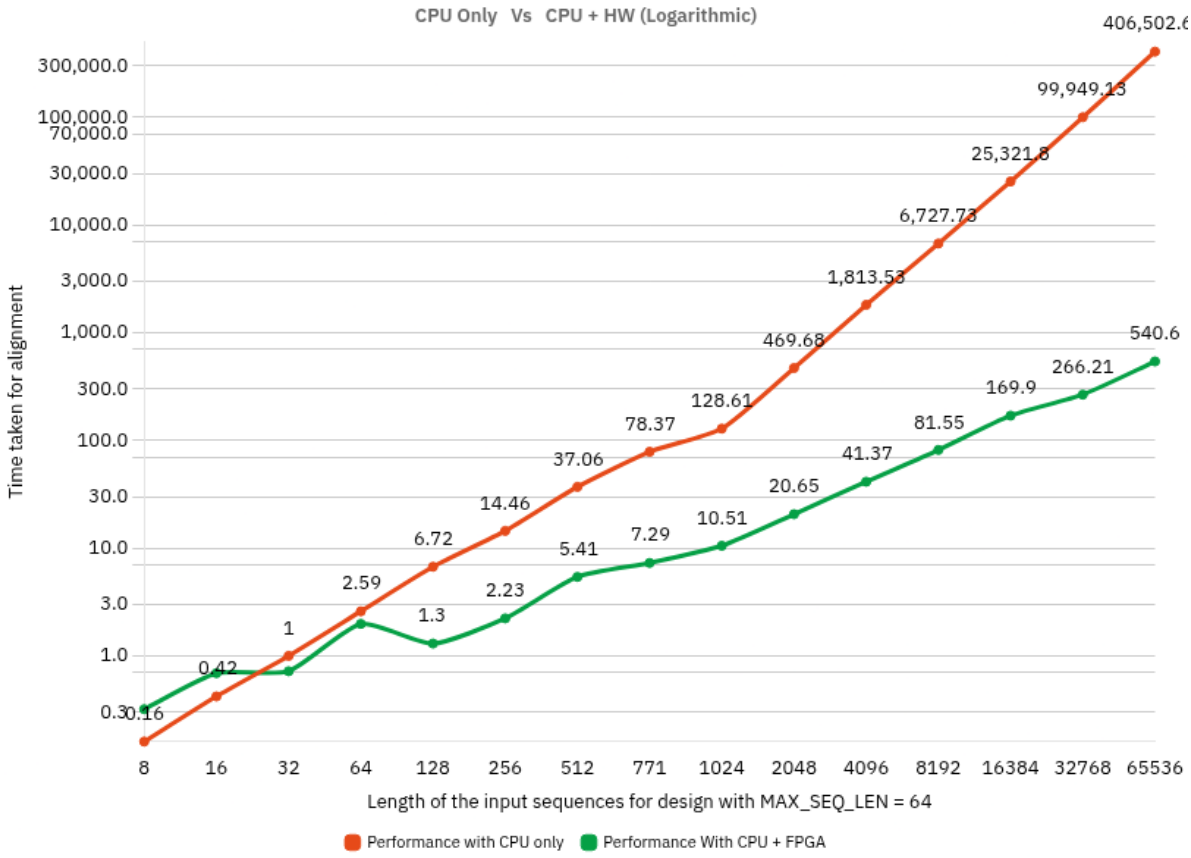


Figure 5.8: CPU Only performance *Vs* CPU + FPGA performance

5.5 Correctness of Aligned Sequences

The National Center for Biotechnology Information (NCBI), a US government agency under the National Institutes of Health (NIH), was established in 1988 and serves as a central resource for biological data, computational tools, and scientific literature related to molecular biology, genomics, and bio-informatics. There is a BLAST tool that is used to compare nucleotide or protein sequences against NCBI databases. The aligned sequences using implemented design and the NCBI tool are shown in Figure 5.9 and Figure 5.10 respectively.

```
==================[ ALIGNMENT RESULT ]==================
🔬 Input Seq1  : CCGTGGCCCCGGGCGCCTTCCTGGGCCTGAAGGCGCTGCGATGGCTGGACCTGTCCCACATGGA
🔬 Input Seq2  : ATGGTGCTGTCTCCTGCCGACAAGACCAACGTCAAGGCCGCCTGGGGTAAGGTCGGCGCGCACG
--------------------------------------------------------
✅ Aligned 1   : CCGTGGCCCCGGGCGCCTTC---CTGGGCC----TGAAGG-CGC-TGCGATGGCT--GGACCTGTCCCACATGGA
✅ Aligned 2   : A--TGGTGCTGT-CTCCTGCCGACAAGACCAACGTCAAGGCCGCCTG----GGGTAAGGTC--GGCGCGCACG--
========================================================
```

Figure 5.9: Results from the Implemented Design

NW Score	Identities	Gaps	Strand
-39	37/76(49%)	24/76(31%)	Plus/Plus

```
Query  1   CCGTGGCCCCGGGCGCCTTCCTG----GGCC----TGAAGGC-GC-TGCGATGGCT--GG  48
           ||| | | | ||| || |    | ||    | |||||| || ||    ||| | ||
Sbjct  1   A--TGGTGCTGT-CTCCTGCC-GACAAGACCAACGTCAAGGCCGCCTG----GGGTAAGG  52

Query  49  ACCTGTCCCACATGGA  64
           |  | | | | || |
Sbjct  53  TC--GGCGCGCACG--  64
```

Figure 5.10: Results from National Center for Biotechnology Information Website

5.6 Conclusion and Summary

Genome sequencing is a vast field of bio-informatics where determining the order of nucleotide bases (A, T, C, and G) is of high importance in the analysis and study of the genome under consideration. Pairwise sequence alignment is one such field under genome sequencing that arranges nucleotide bases in both sequences to identify regions of similarity that may be the result of functional, structural or evolutionary relationships between both sequences. Performing sequence alignment between two nucleotide sequences is of high importance to identify regions of similarity and evolutionary differences between the sequences. The alignment also enables bio-informaticians to construct a phylogenetic tree, which signifies how the current genome under study could have undergone evolutionary changes from a parent nucleotide over time. Current sequence alignment problems require the alignment of lengthy, highly variable and extremely numerous sequences. Hence, significant research has gone into devising algorithms to produce high quality sequence alignments and reducing the time and memory space required for computations. The Bi-WFA acts like a divide and conquer method, involved recursive computations of problems which increased in number with time. Therefore, the ability to perform these independent computations simultaneously motivates the development of an FPGA-based hardware accelerator for the Bi-WFA algorithm. In this context, a novel design was realised to implement the Bi-WFA algorithm in hardware to speedup the computation of the alignment with respect to software. Design details and summary of the work done till now is presented in this report. The design was implemented successfully on the Alveo U50 data center based accelerator card, and the results were in accordance with the same algorithm implemented in software.

A complete demo setup that involved a CPU-FPGA co-design was created with the help of Xilinx runtime libraries (XRT) that provided the user with an end-to-end means of providing DNA

sequences as inputs and observing the alignment outputs to interact with the hardware running on the FPGA. A vast number of test DNA sequences which had a dissimilarity percentage ranging from 10% to 100% were provided as input to the FPGA and the results were verified with the alignment obtained in software. The host-kernel interface presented in this work has evolved from a basic single-kernel prototype to a high-performance, multi-kernel solution with an integrated graphical user interface (GUI). By leveraging the massive parallelism offered by Xilinx Alveo FPGA cards, the system significantly improves the throughput and responsiveness of sequence alignment tasks, especially when dealing with arbitrarily long DNA inputs. The adoption of parallelism through multiple HLS-based compute kernels and iteration-based processing allowed the system to efficiently divide and conquer input sequences, achieving high scalability. Each kernel accesses dedicated HBM pseudo-channels, ensuring memory access independence and minimizing bandwidth contention. This hardware-level optimization, facilitated by Xilinx's Vitis toolchain provides fine-grained control over memory allocation, buffer management, and kernel execution profiling. A critical challenge addressed in this evolution was the shift from handling fixed-length inputs to managing variable length DNA sequences. This required dynamic yet controlled memory operations, iterative kernel launches, and robust memory synchronization logic via DMA over PCIe. The host-side C++ application orchestrates this flow, handling tasks such as sequence chunking, buffer allocation, execution timing, and user interaction, while the FPGA performs high-speed alignment operations.

Through practical measurements, the system demonstrates its ability to process thousands of sequence chunks with execution times as low as a few milliseconds per chunk. In contrast, CPU-only implementations suffer from long runtimes (especially with medium to large read lengths ranging from 1,000 to 1,000,000 bases) and tend to monopolize CPU resources, making them unsuitable for multi-tasking environments or low-resource edge devices. This implementation shows that domain-specific acceleration using FPGA hardware can bridge the performance gap in computational biology, offering real-time capabilities, energy efficiency, and substantial acceleration over traditional CPU-based methods.

5.7 Future Scope

The demonstrated architecture is a strong proof-of-concept for FPGA acceleration in computational genomics. It not only proves the feasibility of real-time DNA alignment using parallel compute hardware but also sets a strong foundation for future developments that can scale with the rapidly growing demand of precision medicine and personalized genomics. The following points summarize the specifications and working conditions of the current design along with the fields where improvements can be made.

- **Improvements in Algorithm:** We can improve the existing algorithm or replace it with a better algorithm which reduces the time and memory space required for computations. Prospects regarding implementing multiple sequence alignment which aligns more than two sequences can also be studied.
- **Support for Large-Scale Genomics Pipelines:** The current architecture can be integrated into full-genome alignment tools by interfacing with external data parsers and

genomics databases. Enhancing the system to handle batch alignment of thousands of sequences would support use cases like genome-wide association studies (GWAS) and next-generation sequencing (NGS) workflows.

- **Dynamic Kernel Scaling:** Implementing a scheduler that dynamically adjusts the number of active kernels based on input load, device utilization, and thermal headroom would improve efficiency in diverse runtime conditions. No of kernel instances per iteration can be increased which in turn use more resources but yields in better performance.
- **Multi-FPGA Scaling and Inter-FPGA Communication:** Extending the design to multi-FPGA platforms (such as using multiple Alveo cards or FPGAs in data center racks) can handle extremely long genomic reads and large batch processing. Inter-device communication protocols (e.g., CCIX, PCIe peer-to-peer) can be explored for such scalability.
- **AI Integration:** Combining alignment with AI-based post-processing (e.g., variant calling or clustering) could transform this FPGA-accelerated system into a hybrid smart pipeline capable of end-to-end genomic analysis.
- **Multiple Users Support:** With more kernels we can also facilitate multiple users to give different input sequences they want to align and process them on the same resources based on their lengths.

Bibliography

[1] *Textbook*,Bio informatics Sequence and Genome Analysis, David W. Mount University of Arizona, Tucson

[2] *Textbook*,CIGAR string description

[3] *Website*,EMBL format

[4] *Website*,National Center for Biotechnology Information

[5] *Website*,Sequence Analysis Tools, EMBL's European Bioinformatics Institute

[6] *Website*,Genome Magician - Ultra fast local DNA sequence search for NGS data

[7] *Website*,CUDAlign - tool that aligns huge DNA sequences in CUDA capable GPUs

[8] *Website*,G-PAS – Fast GPU version of a Pairwise Alignment algorithms

[9] *Website*,Kalign — A Fast Multiple Sequence Alignment Tool

[10] *Website*,Freiburg RNA Tools Teaching - Needleman-Wunsch

[11] *Website*,National Library of Medicine

[12] WFA-FPGA: An efficient accelerator of the wavefront algorithm for short and long read genomics alignment

[13] *Textbook*,An Introduction To Bioinformatics Algorithms

[14] Wavefront Algorithm

[15] Myers Difference Algorithm

[16] Optimal gap-affine alignment

[17] Reducing the Search Space and Time Complexity of Needleman-Wunsch Algorithm and Smith Waterman Algorithm for DNA Sequence Alignment

[18] *Textbook*, Bioinformatics Algorithms : An Active Learning Approach

[19] Fundamentals of Dynamic Programming

[20] WFA-GPU: gap-affine pairwise read-alignment using GPUs

[21] Vitis High-Level Synthesis User Guide (UG1399)

[22] Alveo U50 Data Center Accelerator Card

[23] Vitis Tutorials: Hardware Acceleration (XD099)

[24] *Website*,Hardware Acceleration and its importance

[25] Alveo U50 Data Center Accelerator Card Installation Guide (UG1370)

[26] Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400)

[27] Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)

[28] *Website*,PCIe interface

[29] *Website*,PCIe system

[30] *Website*,Introduction to PCIe Express

[31] Technical Information Portal

[32] Adaptive SoC & FPGA Support

[33] High-Level Synthesis for FPGA, Part 1-Combinational Circuits

[34] High-Level Synthesis for FPGA, Part 2 - Sequential Circuits Logic Design with Vitis-HLS