THE ONLY ORDER IS K.A.O.S: HARDWARE FOR SEQUENCING-BASED PATHOGEN DETECTION

KELECHI OTU

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCES
YORK UNIVERSITY
TORONTO, ONTARIO

JUNE, 2026

© KELECHI OTU, 2026

Abstract

Despite the remarkable miniaturization achieved by new DNA sequencing technologies, their practical applicability remains constrained by limited embedded computing capabilities. Currently, these technologies depend on large external commodity computing resources to handle their bioinformatics workloads. This thesis presents a system-on-chip (SoC) design that could potentially be integrated into the internal chipset of such miniature sequencers in the future. The proposed design features an architecture combining general-purpose computing with RISC-V processors and a specialized accelerator for rapid sequence comparison. The hardware accelerator is primary contribution of this work, it is designed for real-time pathogen detection through the alignment of incoming sequencing reads against a pathogen's genome. This thesis examines the motivations underlying the design decisions, the hardware architecture governing its implementation, and experimental results demonstrating its ability to support the computational demands of real-time sequence comparisons within an embedded systems environment.

Acknowledgements

I would like to start by expressing my sincere gratitude to my supervisor, Dr. Sebastian Magierowski. His guidance, advice, and support throughout the process of completing this thesis has been invaluable. His supervision greatly improved my approach to the design, research, and presentation of this work. I would also like to thank Zhongpan Wu for his willingness to share his knowledge and expertise. His postgraduate work was not only an inspiration for this thesis, but also provided the foundation upon which it was built. I am also profoundly grateful to Amin Savari who was a collaborator for the vast majority of this work. The countless hours he spent helping to design and test this project cannot be overstated. I am truly fortunate to have had the opportunity to undertake this work with him. Lastly, I extend my heartfelt appreciation to the rest of my colleagues at E.M.I.L for their support, insight, and encouragement.

Table of Contents

Abstract **ii**

Acknowledgements **iii**

Table of Contents **iv**

List of Tables **viii**

List of Figures **ix**

Abbreviations **xii**

1 Introduction **1**

1.1 Importance of Pathogen Detection 1

1.2 DNA BACKGROUND 2

1.3 Brief history of DNA Sequencers 4

1.4 Third Generation Basecalling Pipeline 5

1.5 Motivation 7

1.5.1 Alignment and Hardware Constraints 8

1.6 Viral Detection 9

1.7 Design Vision and Alternatives 10

1.7.1 Kraken 2 11
1.8 Thesis Outline and Goal 12

2 Alignment Algorithms **14**
2.1 Algorithm Overview 14
2.2 Sequence Comparisons 15
2.2.1 Classic Dynamic Programming Framework 16
2.2.2 Dynamic Programming for Sequence Alignment 16
2.2.3 Global Alignment Algorithm 20
2.2.4 Local Alignment Algorithm 22
2.2.5 Fitting Alignment Algorithm 25
2.3 Correctness 27
2.4 Alignment Scores and Visualizing 29

3 System Architecture **31**
3.1 Motivation 31
3.2 System Components 33
3.2.1 Data Flow 35
3.3 Systolic Design 36
3.3.1 Processing Element Architecture 36
3.3.2 Systolic Array Architecture 37
3.4 Algorithm Implementation 40
3.5 KAOS Configurations 43
3.5.1 KAOS_C 43
3.5.2 KAOS_Q 43
3.5.3 KAOS_T 44
3.6 Total Memory Requirements 45

4 Traceback **47**

4.1 Traceback Overview 47

4.2 Finding the sub-DAG 48

4.3 Divide-and-Conquer 49

4.4 Pointers 53

4.5 CIGAR Format 56

5 Results **58**

5.1 Methodology 58

5.1.1 Data 58

5.1.2 Setup 59

5.2 Measurements 60

5.2.1 Throughput 60

5.2.2 Power 64

5.2.3 Cross-Platform Comparisons 67

5.2.4 State-of-the-art Comparisons 68

5.3 Fulfilling Requirements 70

5.3.1 Throughput Requirement 70

5.3.2 Power Requirement 71

5.3.3 Accuracy Requirement 71

6 Conclusion **73**

6.1 Summary 73

6.2 Future Works 75

6.2.1 Affine Penalties and Customizable Scoring matrix 75

6.2.2 Traceback in Hardware 76

6.2.3 Pathogen Detection Framework 77

Bibliography **79**

A Appendix **88**

A.1 C Implementation of Fitting Alignment 88

A.2 C Implementation of Global Alignment 90

A.3 Verilog Implementation of Processing Elements 92

List of Tables

5.1 Performance, Runtime, and Power Comparison of Fitting Alignment Software Running Across Different Architectures, Including Our Proposed Accelerator. 67

5.2 Comparison of SquiggleFilter, WFAasic, and Our KAOS variants 68

List of Figures

1.1 Double Helix Structure of a DNA molecule with nucleotides [1]. 3

1.2 Three generations of DNA sequencing technology [2]. 5

1.3 Outputs at different stages of a nanopore sequencing pipeline [3]. 6

1.4 A typical MinION setup [4]. 7

2.1 Example of a scoring matrix for a sequence alignment algorithm [5]. 17

2.2 Sequence alignment DAG arranged in a two-dimensional matrix [6]. 18

2.3 A DAG cell's three neighboring cells and their corresponding operations. . . 19

2.4 An abstract example of the DAG for global alignment. Optimal path starts at the origin and ends at the last cell in the DAG. 21

2.5 an abstract example of the DAG for local alignment. The optimal path can start and end anywhere in the DAG. 23

2.6 an abstract example of the DAG for fitting alignment. Optimal path can start anywhere in the first column and end anywhere in the last column. 25

3.1 Basic KAOS Architecture showing dataflow between the controller and datapath. Data flow is shown with arrows. 34

3.2 SoC architecture showing the flow of data between the core, memory components, RoCC interface and KAOS. 35

3.3 Logic inside the processing elements. A Verilog implementation is in Appendix A.3. 36
3.4 Architecture of the systolic array. 39
3.5 Cycle by cycle processing of a miniature version of the DAG by the systolic array. The DAG has a query with 4 bases, reference with 6 bases, and is processed by a systolic array with 3 PEs. The figure shows the wavefront parallel manner of computation. Also shows how PEs transition from one round to another. 41
3.6 The SoC architecture for KAOS_Q. 44

4.1 Finding the sub-DAG from the original. The sub-DAG is in red, original in black. 50
4.2 Divide-and-Conquer steps used to find and weave optimal nodes into an alignment path. The optimal node from middle column is used to split the DAG into a smaller DAGs shaded in blue [5]. 52
4.3 Lateral pointer connecting optimal nodes for an insertion operation. 54
4.4 Diagonal pointer connecting optimal nodes for a match operation. 55
4.5 Diagonal pointer connecting optimal nodes for a substitution operation. . . . 55
4.6 Downward and diagonal pointers used to connect optimal nodes with a vertical gap. A mixture of diagonal and downward pointers are always needed but the type of diagonal pointer and its position depends on the case. Red diagonal pointer are for match operations, purple diagonal pointer are for substitution operations 56
4.7 Base level textual representation of alignment. The reference sequence is at the top while the query sequence is at the bottom 57
4.8 CIGAR Format of alignment depicted in Fig 4.7. 57

5.1 VC707 Evaluation Board from AMD/Xilinx [7]. 59
5.2 Performance trends of KAOS with only one engine and up to 1024 PEs activated, showing processing efficiency based on the number of PEs. 61
5.3 Performance scaling trends for up to 4 engines with 64-PE configuration running in parallel, highlighting alignments per second across different clock frequencies. 62
5.4 Alignments per second measurements with 4 engines each having 64 PEs at different frequencies. 62
5.5 (a) Alignments Per Second Across number of Engines and PEs, (b) GCUPS Across number of Engines and PEs. 63
5.6 (a) Cycles per Alignment, (b) Elapsed Time per Alignment. 64
5.7 Power consumption measurements with 4 engines each having 64 PEs at different frequencies. 65
5.8 Energy efficiency measurements with 4 engines each having 64 PEs at different frequencies. 66
5.9 Power consumption across different number of engines and PEs. 66

Abbreviations

FPGA	Field-Programmable Gate Array
SoC	System on Chip
DAG	Directed Acyclic Graph
DNA	Deoxyribonucleic Acid
ASIC	Application-Specific Integrated Circuit
RISC-V	Reduced Instruction Set Computing Fifth Edition
ONT	Oxford Nanopore Technologies
CPU	Central Processing Unit
GPU	Graphics Processing Unit
RTL	Register-Transfer Level
FIFO	First-In-First-Out
GCUPS	Giga Cell Updates Per Second
ISA	Instruction Set Architecture
RoCC	Rocket Custom Coprocessor
SRAM	Static Random-Access Memory
PE	Processing Element
CIGAR	Compact Idiosyncratic Gapped Alignment Report
KAOS	K-Mer Alignment On Silicon

Chapter 1

Introduction

1.1 Importance of Pathogen Detection

Pathogen detection refers to the identification of disease-causing microorganisms, including bacteria, viruses, fungi, and parasites, within biological or environmental samples. Accurate and timely detection is a cornerstone of modern healthcare, enabling effective diagnosis, treatment selection, and epidemiological control. Pathogens are responsible for a wide range of infectious diseases, many of which can progress rapidly or spread extensively if not identified early. Consequently, robust detection methodologies are essential for both clinical diagnostics and public health surveillance.

Traditional pathogen detection techniques include culture-based methods, immunoassays, and microscopy. While these approaches can provide high specificity(true negative rate), they are often limited by long processing times, low sensitivity(true positive rate) for certain organisms, and dependence on prior knowledge of the suspected pathogen [8]. Molecular methods, particularly those based on nucleic acid analysis such as polymerase chain reaction (PCR), have significantly improved sensitivity and specificity by targeting pathogen-specific genetic sequences [9]. However, these techniques typically require laboratory infrastructure,

trained personnel, and predefined primers, which constrain their scalability and responsiveness to emerging threats.

Advances in DNA sequencing technologies have enabled a shift toward sequence-based pathogen detection, where the presence of pathogens is inferred directly from genetic material within a sample. High-throughput sequencing platforms allow for meta-genomic analysis, facilitating the unbiased detection of multiple organisms simultaneously without prior assumptions [10]. This capability is particularly important for identifying novel or rapidly mutating pathogens, as demonstrated during recent global infectious disease outbreaks. Rapid and accurate pathogen detection is critical for several reasons. First, it enables early diagnosis, which is directly correlated with improved patient outcomes and reduced transmission rates [11]. Second, it plays a vital role in outbreak monitoring and containment, providing actionable data for public health interventions and policy decisions.

Despite these advances, significant challenges remain in achieving real-time, portable, and energy-efficient pathogen detection systems. Embedded computational platforms presents an opportunity to address these issues while enabling on-site and low-latency analysis. Such systems are especially relevant in resource-constrained or point-of-care settings, where rapid decision-making is essential. This motivates the development of hardware-accelerated and embedded solutions for sequence analysis.

1.2 DNA BACKGROUND

Deoxyribonucleic acid (DNA) is the primary bio-molecule responsible for storing genetic information in living organisms. It forms the core structure of the chromosomes within the cell's nucleus and encodes the instructions necessary for biological development and function. The structure of DNA consists of two strands coiled into a double helix, where each strand is composed of nucleotide units containing one of four bases: Adenine (A), Thymine (T),

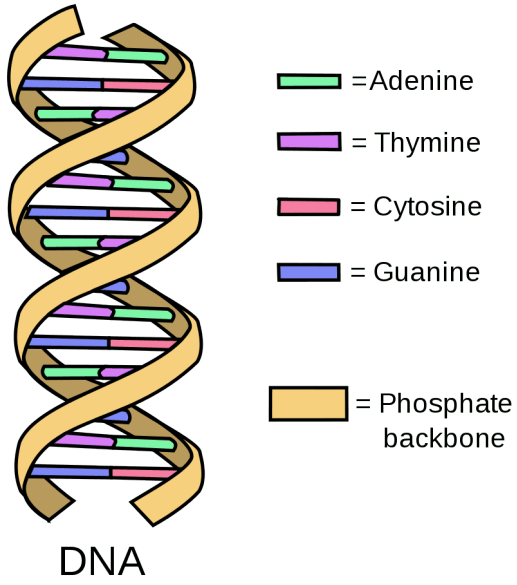


Figure 1.1: Double Helix Structure of a DNA molecule with nucleotides [1].

Cytosine (C), and Guanine (G). The strands are connected through complementary pairings, A pairs with T and C pairs with G [12], forming base pairs (bp) as illustrated in Fig1.1. These base pairs constitute the fundamental structural and informational units of DNA.

Certain groups of nucleotide bases can encode for specific physical and genetic traits, these are known as genes. Genes define the instructions for protein synthesis by specifying the order of amino acids required to construct functional proteins. During this process, the double helix is unwound, allowing a RNA polymerase enzyme to synthesize a complementary messenger RNA(mRNA) strand from the exposed template. This mRNA then transmits the encoded information to ribosomes in the cell, guiding the sequential assembly of amino acids into proteins [13]. Given that biological functions are fundamentally determined by the ordering of nucleotides, it is critical to accurately determine DNA sequences.

1.3 Brief history of DNA Sequencers

DNA sequencing aims to determine the precise ordering of nucleotide bases within a DNA sample [14]. This task is traditionally been performed using dedicated instrumentation referred to as DNA sequencers. The evolution of these technologies is commonly categorized into three generations, as illustrated in Fig. 1.2. Early first-generation sequencing methods, developed in the 1980s, relied on the Maxam–Gilbert chemical degradation approach and the Sanger dideoxy chain termination method, both of which enabled accurate but relatively low-throughput sequencing [15].

Subsequent advances led to the development of next-generation sequencing (NGS) platforms, which significantly improved throughput and scalability. Modern systems, such as the Illumina HiSeq X, are capable of sequencing DNA at rates approaching multiple human genomes per hour [16], [17]. NGS technologies primarily employ two sequencing paradigms: sequencing by ligation (SBL) and sequencing by synthesis (SBS). These approaches utilize DNA polymerase-mediated synthesis, incorporating nucleotides in a stepwise manner. Modified nucleotides emit distinct signals during incorporation, enabling optical or electronic sensors to infer base identities and reconstruct the underlying sequence [18], [19], [20]. Despite their high accuracy, these methods are constrained by biochemical limitations that typically restrict read lengths to approximately 100–200 base pairs (bp), commonly referred to as short reads.

More recently, third-generation sequencing technologies have emerged to address these limitations. In 2014, Oxford Nanopore Technologies (ONT) introduced the portable MinION sequencer, shown in Fig 1.2. This device operates by measuring variations in ionic current as nucleic acid strands translocate through nanoscale pores. Temporal fluctuations in the measured current corresponds to characteristic signatures of nucleotide sequences, enabling direct inference of base composition [21]. Unlike NGS platforms, nanopore sequencing facilitates

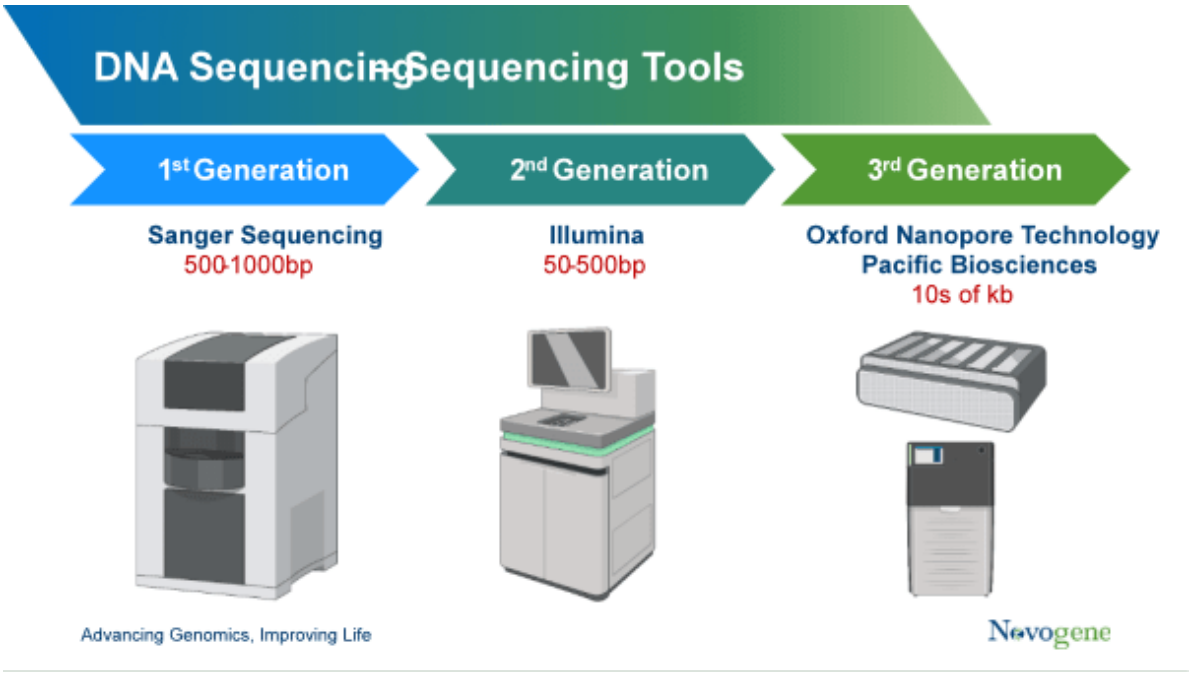


Figure 1.2: Three generations of DNA sequencing technology [2].

single-molecule analysis without requiring prior amplification or chemical labeling, allowing direct sequencing of native DNA molecules [22]. This fundamental shift in sensing modality and workflow has led to its classification as a third-generation sequencing technology [23].

1.4 Third Generation Basecalling Pipeline

The technology for third generation sequencers like MinION is based around nanopores. These are nanoscale channels that allow molecules to pass through them. They can be organic in the form of a protein or synthetic. Nanopores contain a passage just big enough to allow a single strand of DNA to travel through them. They utilize an electric field to attract DNA molecules and move them through at a constant pace, DNA molecules are unwound upon entry [24]. As the strand moves through, it disrupts an Ionic current within the nanopore. This disruption is sampled, digitized and recorded [25]. These are known as raw signals, also

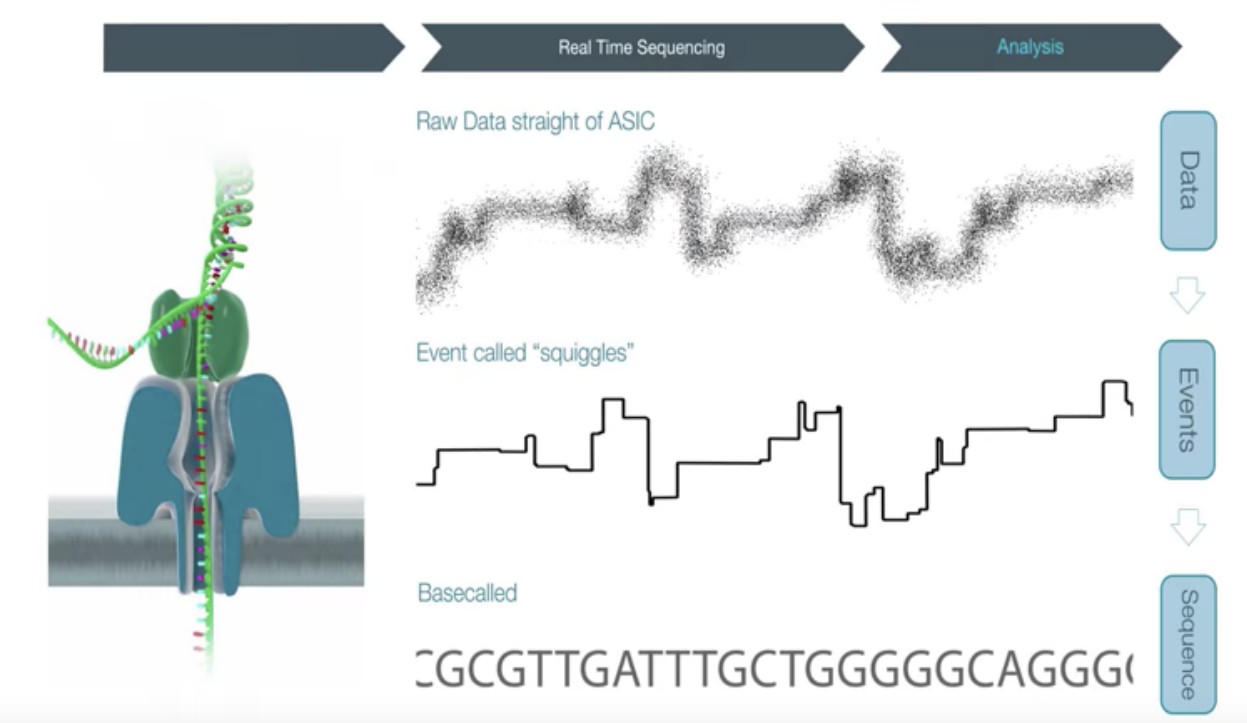


Figure 1.3: Outputs at different stages of a nanopore sequencing pipeline [3].

called squiggles. After filtering to remove noise, the signal is then known as events.

Basecalling specifically refers to the computational process of translating raw/event signals into sequences of letters that represent the measured nucleotide sequence. In modern platforms, particularly nanopore based systems, this involves inferring base identities via software algorithms. FAST5 files containing raws/events are used as input and FASTQ/FASTA containing the translated sequences are returned as output. Basecalling programs built on deep-learning models are the modern standard, some examples are ONT's Dorado, Bonito and Guppy [26]. Any embedded systems platform aiming to perform real-time pathogen detection must rely on accurately basecalled sequences. This reliance on the pipeline imposes constraints that motivate the need for efficient post-basecalling solutions.

1.5 Motivation

DNA sequencing technology has undergone a remarkable size reduction over the last decade. Most notably, low-power, pocket-sized sequencers have emerged and have been adopted by the scientific community [27]. Based on their small footprint alone, these devices hold significant potential for applications to genomic analysis, surveillance, and detection in a mobile context [28, 29]. Besides this, they also present other benefits including real-time sequencing. In such cases, electronic measurement output streams become available almost as soon as DNA samples start moving through parallel sensor arrays within the device. Further, the ability of this technology to measure very long strands of DNA compared to incumbent sequencers offers desirable advantages for downstream bioinformatic analysis [30].



Figure 1.4: A typical MinION setup [4].

Presently lacking within these small sequencers is any computational hardware dedicated to processing the measurements that are being collected, or even an internet connection. Rather, in existing hand-sized systems the data is sent to some larger adjoining computing resources (Fig 1.4) through a short wired connection. This resource may carry out some computing itself or simply forward the data to some other remote services. Although such

setups are adequate for work inside science laboratories, they clearly dampen the potential of the device for field research in remote locales. More generally, the reliance on large general-purpose computing hampers the opportunities for applying sequencing technology to a much broader array of medical and industrial products in some embedded form.

In this thesis, a System on Chip (SoC) for rapid DNA sequence comparisons is introduced to help open opportunities for miniature sequencers to serve a broader array of applications, especially those heavily emphasizing mobility, portability, or embedded needs. The vision for the SoC is to be part of a computing system encapsulated entirely within the chassis of future sequencer iterations. Alternatively, even within the current paradigm of external computing resources, our SoC could serve as an additional component sharing silicon with commodity computing parts.

1.5.1 Alignment and Hardware Constraints

The problem of genomic sequence comparison has been the subject of vigorous research for decades, a sign of the foundational importance of this work to genomic science and medicine [31]. Prominent challenges in this space often present researchers with the need to apply algorithms that span very large strings in search of sequence commonalities. These challenges include tasks such as reference-guided assembly [32] of individual human genomes or de novo assembly of newly discovered species [33] or searching vast databases for matching genomic patterns [34].

For such work, vast computing resources are needed and even clever hardware innovations impose significant resource constraints. For example, a hardware acceleration proposal [35] that achieves 1000x long-read assembly speed-up on nanopore reads would still require $\sim$12 hours to complete a human-sized genome alignment at 30X coverage. If implemented in a technology like 22-nm CMOS which is popular in embedded performance applications [**?**]

such a design would require 1 cm^2 of chip area and require nearly 10 W.

1.6 Viral Detection

Popular alignment uses and their accompanying hardware solutions as stated above are clearly too intensive for purely mobile applications and the associated constraints of the embedded computing resources upon which they are based. However, there exist alignment problems at smaller scales with significant application importance. Particularly problems that address the threat of viral pathogens, whose genomes are roughly five orders-of-magnitude smaller than human ones, this was underscored by the global SARS-CoV-2 pandemic. In this case, as in other outbreaks, the need for rapid and widely available virus detection was appreciated as a key part in managing and curtailing viral spread [36].

Among a number of techniques developed for the purpose of viral detection, the excellent potential for miniature sequencing technology was demonstrated [37, 38, 39, 40]. In these works, the utility of sequencing for hypothesis-free metagenomic analysis was emphasized. As a result, a broad spectrum pathogen detection capability was sought and sequencer outputs were used to query large databases containing billions of bases [41]. Although this approach provides wide pathogen coverage, its scale and the need to sift through tens of thousands of remotely stored records makes it impractical for today's embedded systems and lightweight point-of-care applications. Fortunately, more targeted searches, where a small set of viruses or even a single virus is screened for, are still of great use for health-care and epidemic containment. These needs have much greater chance for realization in low-power computing systems and are the inspiration for the work described in this thesis.

1.7 Design Vision and Alternatives

In this thesis, we envision a portable device in which a miniature DNA sequencer shares a chassis with an SoC. This SoC is responsible for taking inputs from the sequencer to perform basecalling and rapid DNA sequence comparisons. The SoC utilizes an accelerator in order to achieve rapid sequence comparisons, this accelerator is the primary focus of this thesis. By rapidly comparing sequenced reads from a patient's sample to the genome of a pathogen, we aim to leverage the accelerator to create a real-time pathogen detection platform. The accelerator is needed because from the context of an embedded system with limited memory looking to achieve real-time sequencing based pathogen detection, computational throughput is a bottleneck. A system with limited compute capacity would be unable to form a cohesive pipeline with the sequencer as it would be struggle to keep pace with its throughput.

The proposed SoC is primarily designed to detect a known pathogen at a time, although the architecture can support up to four pathogens depending on the configuration of the design. It is intended for scenarios in which the objective is to screen for a specific pathogen and its variants, such as during the SARS-CoV-2 pandemic. The pandemic demonstrated the value of portable and rapidly deployable diagnostic technologies. Rapid antigen tests (RATs), for example, played an important role because they were fast, widely available, and could be deployed outside traditional laboratory environments. Rapid pathogen detection helped initiate appropriate treatment, support outbreak containment, reduce adverse health outcomes, and minimize economic disruption. We believe our proposed device can have a similar impact. It also has certain advantages over chemical tests like the Rapid Antigen Test. Unlike with chemical tests, our design digitizes the majority of the process. This enables faster deployment as it does not rely on primers/reagents, it primarily needs the reference genome and its Pathogenicity Islands (PAIs). It should be noted, however, that while the computational detection process is digitally configurable, the underlying sequencing

technology may still require the biochemical reagents necessary for DNA sequencing.

PAIs are a distinct collection of segments in a pathogen's genome that allow it to be differentiated from non-pathogenic genomes, even if they have significant genomic overlaps. It should also reduce the risk of genomic overlap between different pathogens affecting detection. Our detection platform is also programmable and reusable unlike most portable chemical tests which are single-use consumables.

Inspired by portable sequencing devices such as the Oxford Nanopore MinION, the proposed architecture is envisioned for use across a broad range of diagnostic settings. It is particularly well suited to low-resource, remote, and point-of-care environments, where its portability, speed, and potential for real-time analysis may provide advantages over traditional laboratory-based pathogen detection techniques such as RT-PCR.

1.7.1 Kraken 2

As with any design, it is always important to assess alternatives. Since the SoC is capable of running software, one could question the need for a custom hardware accelerator when there are software tools like Kraken 2. Kraken 2 is a high-speed, alignment-free taxonomic classification tool that identifies the organism from which a DNA sequencing read originated. Rather than aligning reads to a reference genome, Kraken 2 analyzes the k-mers contained within each read and uses an exact k-mer lookup strategy to assign the read to the most likely taxonomic group. It reports to be able to do millions of reads per minute [42].

Even though the task tackled in this work isn't exactly classification but detection, the two tasks are closely related enough that this is a real alternative. It is also worth emphasizing that Kraken 2's throughput is platform-dependent. The same software can vary significantly in performance across different systems because its execution is constrained by the host computer's CPU and memory subsystem, whereas our ASIC accelerator has a more predictable

throughput determined by its hardware architecture. Kraken 2 obtained its measurements on a computer with 32 Intel Xeon 2.3 GHz CPUs (16 hyper-threaded cores) and 244 GB of RAM, a platform with significantly more resources than our embedded SoC. It is safe to say that running Kraken 2 on our embedded environment would see a drastic reduction in throughput relative to its reported numbers. Memory is also a concern. Kraken 2 would need frequent memory access which would clash with the basecalling software on our SoC which also demands nearly constant memory access, therefore slowing down throughput even more. Lastly, unlike our proposed design Kraken 2 produces taxonomic classifications, not sequence alignments or CIGAR strings. This extra information is useful because it facilitates deeper analyses of detections by experts which could prove vital for patient treatment. For these reasons we decided to go forward with the design for a hardware accelerator instead.

1.8 Thesis Outline and Goal

The goal of this thesis is to present K-Mer Alignment on Silicon (KAOS), a hardware accelerator integrated within a RISC-V-based SoC. KAOS implements a dynamic programming-based sequence comparison algorithm designed to determine the similarity between DNA sequences. The accelerator is intended to operate near the end of a DNA sequencing pipeline, where incoming sequencing reads are compared against the reference genome of one or more target pathogens. In this work reads contain up to 1000 bases, while the reference pathogen genome may contain up to 50,000 bases. The results of these sequence comparisons can subsequently be used to estimate the likelihood that a target pathogen is present within a biological sample.

The remainder of this thesis is organized as follows. The next chapter introduces sequence alignment algorithms, describing how they work and the rationale for selecting the algorithm implemented in KAOS. The following chapter presents the SoC and accelerator architectures, explaining how the selected algorithm is translated into a functional hardware implementation

and discussing the motivations underlying the major design decisions. The subsequent chapter addresses the traceback process and describes how sequence alignment results are reconstructed and presented while operating within the constraints of the proposed system. The penultimate chapter presents the experimental results, including measurements of power consumption, throughput, and computational efficiency. Finally, the thesis concludes with a summary of the major contributions and findings of this work, followed by a discussion of potential directions for future work.

Chapter 2

Alignment Algorithms

2.1 Algorithm Overview

In Bioinformatics, sequence alignment refers to a collection of methods for finding similarities between strings such as those derived from reads of proteins, DNA, or RNA [43]. Among these methods, dynamic programming (DP) algorithms are very common [44]. For large genome alignments, DP methods are often tasked with robustly executing a pairwise alignment between two smaller sub-sequences within similar parts of the larger given genome. This method is known as seed-and-extend. Seeding is the act of finding these very small (up to 15 base pairs in length) sub-sequence locations that match, usually guided by heuristic algorithms. Once these locations have been found, DP algorithms "extends" the search to regions before and after using DP-based string comparison algorithms [6]. The result is an efficient alignment of large genomes. Popular tools like BLAST (Basic Local Alignment Search Tool) and Minimap2 utilize seed-and-extend to align sequences [45] [46].

The context for this work is different because the genome size of pathogens are considerably smaller relative to other organisms. For example, the human genome is over three billion base pairs long while the SARS-CoV-2 virus has a genome length of around 30,000 base

pairs [47] [48]. Since our work already emphasizes rapid alignment-based detection of smaller sequences, we decided to focus on the implementation and acceleration of the DP-based string comparison method and avoid seeding entirely. This is because we could leverage the throughput of our design to process these string comparisons at the rate of the MinION sequencer, if not faster. This approach trades the generality of large-scale heuristic alignment for deterministic sequence comparison suitable for an adequately accelerated embedded system.

2.2 Sequence Comparisons

In the case of pairwise alignment, a sequence alignment algorithm's computed alignment score is an indicator of how well two sequences align. Many alignment algorithms can be used to quantify good and bad alignment between two sequences. Common examples used in genomics include Smith-Waterman also known as local alignment, Needleman-Wunsch also known as global alignment [49], and edit distance. For our design, we focus on implementing an algorithm very similar to the Shuffle-LAGAN Algorithm, a comparison method better suited to the detection approach we envision. Shuffle-LAGAN is a glocal sequence alignment algorithm introduced by Brudno et al in 2003 to address the limitations of conventional global and local alignment methods when comparing genomic sequences. The authors tested the method by splitting the mouse genome into BAC-sized pieces and aligning them against the human genome. Their analysis found that approximately 9 percent of human–mouse homology could be attributed to small rearrangements like inversions, transpositions, and duplications [50]. It is also known as semi-global alignment but for this work we will be calling it the fitting alignment algorithm.

2.2.1 Classic Dynamic Programming Framework

Dynamic programming (DP) is an algorithmic optimization technique used to solve complex problems by breaking them down into simpler, overlapping subproblems, solving each subproblem exactly once, and storing their results to avoid redundant calculations. A common tool used in DP is the Directed Acyclic graph (DAG). The graph has nodes, directed edges between nodes, and lastly no cyclic paths. The nodes symbolize a sub-instance of the problem while the directed edges represent the transition between subproblems. Edges are also weighted, this weight is the value gained or spent when moving from one subproblem state to the next. When a subproblem is solved its answer is stored and used to solve a bigger subproblem, this dependency is depicted by the connection of nodes. Since the graph is acyclic, it guarantees a valid evaluation order without infinite loops. The algorithm terminates when all nodes/subproblems have been evaluated and the answer to the problem as a whole is returned as a path in the DAG.

2.2.2 Dynamic Programming for Sequence Alignment

Sequence alignment algorithms are tasked with finding an optimal alignment of two string sequences, a reference and a query. In this case these sequences are string representations of DNA sequences. This problem can be solved using a dynamic programming framework. First a DAG is created, each subsequence alignment is assigned to a node as it represents a subproblem of the full sequence alignment problem. Nodes are connected to the nodes of slightly bigger subsequence alignments. In our case, since we are dealing with DNA sequence alignment there are only four ways to get a slightly bigger subsequence alignment. We can insert a base, delete a base, match bases, or change a base into another. Each of these actions correspond to a directed edge, so each node has at least 4 edges as input. The weight of these edges are what the algorithm deems as a penalty or reward for each action in order to get to

	A	C	G	T	–
A	+1	-μ	-μ	-μ	-σ
C	-μ	+1	-μ	-μ	-σ
G	-μ	-μ	+1	-μ	-σ
T	-μ	-μ	-μ	+1	-σ
–	-σ	-σ	-σ	-σ	

Figure 2.1: Example of a scoring matrix for a sequence alignment algorithm [5].

the next subsequence alignment. The reward and penalty values are put in a scoring matrix, an example of a scoring matrix can be seen in Fig 2.1. It shows that matches are rewarded with +1, mismatches are penalized with -μ and indels(insertion and deletions) are penalized with -σ.

Every node has an alignment score, this score rates how good the optimal subsequence alignment is for the node's subproblem. A node determines its score by looking at the scores of the nodes with smaller subproblems connected to it. It then adds the values of the edge weights to the corresponding node scores and evaluates which is the best, that score is then stored. The scores of the nodes with the smallest subsequence alignments in the DAG are calculated first and then used to determine the scores of nodes for bigger subproblems. Two things are returned, the best score from the DAG and the path to the node that produces it, this path is seen as the optimal alignment of the reference and query. This framing makes the problem very similar to the classic DP problem of finding the longest path in a DAG.

The DAG can be rearranged in a two-dimensional grid pattern as seen in Fig 2.2. It makes it easier to envision the subsequence alignments and how they are connected. Since every node is only connected to the nodes in its immediate vicinity and the values of the weights per direction is constant, this grid can be abstracted and implemented with a two

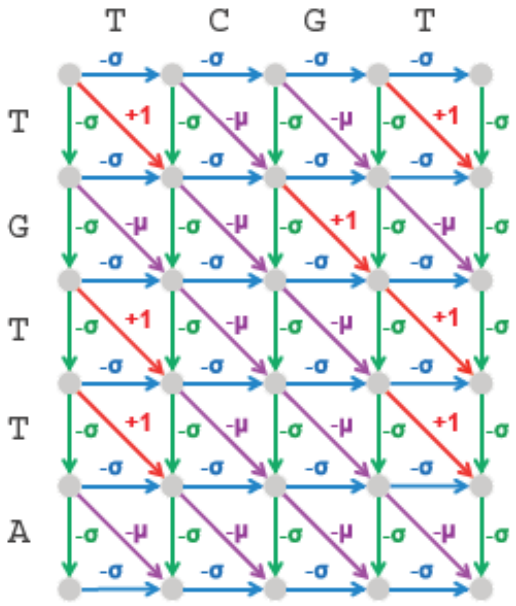


Figure 2.2: Sequence alignment DAG arranged in a two-dimensional matrix [6].

dimensional matrix. Each sequence is assigned to an axis, the reference sequence to the y-axis, and the query sequence to the x-axis. This means apart from the first row and column, every reference base is assigned to a row in sequence order and every query base is assigned to a column in sequence order as depicted in Fig 2.2. For the rest of this thesis we will represent the DAG matrix with S. Every cell in S has a score $S[i, j]$ from which, depending on the algorithm, a potential final alignment score can be picked. For example S[4, 3] will give use the of the optimal alignment between the first 4 bases of the reference and the first 3 bases of the query. In the example on Fig 2.2, this would be the subsequence alignment of TGTT for the reference and TCG for the query. The acyclic in DAG means that paths can only move in one direction per axis, in most examples it means paths can only move side ways to the right, downwards, or diagonally. Different paths across the DAG matrix S represent different alignments of the two sequences, so the optimal path represents the optimal alignment.

A cell's score is determined by looking at the scores of neighboring cells and applying operations to them to find the optimal alignment for itself. There are 4 types of operations,

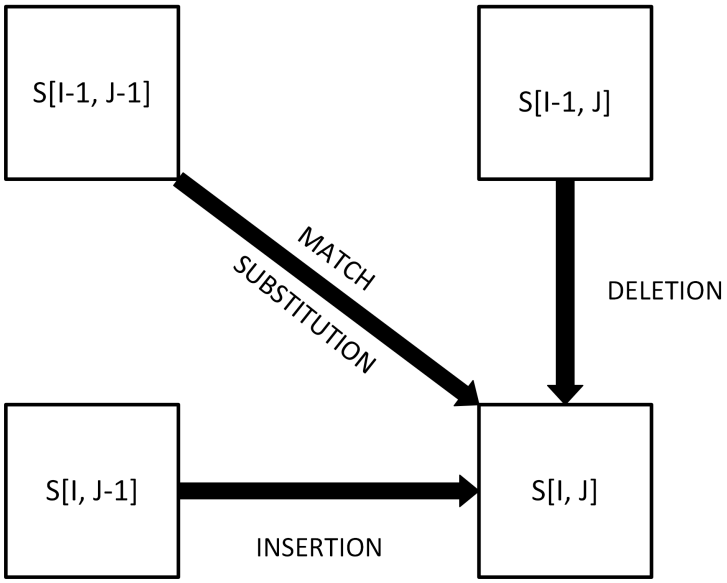


Figure 2.3: A DAG cell's three neighboring cells and their corresponding operations.

matching, mismatching/substitution, insertion and deletion. The objective is to reward for matches and to penalize for the other operations. The reward and penalty values come from the scoring matrix as explained earlier. Due to the acyclic nature of the DAG, every cell has 3 neighbors, the cell to the left S[i, j-1], top left S[i-1, j-1], and top S[i-1, j]. The operations performed on them are insertion, substitution or match, and deletion respectively. For deletion -σ is applied to S[i-1, j] and for insertion -σ is applied to S[i, j-1]. For matching and substitution the reference and query bases associated with S[i,j] are checked. If they match, then +1 is applied to S[i-1, j-1], if not then -μ is applied to S[i-1, j-1]. The cell's score is chosen from the operation that results in the highest score. Deletions are represented with a down arrow, insertion are represented with a right arrow, matches and substitutions are represented with diagonal arrows. This is usually the case for most algorithms, but keep in mind that for edit distance it is the opposite. The best score is seen as the lowest, its scoring matrix reflects this.

As seen in Fig 2.4 2.5 2.6, the first row and column are not assigned to a reference and

query base respectively. This is because they are used to incentivize the algorithms to either give the alignment a "free ride" or constrain it. Depending on the algorithm, the optimal path and its alignment score is chosen from one of the cell's in the DAG. In the following sections the rational behind the location each algorithm looks to determine its final alignment score and path will be explored.

2.2.3 Global Alignment Algorithm

Global / Needleman-Wunsch alignment is one of the most famous dynamic programming alignment algorithms. Its scoring matrix usually consists of -μ and -σ that are equal to -1 and matches are rewarded with +1. It finds the optimal alignment between the whole of the reference sequence and the whole of the query sequence. To ensure that the whole reference and query are taken into account, the algorithm does two things. It forces the optimal path to start from the beginning of both sequences and then only picks alignments that finish at the end of both sequences [5].

The algorithm uses the first row and the first column for the former, and its location of final score for the latter. The first row and column are filled with decreasing scores starting at 0 as shown in Fig 2.4. Since a cells score is related to those of the cells before it, this act forces the algorithm always take the beginning of both sequences into account when looking for an optimal alignment. If the beginning of both sequences does not match then it gives the algorithm the flexibility of perform insert and deletion (indel) operations to assure that they align. For example, if we have a reference sequence of GCTAG, and a query sequence of AGCTA, the clear alignment are the regions with GCTA. The algorithm will allow this alignment but will penalize because indel operations are needed to make start of both sequences align, that is the insertion of an A base to the start of the reference, incurring a penalty of -1. This is preferable to setting every cell in the first row and column to a large

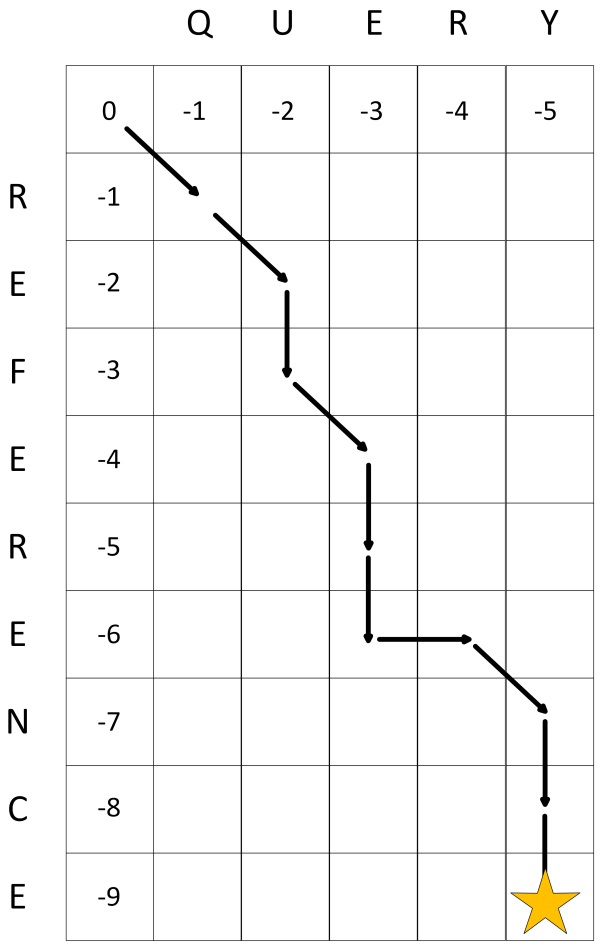


Figure 2.4: An abstract example of the DAG for global alignment. Optimal path starts at the origin and ends at the last cell in the DAG.

negative number like - ∞. That would force the algorithm to only use substitution operations to align the beginning of sequences if they don't match. Going back to the previous example, instead of simply inserting A, the algorithm would need to substitute the first base from G to A and would then have to insert a G next to it to get the same alignment, but now it has incurred a penalty of -2. This over penalization can stop the algorithm from picking the correct alignment as its optimal alignment hence why it is avoided.

The alignment score returned by the algorithm is from the cell at [n, m], n being the length of the reference sequence and m being the length of the query sequence. This is the

cell at the bottom right corner of the DAG (Fig 2.4) and it means that only alignments that include the ending of both sequences will be chosen. This is how the algorithm forces end to end sequence alignment.

Global alignment is not well suited to the primary application considered in this thesis because the reference and query sequences differ substantially in length. Forcing an end-to-end alignment between a long pathogen genome and a much shorter sequencing read would introduce large penalties that do not necessarily reflect the genomic similarity of the sequences.

Algorithm 1: Global Alignment

```
1  Function gloAl(ref, ref_len, qry, qry_len):
2      Declare DAG Matrix S with size [ref_len + 1][qry_len + 1];
3      Declare scoring matrix score() with match ← 1, -μ and -σ ← −1;
4      S[0][0] ← 0;
5      for i = 1 to ref_len do
6          S[i][0] ← −i;
7      for j = 1 to qry_len do
8          S[0][j] ← −j;
9      for i = 1 to ref_len do
10         for j = 1 to qry_len do
11             insertion ← S[i][j − 1] + score(−, qry[j − 1]);
12             deletion ← S[i − 1][j] + score(ref[i − 1], −);
13             matching ← S[i − 1][j − 1] + score(ref[i − 1], qry[j − 1]);
14             S[i][j] ← max(insertion, deletion, matching);
15     return S[ref_len, qry_len];
```

2.2.4 Local Alignment Algorithm

Local / Smith-Waterman alignment is another prominent dynamic programming alignment algorithm. Like global alignment its scoring matrix usually consists of -μ and -σ that are equal to -1 and matches are rewarded with +1. It finds the optimal alignment between

the subsequences of the reference and the subsequences of the query. To achieve this, all alignments are given a "free ride" to start anywhere along the two sequences. Then the final alignment score is picked from the whole of the DAG. These two steps guarantee that the algorithm will only look for the best subsequence to subsequence alignments [5].

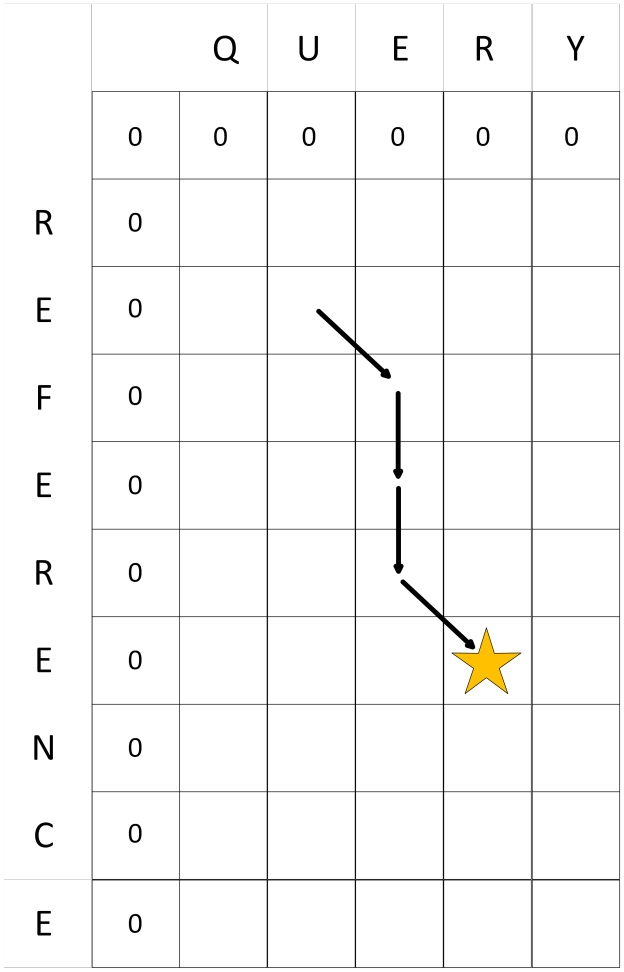


Figure 2.5: an abstract example of the DAG for local alignment. The optimal path can start and end anywhere in the DAG.

To implement the "free rides" The algorithm fills the first row and the first column with scores of zero (Fig 2.5). This means that unlike global alignment, local alignment does not penalize paths for where they begin. Paths starting at [0,0] will start off with the same score as those starting at [0,m], for example. The algorithm then picks the cell with the highest

score across the DAG as the alignment score. Picking a score from anywhere within the DAG means that they is no requirement for optimal alignments include the ending of both sequences. Having the freedom to look for alignments that start and end anywhere in the DAG is how the algorithm finds the optimal subsequence to subsequence alignment.

This algorithm is closer to what was needed for our design than global alignment and came close to being chosen. ultimately, it was not picked because it would have required a more complex hardware design. Another reason stemmed from the fact that incoming queries were already relatively small chunks from longer reads, making them essentially subsequences. So instead of finding optimal alignment of the subsequence of a subsequence, it was seen as more advantageous to take the whole query into account when finding alignments. Finding an algorithm tailor made for this was seen as a better use of resources.

Algorithm 2: local Alignment

```
1  Function locAl(ref, ref_len, qry, qry_len):
2      Declare DAG Matrix S with size [ref_len + 1][qry_len + 1];
3      Declare scoring matrix score() with match ← 1, -μ and -σ ← −1;
4      S[0][0] ← 0;
5      for i = 1 to ref_len do
6          S[i][0] ← 0;
7      for j = 1 to qry_len do
8          S[0][j] ← 0;
9      for i = 1 to ref_len do
10         for j = 1 to qry_len do
11             insertion ← S[i][j − 1] + score(−, qry[j − 1]);
12             deletion ← S[i − 1][j] + score(ref[i − 1], −);
13             matching ← S[i − 1][j − 1] + score(ref[i − 1], qry[j − 1]);
14             S[i][j] ← max(insertion, deletion, matching);
15     return max(S);
```

2.2.5 Fitting Alignment Algorithm

The fitting alignment algorithm also goes by semi-global or glocal alignment, this is because it is a mix of both global and local alignment. It is designed with the objective of identifying the optimal alignment of the whole query sequence within the reference sequence. Like the other two algorithms its scoring matrix uses -μ and -σ that are equal to -1 and matches are rewarded with +1.

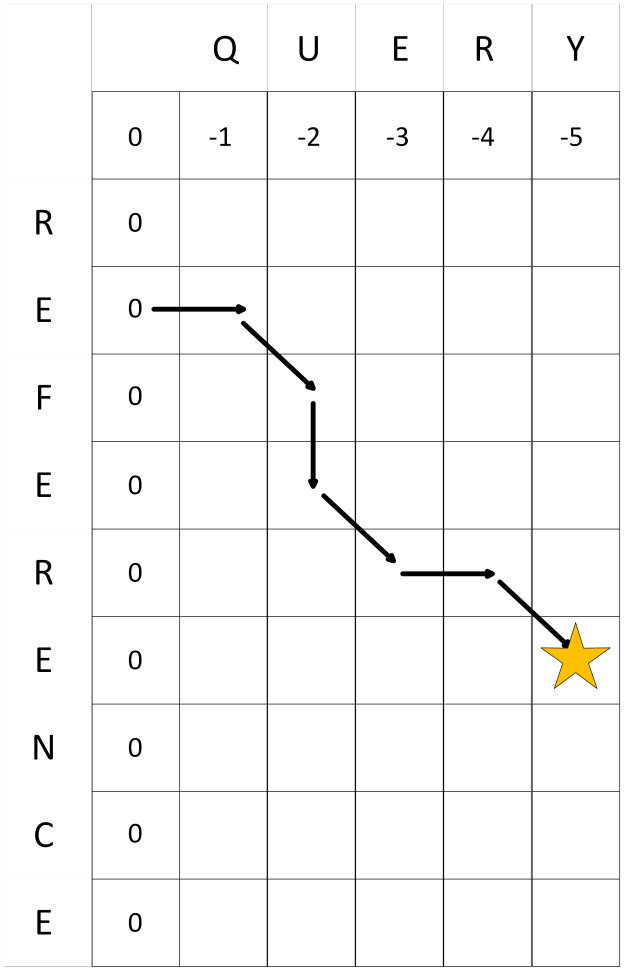


Figure 2.6: an abstract example of the DAG for fitting alignment. Optimal path can start anywhere in the first column and end anywhere in the last column.

To make sure the whole query is accounted for the algorithm populates the first row with

scores of decreasing value exactly like global alignment. Then the final score is chosen from the last column. This forces any optimal path to start at the first column and end at the last column, meaning any optimal alignment must include the whole query. It also populates the first column with scores of zero like with local alignment. This gives all alignments a "free ride" to start anywhere along the reference without penalties(Fig 2.6). The combination of global and local alignment gives us an algorithm that determines how well a query fits into a reference. The rationale behind its selection as the foundation of our virus detection hardware is clear. It is the perfect algorithm for scenarios where the desire is to detect whether a read from the basecaller belongs to a reference genome. A C language implementation of the fitting alignment algorithm is in Appendix A.1.

Algorithm 3: Fitting alignment

Function `fitAl`$(ref, ref_len, qry, qry_len)$**:**
- Declare DAG Matrix S with size $[ref_len + 1][qry_len + 1]$;
- Declare scoring matrix score() with match $\leftarrow 1$, -μ and -$\sigma \leftarrow -1$;
- $S[0][0] \leftarrow 0$;
- **for** $i = 1$ *to* ref_len **do**
  - $S[i][0] \leftarrow 0$;
- **for** $j = 1$ *to* qry_len **do**
  - $S[0][j] \leftarrow -j$;
- **for** $i = 1$ *to* ref_len **do**
  - **for** $j = 1$ *to* qry_len **do**
    - $insertion \leftarrow S[i][j-1] + \text{score}(-, qry[j-1])$;
    - $deletion \leftarrow S[i-1][j] + \text{score}(ref[i-1], -)$;
    - $matching \leftarrow S[i-1][j-1] + \text{score}(ref[i-1], qry[j-1])$;
    - $S[i][j] \leftarrow \max(insertion, deletion, matching)$;
- **return** $\max(S[:, qry_len])$;

2.3 Correctness

Even though the fitting algorithm seems like the best sequence alignment algorithm for our needs, it is important to prove that it is also correct. First we need to establish a loop invariant. The obvious loop invariant here is that for any cell [i, j] in DAG S that is being evaluated, as long as it is not part of the first row or column, it will always have access to the scores of its 3 nearest neighbors $S[i-1, j-1]$, $S[i, j-1]$, and $S[i-1, j]$. also, Before a cell is evaluated, all cells required to compute it contain the optimal fitting alignment scores for their corresponding subproblems. These invariants can be used to prove the correctness of the fitting algorithm via induction.

In the initial step, we verify that the values in the first row and column are correct. Cell [0, 0] has a score of 0, this correct as we are comparing an empty reference sequence to an empty query sequence. There isnt an operation that can make these two sequences align so a score of 0 is correct. For the other cells in the first column, a score of 0 is also appropriate. We are comparing an empty query sequence to an arbitrarily long reference sequence. The fitting alignment by its nature will not penalize for a query being smaller than a reference. Also, there is no set of operations performed on the reference that can get it to align with the query, leaving us with a score of 0. For the other cells in the first row, we are comparing an empty reference sequence to an arbitrarily long query sequence. Unlike before, for this condition there are operations we can apply to the reference to align it with the query. We can do a series of insertions to insert every query base to the end of the reference. According to the scoring matrix each insertion operation will cost -1, so getting a front row where scores are decreasing by -1 every column is correct.

In the inductive step the first row and column are already populated. Since the loop traverses the DAG in row major order, whenever it evaluates an arbitrary cell, the row above it and the column before will have already been computed. This means that any arbitrary cell

in the loop will always have access to the scores of its 3 nearest neighbours, fulfilling the loop invariant. Let [i, j] represent an arbitrary cell in the DAG. We know that the last operation in its optimal alignment can be one of 4 possibilities. Matching, substitution, insertion and deletion. Access to the scores of the 3 nearest neighbours, means that the algorithm already has all the scores of the optimal subsequence alignments in which that last operation relies on. For matching and substitution we have $S[i-1, j-1]$, for insertion we have $S[i, j-1]$, and for insertion we have $S[i-1, j]$. Since we already have all possible optimal subsequence alignment scores, we can assume that by adding the appropriate scoring matrix value for the operations and picking the maximum resultant score that we will find the optimal alignment score for cell [i, j].

Since the size of both query and reference sequences are finite and all paths in the DAG are acyclic, the loop is guaranteed to eventually terminate and end the algorithm. As mentioned earlier, because there are no cyclic paths in the DAG, the fitting alignment problem essentially turns into the longest path in DAG problem, a classic dynamic programming problem. The longest-path problem can be solved efficiently in this setting, making the fitting algorithm not NP-complete. The runtime complexity of the fitting algorithm is O(N*M), where N is the length of the reference and M is the length of the query. The memory complexity depends on the approach. If only the final alignment score is sought-after, then the algorithm only needs to retain the previous row in memory, making it O(M). If seeing the full alignment is sought-after, then the algorithm has to use the traceback process to discover it from the end point. This will require all pointers / direct edges to be stored, increase the memory complexity to O(N*M).

2.4 Alignment Scores and Visualizing

The alignment score precisely measures how well the entire query aligns with the reference genome. A negative score signifies a significant misalignment, while a positive score, equivalent to the length of the query, denotes a perfect alignment. Scores falling in between are assessed on the basis of their proximity to either end of this alignment spectrum.

Alignment visualization is important because it provides a human-interpretable representation of the relationship between two biological sequences. While an alignment algorithm produces a numerical score, the visualization allows researchers to directly inspect where sequences align and differ. This is particularly useful for identifying conserved regions, mutations, substitutions, insertions, deletions, and larger structural differences that may have biological significance.

For pathogen detection, alignment visualization can also provide an important verification step. A classification algorithm may indicate that a read originates from a particular pathogen, but visualizing its alignment against a reference genome allows the researcher to inspect the evidence supporting that classification. Regions containing mutations or gaps can be examined directly, while conserved regions can provide confidence that the sequences are genuinely homologous. Visualization is not only useful for interpreting alignment results, but also for debugging and validating alignment algorithms, since incorrect traceback paths, unexpected gaps, or anomalous alignment regions can be readily identified.

Alignments are visualized in a myriad of ways. The first is that of a path across the DAG as seen in Fig 2.4 2.5 2.6. The path consists of arrows that show the operations needed to convert the query to the reference. As stated earlier, down is for deletion, right is for insertion, and diagonal is for match and substitutions. The second type is a text representation. Fig 4.7 shows it clearly. It lines up the two sequences to do a base to base level representation of the alignment. Indels are shown with "-", which represents a gap. For insertions it is given

to the reference, for deletion it is given to the query. If the bases match then that shows a match operation, if there is a mismatch then it shows a substitution operation. The third type of visualization is through whats known as the CIGAR (Compact Idiosyncratic Gapped Alignment Report) format. It is essentially an encoding of the base level representation as seen in Fig 4.8. This more compact representation of alignment is useful because it can reduce the memory usage demands when storing the alignment of large sequences.

In order to get any sort of alignment visualization, the traceback process is needed. Traceback reconstructs the sequence of matches, substitutions, insertions, and deletions that produced the optimal score. Traceback works backwards, it follows the edges through the DAG from the node at the optimal endpoint to the appropriate starting node on the boundary, recovering the sequence of operations that form the optimal alignment of the two sequences. Traceback in its most naive implementation requires the storage of all edges in the DAG as pointers. For the sequence sizes dealt with in this thesis, this storage need would be too much for our embedded context. To implement traceback our design will need an efficient and effective way to reduce the memory burdens of the process. Traceback and the CIGAR format will be explored more in the traceback chapter (chapter 4).

Chapter 3

System Architecture

3.1 Motivation

a system's architecture should be designed according to the requirements and constraints of its intended operating environment. Understanding the application context of this work is therefore essential for explaining the architectural decisions made throughout this chapter. We envision an operating environment for mobile DNA detection similar to the one described by Dawji et al [51]. The proposed design is intended to be integrated into a RISC-V system-on-chip (SoC) operating at frequencies of up to 200 MHz. Any design addition to this SoC should consume below 100 mW of power. The target platform is a portable, handheld device in which the SoC is integrated alongside a MinION-style nanopore sequencer.

The SoC receives nanopore measurements from the MinION and uses its general-purpose processor, together with specialized accelerators, to convert these measurements into DNA reads through basecalling. The resulting reads may contain up to 1000 bases. These reads are subsequently compared against pathogen reference genomes using the fitting alignment algorithm described in Chapter 2. In this application, each basecalled read acts as the query sequence, while the pathogen genome acts as the reference sequence. The architecture

considered in this work supports reference genomes of up to 50,000 bases.

A real-time pathogen detector must keep up with the throughput of the sequencer. The MinION flow cell has 512 channels [52], each able to measure up to 450 bases per second [53], that is maximum total throughput of 230k bases every second. Various hardware configurations present themselves as viable candidates to fulfill this requirement. The most rudimentary approach entails executing a software implementation of the fitting alignment algorithm (in appendix A.1) running solely on the RISC-V processor.

For a maximum reference length of 50,000 bases and a maximum query length of 1000 bases, the corresponding DAG can contains up to 50 million cells. Assuming, optimistically, that the processor can compute one DAG cell per clock cycle, a processor operating at 200 MHz could evaluate approximately 200 million cells per second. Since processing one query of length 1000 against a 50,000-base reference requires approximately 50 million cell evaluations, the processor could process approximately four complete queries per second, corresponding to approximately 4000 query bases per second.

Compared with the maximum sequencer throughput of approximately 230,000 bases per second, this represents less than 2% of the required processing rate. Under these assumptions, approximately 58 such processing cores operating in parallel would be required to match the throughput of the MinION. A processor-only implementation is therefore insufficient for the real-time requirements of the proposed system and demonstrates the need for dedicated hardware acceleration.

One of the most popular forms of hardware accelerators are GPUs. An alternative avenue to our setup would involve the integration of a GPU alongside the CPU, and delegate the processing of the DAG to it. GPUs are highly effective at executing massively parallel computations and are well suited to processing two-dimensional computational structures such as the DAG matrix. A sufficiently capable GPU could provide enough computational throughput to match the MinION. However, this strategy presents an inherent problem. As a

trade off for its extra co-computing capabilities, GPUs exhibit heightened power consumption thereby diverging from the low-power imperative of our design.

The proposed solution is therefore a dedicated custom co-processing accelerator attached to the RISC-V processor and specifically optimized to compute the fitting alignment algorithm. The name of this accelerator is KAOS. As established previously, approximately 58 DAG cells must be evaluated concurrently to achieve the required throughput under the assumed operating conditions. Consequently, a systolic array containing 64 processing elements (PEs) was selected as the computational core of the KAOS accelerator. This configuration provides sufficient parallelism to meet the estimated throughput requirement while maintaining a specialized and power-efficient architecture.

3.2 System Components

The SoC is made up of various components that facilitate data flow and rapid processing for pathogen detection. For our circumstance, the four most important components are memory, processor, Rocket Custom Coprocessor (RoCC) interface and KAOS. Here memory is used as a catch all term for the memory hierarchy. Whether that is registers, caches or DRAM, they store basecalled DNA sequences needed for computations. The Rocket core [54] is a five-stage, in order processor that uses the RV64GC RISC-V instruction set architecture(ISA). It is the general purpose processing hub of the SoC. It is responsible for running the Linux operating system and other user made software. Through custom ISA instructions, known as RoCC instructions, it also coordinates communications with KAOS via the RoCC interface. The RoCC interface not only facilitates communication between the accelerator and the core, but also between the accelerator and the memory systems. Software executing on the Rocket core can issue custom RoCC instructions to initiate computations, receive alignment results, and provide input parameters and memory addresses to KAOS. The interface also parses

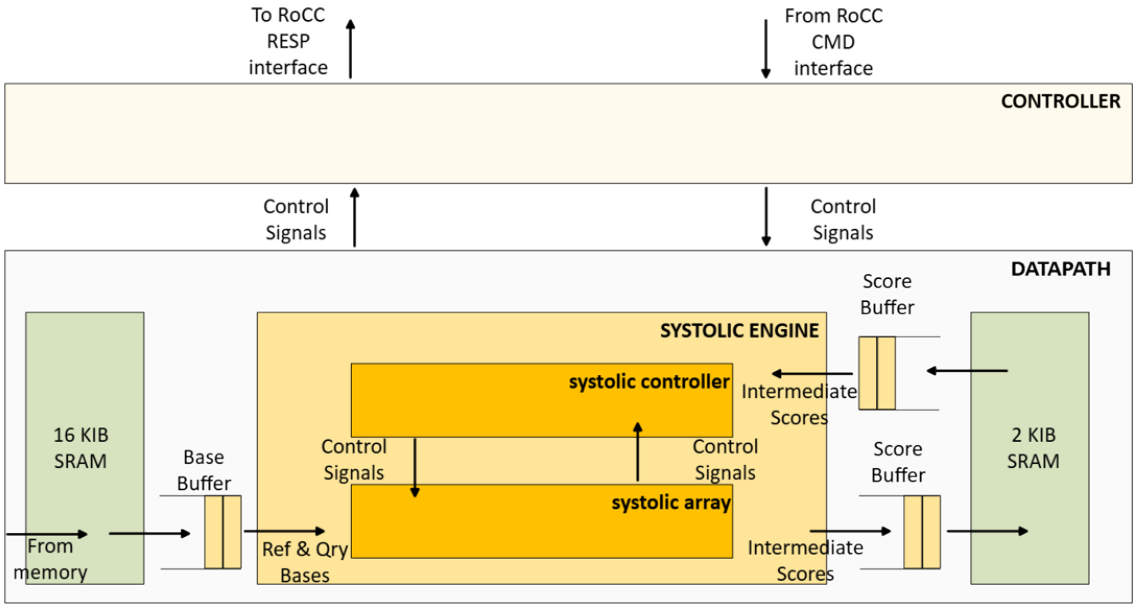


Figure 3.1: Basic KAOS Architecture showing dataflow between the controller and datapath. Data flow is shown with arrows.

these RoCC instructions for KAOS, as seen in Fig 3.2.

At the highest level the KAOS accelerator consists of a controller and datapath(Fig 3.1). The controller decodes RoCC instructions and directs data flow towards the appropriate segments of the datapath using control signals. The datapath is made up of SRAMs, FIFO (first-in, first-out) buffers, and a systolic engine. There are two SRAMs, a 16-KB SRAM for storing DNA sequences named the data SRAM, and a 2-KB SRAM for storing scores called the score SRAM. Four FIFO buffers connect these SRAMs to the systolic engine. A FIFO buffer is needed for each of the following: feeding query data to the systolic engine, feeding reference data to the systolic engine, inputting scores from the systolic engine into the score SRAM, and inputting scores from the score SRAM back into the systolic engine. The systolic engine is a combination of the systolic array and a control unit that implements finite state machines (FSM).

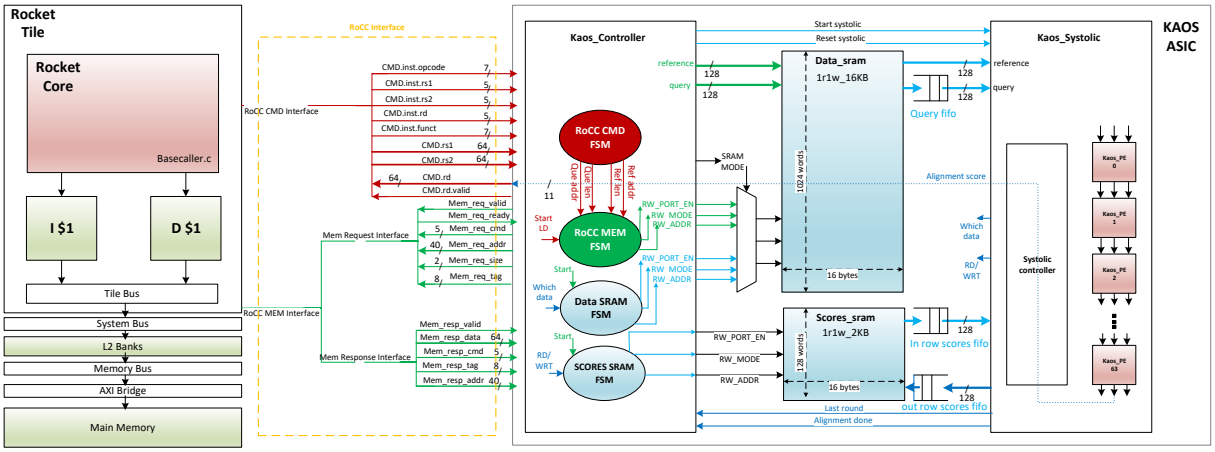


Figure 3.2: SoC architecture showing the flow of data between the core, memory components, RoCC interface and KAOS.

3.2.1 Data Flow

Sequence alignment is initiated by software, in this case a C program being executed by the Rocket core. The program uses RoCC instructions to give KAOS pertinent information such as the memory address of the query, the memory address of the reference, size of query, size of reference and which register should be used to store the returned alignment score. Once KAOS receives all necessary information it immediately starts sending memory requests for DNA sequence data. Sequence data is sent in 64-bit chunks and are stored in the data SRAM. Once all requested data has been received, query data and reference data are fed to the systolic engine via their separate FIFO buffers. The controller then signals the systolic engine to start processing, it will only stop when all of the DAG has been processed or if stall conditions are met. During the processing of the DAG, certain scores are stored in the score SRAM, they are then retrieved later when needed. At the end the systolic engine signals the controller and the alignment score is sent back to the core via the RoCC interface.

3.3 Systolic Design

3.3.1 Processing Element Architecture

The systolic array is comprised of 64 processing elements (PE). The objective of a PE is to compute the score of one cell of the DAG every clock cycle. Specifically, it computes the segment of the fitting algorithm inside the nested loops. That is, it computes all operations, match, substitution, insertion, and deletion, and picks the one that leads to the maximum score. The inner workings of every PE is shown in Fig 3.3.

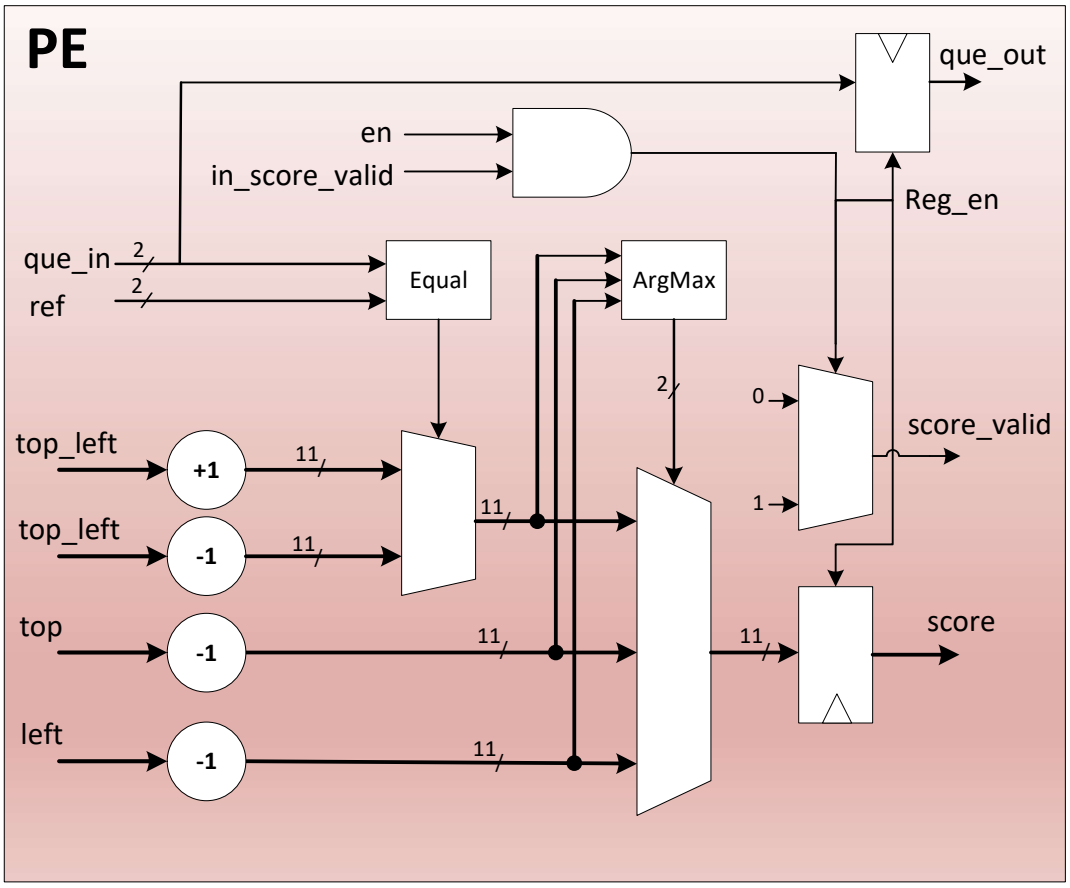


Figure 3.3: Logic inside the processing elements. A Verilog implementation is in Appendix A.3.

Every PE has input ports for the scores of the three neighbouring cells, left, top_left, and top. These inputs are 11 bits each because that is the number of bits needed to capture all possible scores. As Fig 3.3 shows, these inputs immediately have their corresponding scoring matrix values applied to them. This is done through adders that add 1 to top_left for matching, -1 to top_left for substitution, -1 to top for deletion, and -1 to left for insertion.

There are also input ports for the query and reference bases that correspond to the cell's coordinate in the DAG. Each has 2 bits because the 4 nucleotide bases of DNA can be encoded with 2 bits. According to the fitting algorithm, only one of the matching or substitute operations can be considered depending on if the query and reference bases match. An equality comparator is used to see if the bases match and the result is used as the select for a 2:1 multiplexer (MuX) to chose the appropriate operation. Next the resulting operation and the indel operations will have their scores compared by the Arg-Max module. It will pick the index of the highest score and use it as a select for a 3:1 MuX which picks the correct answer as the score for the cell.

A score is only allowed to be output if it is valid. A score is valid if and only if the PE has been enabled by the systolic engine's control unit, using the en (enable) signal, and if the scores coming from the previous PE in the array are also valid indicated by the in_score_valid signal. The PE circuitry does this with an AND logic gate. If both the en and in_score_valid inputs signals are high then the score will be outputted via a register. The other register is used to allow the query base to propagate to the next PE. Lastly, the output of the AND gate is used as the in_score_valid input of the next PE.

3.3.2 Systolic Array Architecture

The previous subsection showed how a PE can be used to process a cell in the DAG. This subsection will show how multiple PEs can be connected in an array in order to process up

to 64 cells in parallel every clock cycle. When processing the DAG each PE is assigned to an adjacent row. Since the PE is going to process the entire row it is assigned the reference base that corresponds to that row. However, it needs to be given a new query base every cycle, one for every column in the row. This is done using the query propagation mentioned in the previous section. The query bases are fed via a shift register to the first PE, one at a time, in order, one every cycle. The query bases then propagate through the systolic array and each PE gets fed the appropriate base for the column it is processing in that cycle. This propagation is facilitated by a couple of registers between each PE, as can be seen in Fig 3.4.

There are other registers between the PEs. One is for the valid score signal that was explained in the previous subsection. The other two are for the score. It has been established that a cell determines its score from the score of the three neighboring cells, to the left, top left, and top. The cell to left is on the same row so its score is computed by the PE in the previous clock cycle. This explains the loop in the architecture that connects back to the left input port via a register. The other two scores come from the preceding PE in the systolic array. The top input is its score from the previous clock cycle and the top left input is its score from two cycles earlier. This is facilitated by a second register, it allows for the storage of the scores from the preceding PE from the last two cycles.

In the fitting algorithm the cells in the first column all have scores of zero. Since the PEs start processing a row from the second column, it needs to use these zero scores as input for the left and top left input ports specifically. To implement this, these ports are connected to reset registers, and the reset signal is set by the control unit. In the Fig 3.4 the signal is called round_reset. When the round reset signal is on, the values of the reset registers are set to zero and this is fed to the left and top left ports.

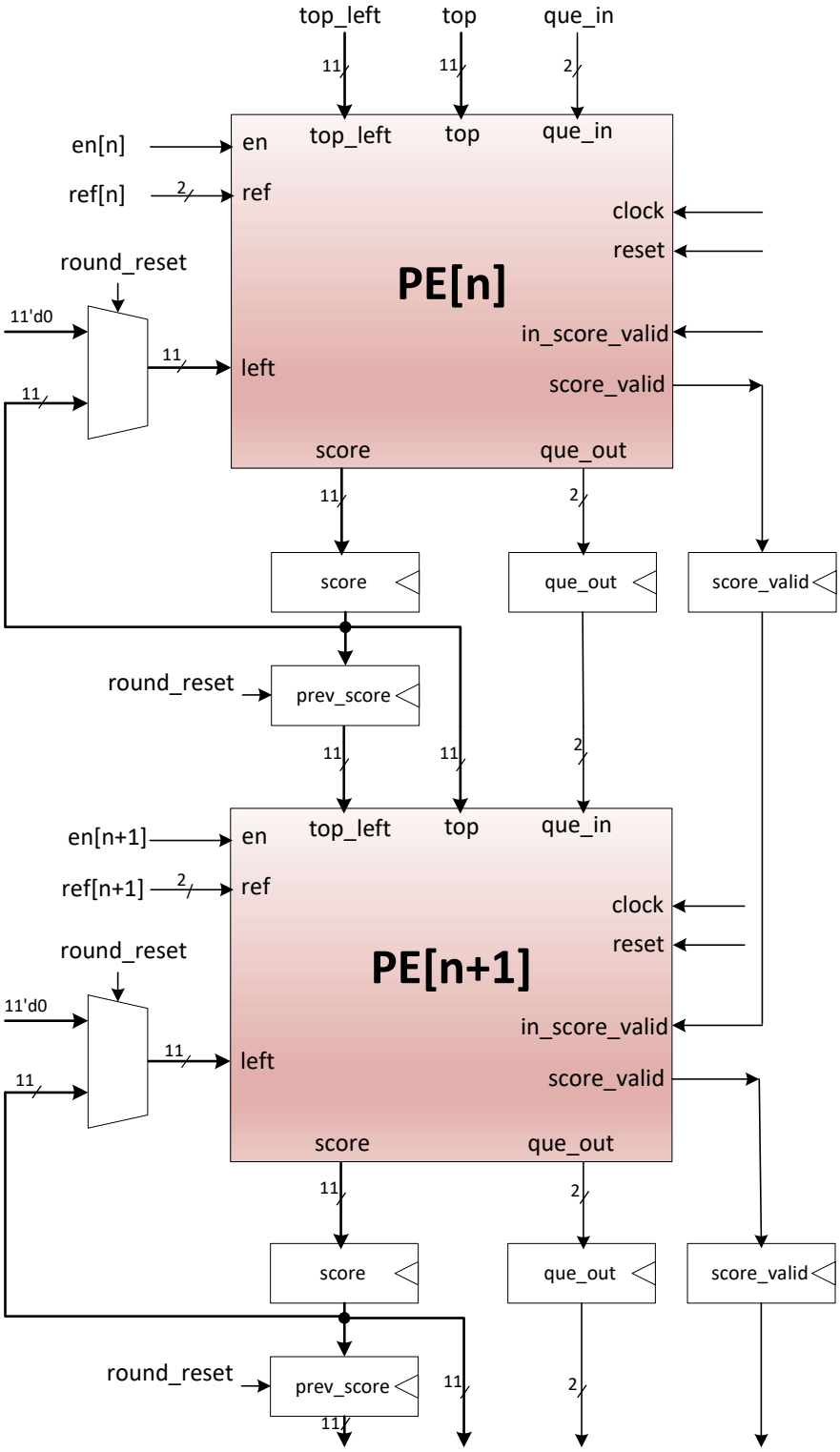


Figure 3.4: Architecture of the systolic array.

3.4 Algorithm Implementation

This section will give an in depth look at how the systolic architecture is used to process the DAG. There are 64 PEs, so only 64 rows can be processed at a time. Since the DAG can have up to 50,000 rows, these rows need to be split into groups. These groups are referred to as rounds, and each round will have 64 rows, except for the last round which may have fewer rows. A round is processed in a wavefront parallel manner, that means at any clock cycle the execution of columns is staggered between adjacent PEs (Fig 3.5).

Let $PE[i]$ present the ith PE in the systolic array. At the beginning of DAG processing, the only scores that are known are those of the first column and the first row. That means $S[1,1]$ is the only cell that can be processed in clock cycle 1 because it is the only cell where the scores of its 3 neighbours ($S[0,0]$, $S[0,1]$, and $S[1,0]$) are known. $PE[1]$ processes $S[1,1]$ in clock cycle 1 and sends the query base it used to $PE[2]$ before the next clock cycle. Now there are two cells that can be processed in cycle 2, $S[1,2]$ and $S[2,1]$. $PE[1]$ processes $S[1,2]$, while $PE[2]$ processes $S[2,1]$ and sends the query base to $PE[3]$. In cycle 3, $PE[1]$ processes $S[1,3]$, $PE[2]$ processes $S[2,2]$, and $PE[3]$ processes $S[3,1]$. This continues until all PEs are processing in parallel, each PE one column ahead of the next.

When $PE[1]$ reaches the end of its assigned row, it must transition to the first row of the next round. A naive implementation would require $PE[1]$ to wait for the remaining PEs to complete their rows. Since the array contains 64 PEs, this could cause $PE[1]$ to remain idle for up to 63 cycles during each transition. Across a complete alignment, this could result in as many as 49,203 idle cycles. This is unacceptable. To avoid this loss of utilization, intermediate scores are stored in the score SRAM.

For $PE[1]$ to start processing the 2nd round (row 65) immediately it needs the score $S[64,1]$ as its top input. This score was calculated by $PE[64]$ at cycle 64 and could have been calculated up to 937 cycles before $PE[1]$ needs it. Such information cannot be retained

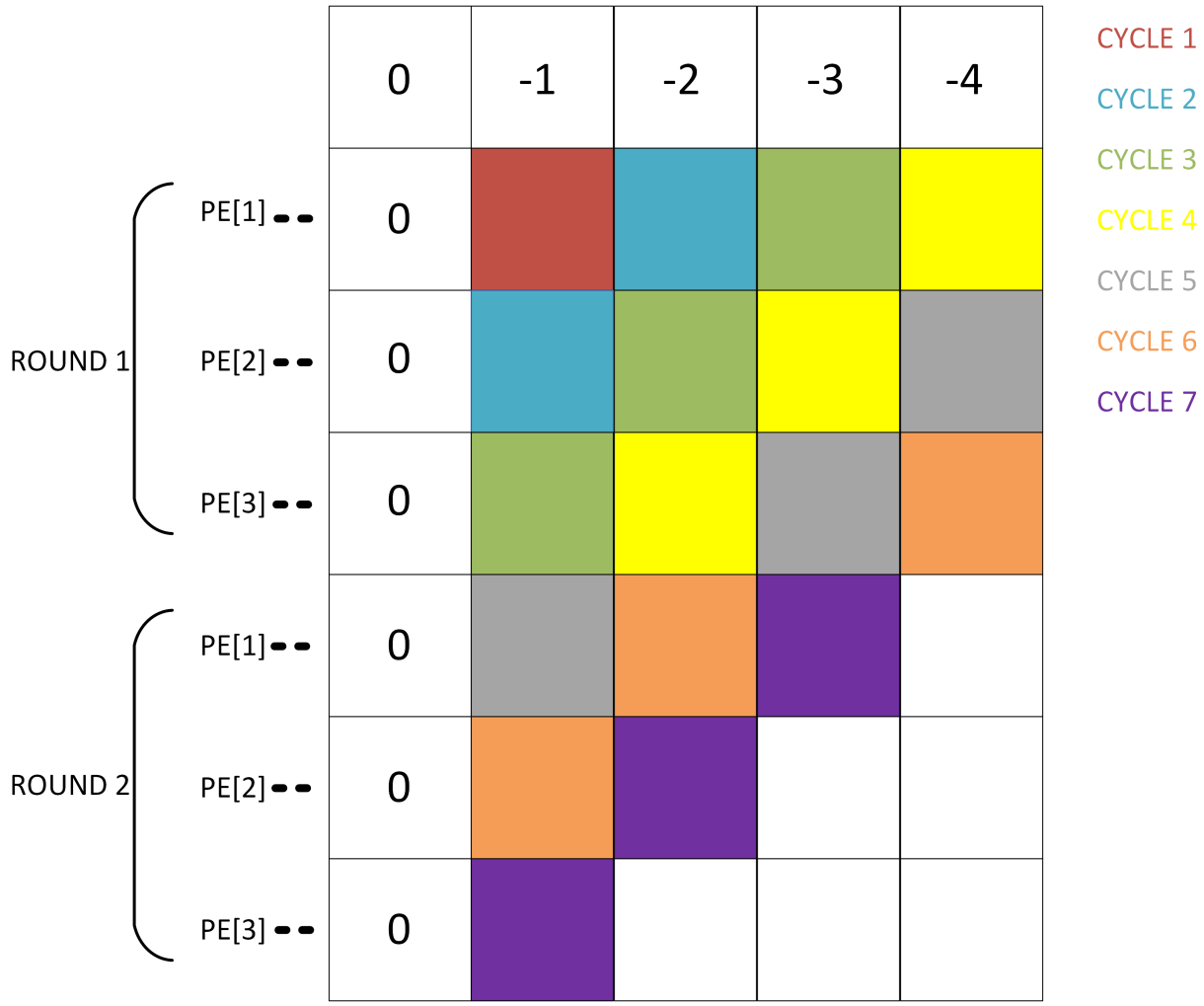


Figure 3.5: Cycle by cycle processing of a miniature version of the DAG by the systolic array. The DAG has a query with 4 bases, reference with 6 bases, and is processed by a systolic array with 3 PEs. The figure shows the wavefront parallel manner of computation. Also shows how PEs transition from one round to another.

indefinitely using only the registers located between neighboring PEs. This is where the score SRAM plays a critical role. In all rounds except for the last, the scores of the last PE ($PE[64]$) are packaged, 11 at a time, and stored in the score SRAM. These scores are referred to as intermediate scores. These intermediate scores are then called back from the score SRAM and fed to $PE[1]$'s top and top left ports so that it can seamlessly transition into processing the next round. This is how multiple rounds are stitched together without

needing to stall PEs in the systolic array.

However, there are certain conditions that do require stalling the systolic array. The first is if any data, whether query, reference, or intermediate score does not get supplied quickly enough to the systolic array to sustain computation. In this case, execution is paused until the required data becomes available. The systolic array will stall until that data clears the buffers. The second condition occurs when the FIFO buffer used to transfer intermediate scores to the score SRAM is full but another package from the systolic array is ready to be entered into the buffer. The systolic array will stall until sufficient space becomes available in the FIFO buffer again. The last condition occurs when the length of the query is smaller than the number of PEs. This creates a scenario where $PE[1]$ has finished processing a row but the last PE has not started processing its row. This means there are no intermediate scores for $PE[1]$ to immediately use to transition to the next round. A "rolling" stall is initiated and PEs are stalled one after the other as they reach the end of their rows. This is to give the last PE enough time to accumulate enough scores to send to the score SRAM. After that the stall is lifted and normal processing resumes. These stall conditions are constantly being checked by the control unit and PEs are stalled by setting their enable (en) input to 0.

The algorithm determines the final fitting score by finding the largest score in the final column. Our systolic architecture implements this by connecting all PEs to a comparator. When a PE reaches the end of a round, the last score it produces is the final column score for its row. The comparator takes this score and compares it to the current largest score. If it is larger, then it becomes the new largest score. At the end of the computations, the largest score from the comparator is the final alignment score that is returned to the processing core via the RoCC interface.

3.5 KAOS Configurations

What has been described so far in this chapter has been the basic version of KAOS, also known as KAOS_B. The following are versions of KAOS that build on the original design for different objectives. This section will show how each version modified the design of the base KAOS to achieve these objectives.

3.5.1 KAOS_C

In order to improve throughput, the number of processing elements needs to be scaled up. The original KAOS design was implemented at the register-transfer level (RTL) using Verilog. Verilog is a very popular hardware descriptive language. Although the base implementation contained 64 processing elements, the original RTL structure did not support convenient scaling. KAOS_C is a modification to the RTL design that allows the number of processing elements to be scaled to multiples of 64. The number of PEs is specified through Verilog parameters, allowing the architecture to be configured according to the required throughput and available hardware resources. As discussed in the results chapter (chapter 5), increasing the number of PEs increases the available computational parallelism and improves throughput approximately linearly, subject to system-level limitations such as memory bandwidth and control overhead. This configuration future-proofs our design, allowing it to be used with sequencers that are faster than the current MinION.

3.5.2 KAOS_Q

KAOS_Q was created to perform multiple sequence alignments at the same time. This version takes the original KAOS design and turns it into a KAOS engine. KAOS_Q has 4 of these engines to allow up to 4 different sequence alignments to be computed at the same time. An additional controller is included to regulate the engines and conduct data flow of

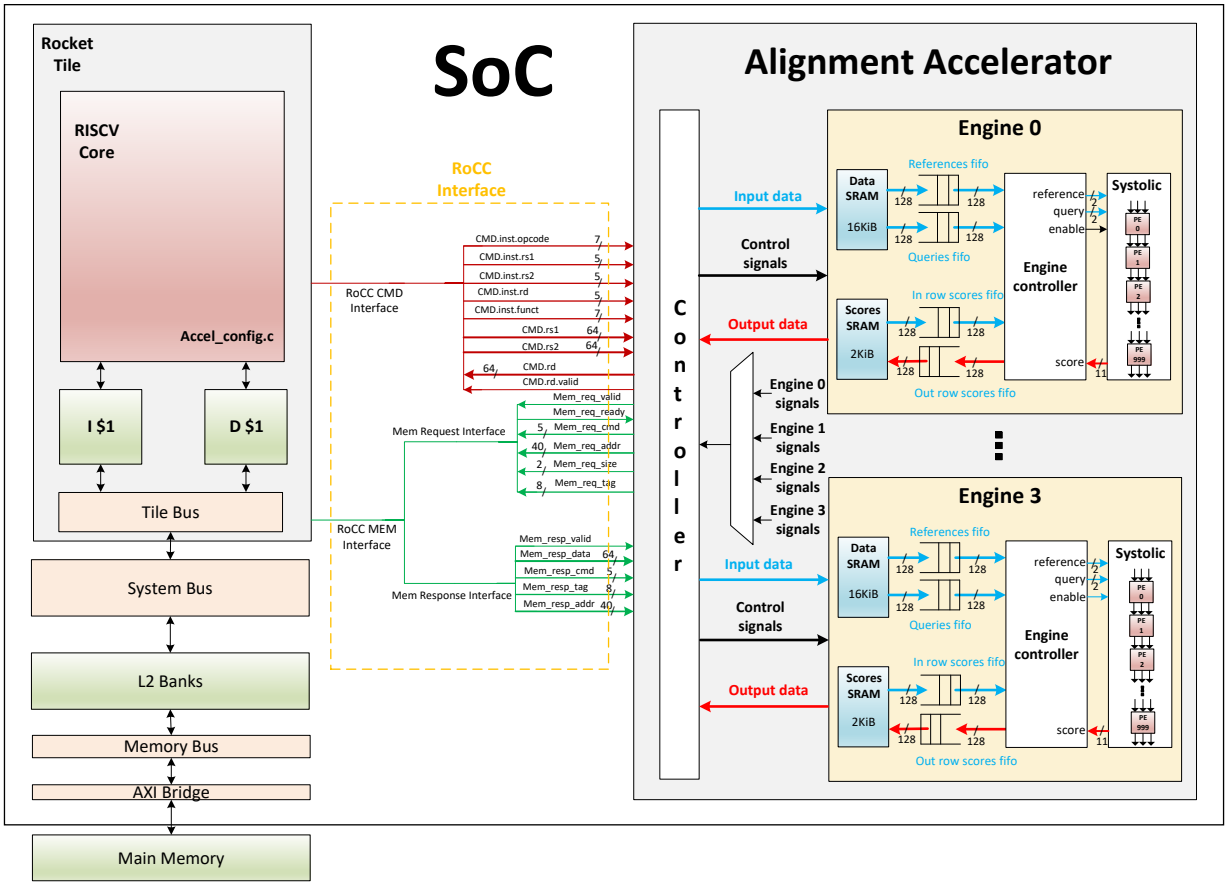


Figure 3.6: The SoC architecture for KAOS_Q.

sequences to each engine through a series of MuXes. The SoC design of KAOS_Q can be seen in Fig 3.6. Scores from all engines are packaged together and sent back to the core when all engines are done. This version can be combined with KAOS_C to create KAOS_QC. A version which the number of PEs in each engine can be scaled.

3.5.3 KAOS_T

This version of the KAOS is designed to help facilitate traceback. Traceback is the act of tracing back the optimal path from its endpoint and visualizing it. To find these endpoints, KAOS was modified to keep track of the index of the cell in the final column from which

the final alignment score is acquired. This index is then returned back to the core alongside the alignment score. Software running on the core uses the index and alignment score to reconstruct the optimal path. This design also borrows from the KAOS_Q design as it has 2 engines. More on this in the traceback chapter (chapter 4).

3.6 Total Memory Requirements

The memory requirements of our accelerator depend on the selected configuration. The majority of this chapter has been describing the smallest configuration, a single KAOS engine with 64 processing elements. However, our design can go up to four KAOS engines each with 1024 processing elements. Because the memory requirements vary significantly between configurations, this section derives a scalable expression for estimating the total memory required by a particular KAOS configuration. There are two things that scale in our design, the number of engines and the number of processing elements within those engines. We can go up to four engines and the number of processing elements are multiples of 64 going up to 1024.

Let's start with a single engine. There are certain memory components that remain the same for every engine. For queries there will always be a 32 B, two entry FIFO buffer and 16 B shift register that feeds the first processing element. The same is true for incoming intermediate scores from the SRAM. Outgoing intermediate scores have the same memory needs but just in reverse, a 16 B register feeding a 32 B two entry FIFO buffer to store packages in the score SRAM. These components use a total of 144 B of memory. The other components that have a constant memory value are the score and data SRAMs. The maximum supported input consists of a 50,000-base reference sequence and a 1000-base query sequence. Because each DNA base is represented using two bits, a 16 KB SRAM is the smallest SRAM available which could hold both sequences. The maximum number of

intermediate scores required per round is 1000, each score is 11 bits. The score SRAM entries each have 16 B, so an entry can only fit 11 intermediate scores. We would need 91 entries, or 1456 B to store all intermediate scores needed in a round. The smallest available SRAM was 2 KB, giving us a total of 18 KB worth of SRAM needed in each engine, and a grand total of 18.144 KB needed for components so far.

Lets discuss the memory components that scale with the number of processing elements. Since the number of processing elements are multiples of 64, lets look at the base situation with 64 PEs because we can always multiply the memory values as the number of PEs scale up. For the reference, the FIFO buffer needs 16 B of memory. Each reference base data is then sent to a PE via a 2 bit register, giving us 16 B worth of registers. In total, 32 B is needed to facilitate reference data movement. Then we have the registers between PEs that facilitate the movement of scores, queries, and valid signals. Refer to Fig 3.4, there are two 11 bit registers that allow a PE to send its score output to its left input port and to the top and top_left ports of the next PE. There is also a 2 bit register between the processing elements that allows query bases to propagate between them. Lastly, there is a 1 bit register that allows for the propagation of the score_valid signal. This means that between each PE we will need 25 bits of memory in order to function, for all 64 PEs that is a total of 200 bytes. If we add this to the memory needed for the reference then we get 232 bytes for every 64 processing elements.

Let Ne represent the number of engines, let Npe represent the number of processing elements. The equation below returns the amount of memory needed for a design configuration in bytes. This gives us a minimum of 18.4 KB required for the single engine 64 PE configuration and a maximum of 76.3 KB required for the four engine 1024 PE configuration.

$$Memory = Ne * (18144 + (232 * Npe)/64)$$

Chapter 4

Traceback

4.1 Traceback Overview

A visual representation of sequence alignment is important for the interpretation, analyses, and validation of bioinformatics results. Previous chapters have focused primarily on the computation of alignment scores, which quantify the similarity between a query sequence and a reference sequence. This chapter describes how the system architecture described in the preceding chapters can be modified and built upon to reconstruct and visualize the optimal alignment of the reference and query. Alignment reconstruction requires the traceback process. Traceback traverses an optimal path through the DAG in reverse, beginning at the endpoint and continuing toward the starting point. Each step along this path corresponds to an alignment operation, such as a match, substitution, insertion, or deletion. Consequently, the recovered sequence of operations can be used to construct a base-level representation of the alignment between the two sequences [55]. Although traceback is conceptually straightforward, storing the information required to reconstruct an alignment can be memory intensive, especially for low resource embedded systems.

The proposed system is designed to process reference sequences of up to 50,000 bases

and query sequences of up to 1,000 bases. The corresponding DAG matrix may therefore contain up to 50 million cells. Every step in the traceback process is one of the four possible operations. If the operation associated with each cell is encoded using two bits, storing traceback information for the complete matrix would require approximately 100 million bits, or 12.5 MB, per sequence alignment. Such a memory requirement is undesirable for a resource-constrained embedded accelerator because of the associated area and implementation costs.

However, in the pathogen-detection application considered in this work, the optimal fitting alignment of a query to a reference typically occupies only a relatively small region of the complete reference sequence. If the boundaries of this alignment region can be identified, traceback can be restricted to a smaller sub-DAG rather than requiring storage of information for the entire DAG matrix. This approach provides a substantial opportunity to reduce the memory requirements of alignment reconstruction.

4.2 Finding the sub-DAG

To restrict traceback to the relevant region of the DAG matrix, the start and end coordinates of the optimal alignment must first be determined. In fitting alignment, the optimal alignment score is selected from the final column of the DAG. Therefore, the endpoint of the alignment can be determined by recording the row index associated with the maximum score in that column. Determining the starting position is slightly trickier. The start coordinate can be inferred by performing an equivalent alignment on the reversed version of the reference and query sequences. The endpoint of the optimal alignment in the reversed problem corresponds to the starting position of the alignment in the original sequence orientation. By transforming this coordinate relative to the reference length, the starting index of the original alignment can be recovered.

Knowledge of the start and end coordinates allows for the construction of a sub-DAG containing the region relevant to the optimal alignment path. Since the maximum query size in our context is 1000 bases, the maximum number of each operation in the optimal path is also constricted to 1000. This means that in the extreme cases the optimal path can have 1000 movements rightward, 1000 movements downward, and 1000 movements diagonally. This bounds the maximum size of a sub-DAG to 2 million cells, 4% of the maximum size of the original DAG. Restricting traceback to this region eliminates the need to store information for 96% the original DAG, this portion does not contribute to recovering the optimal alignment. This reduction in memory requirements introduces additional computational and runtime cost because the alignment process must be performed in both the forward and reversed sequence orientations to determine the two boundaries of the sub-DAG.

Relatively minor modifications to the KAOS architecture can be used to return the indices associated with the optimal alignment boundaries. A two-engine KAOS_Q configuration can execute the forward and reversed alignment computations in parallel, allowing the start and end coordinates of the relevant sub-DAG to be determined with reduced latency. This approach leverages the parallel processing capability of the accelerator to reduce the runtime hit for finding the sub-DAG.

4.3 Divide-and-Conquer

A maximum of two million cells means that 500 kilobytes are needed to store this sub-DAG. Even after restricting the problem to the sub-DAG, storing traceback information for every cell exceeds the amount of memory available locally within the accelerator. There are multiple techniques that can further reduce memory use, but the most effective option that optimizes the balance between memory and runtime is a technique known as divide-and-conquer [5].The divide-and-conquer method considered is best suited for global alignment because it requires

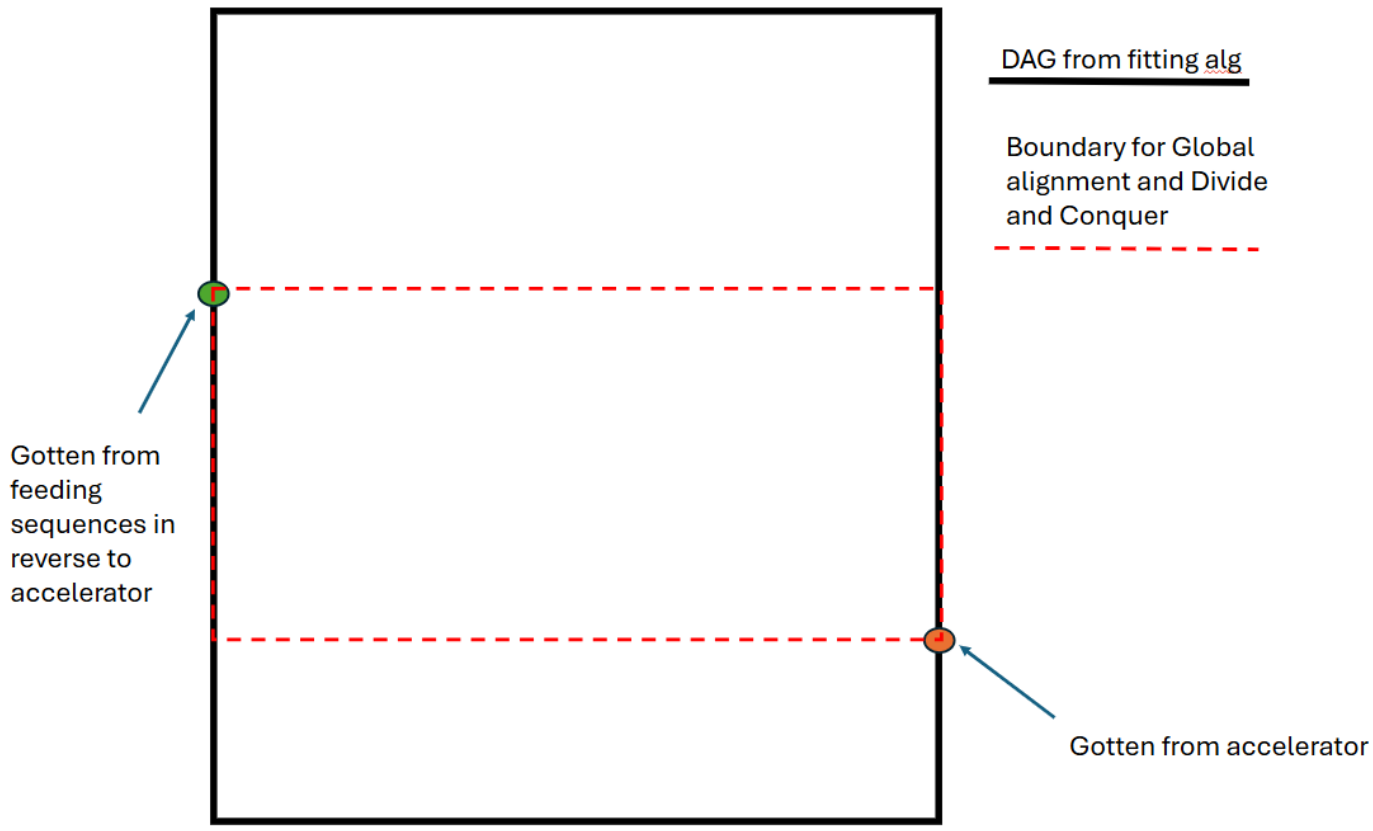


Figure 4.1: Finding the sub-DAG from the original. The sub-DAG is in red, original in black.

predetermined source and sink coordinates for the optimal alignment path [55]. In global alignment, all valid alignment paths begin at coordinate $(0, 0)$ and terminate at coordinate (n, m), where n denotes the reference length and m denotes the query length.

Divide-and-conquer identifies the node on the optimal alignment path that lies within the middle column of the DAG. This is accomplished by calculating the optimal global alignment scores from the source to each candidate node in the middle column and combining them with the corresponding optimal global alignment scores from each candidate node to the sink. The candidate node with the highest combined score lies on the optimal alignment path. The original problem can then be divided into two smaller subproblems, which are processed recursively until the complete alignment path has been reconstructed.

This approach reduces the memory requirements associated with traceback from quadratic to linear complexity ($O(m)$). Rather than storing traceback information for every cell in the complete DP matrix, only the information required to identify the recursive path partitioning points must be retained. In this case, only the row index of the optimal node in column needs to be stored, each will be a 16 bit integer. Since the query is capped at 1000 bases, there are at most 1000 columns and we only need a maximum of 2 kilobytes of memory per sequence alignment.

The sub-DAG identified by the KAOS_Q accelerator can be treated as a global alignment problem because its source and sink coordinates correspond to the previously determined start and end points of the optimal fitting alignment path. This enables divide-and-Conquer to be applied to the reduced problem, despite the original alignment having been computed using the fitting algorithm.

Unfortunately, implementing Divide-and-conquer in our accelerators would mean wholesale changes to our design. It would drastically increase the complexity of the design and the time needed for brainstorming, implementing and testing wouldn't comply with time constraints for this thesis. So it was decided to use the RISC-V processor to implement Divide-and-Conquer

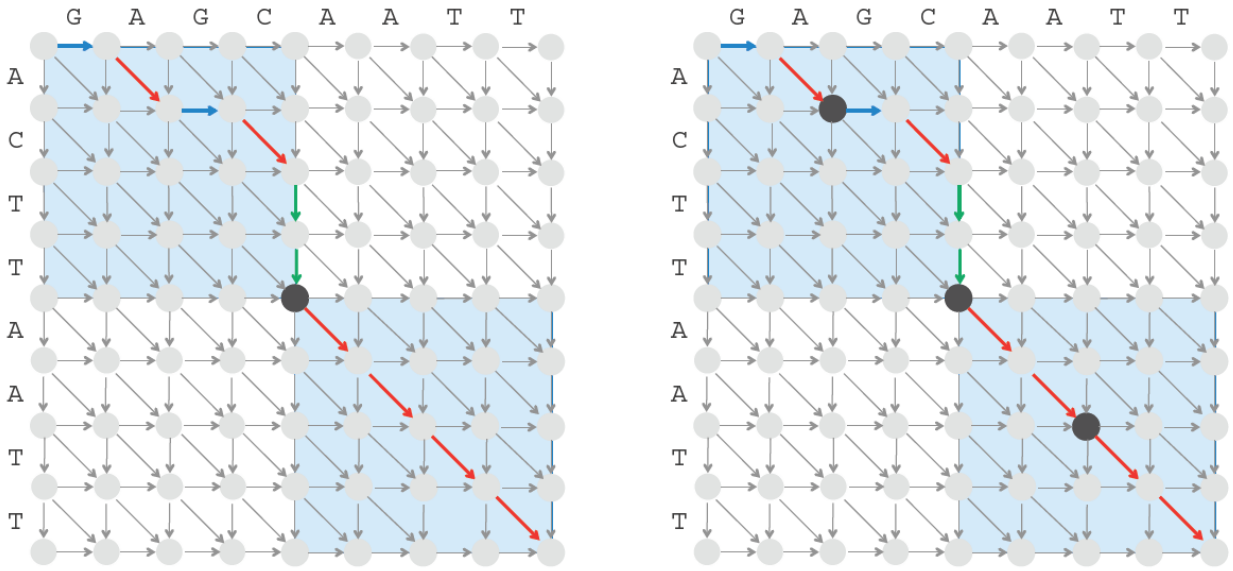


Figure 4.2: Divide-and-Conquer steps used to find and weave optimal nodes into an alignment path. The optimal node from middle column is used to split the DAG into a smaller DAGs shaded in blue [5].

via software. Traceback will still be accelerated by our accelerators since it quickly finds the start and end points of the optimal alignment and eliminates 96% of the DAG problem space. The processor then takes the resulting global alignment problem and applies divide-and-Conquer to return an alignment in a memory and runtime efficient manner. This approach also enriches the hardware software co-design aspect of our design. C implementation of the global alignment algorithm can be found in Appendix A.2.

The Divide-and-Conquer algorithm finds the optimal node in every column and then uses pointers to connect the nodes and weave the optimal path. Since the path always starts at the first index of the first column and ends at the last index in the last column, Divide-and-Conquer only needs to be called for DAGs with 3 or more columns. Every recursive call of divide-and-conquer halves the relevant problem space as shown in Fig 4.2. This continues until the base case of less than 3 columns is reached. At the end, a list of the row index of every column's optimal node will be returned. This process is shown in Algorithm 4.

Algorithm 4: Divide-and-Conquer

Input: DAG S
Output: List of optimal node indices along the optimal alignment path

```
1  Function DivNCon(S):
2      if Number of Columns < 3 then
3          return Empty List
4      Find middle column m;
5      for i ← 0 to |m| do
6          scr_src i ← GlobalAlignFromSRC(m[i]);
7          scr_sink i ← GlobalAlignToSINK(m[i]);
8          scr ← scr_src i + scr_sink i;
9          if scr > scr_max then
10             m_max ← m[i];
11     Sm_max L ← Portion of DAG before m_max;
12     Sm_max R ← Portion of DAG after m_max;
13     L ← DivNCon(Sm_max L);
14     R ← DivNCon(Sm_max R);
15     return AppendList(L, m_max, R);
```

4.4 Pointers

Once the optimal nodes have been identified, the optimal alignment path can be reconstructed by establishing pointers between consecutive nodes. These pointers describe the transitions between nodes and can subsequently be used to determine the sequence of alignment operations required to construct the optimal alignment. Since the list of optimal nodes is ordered according to the columns of the DAG, only the row index of the optimal node in each column must be stored. The pointers connecting optimal nodes in adjacent columns can therefore be determined by comparing their corresponding row indices. Four possible cases must be considered.

Let L denote the list of optimal nodes, let I represent an arbitrary index in L such that $I > 0$, and let $L[I]$ denote the row index of the optimal node associated with column I. If $L[I] = L[I-1]$ then we have a lateral pointer as can be seen in Fig 4.3. This corresponds

to an insertion operation. In the textual representation of the alignment, an insertion is represented by a gap (-) in the reference sequence, while the query sequence contains the base corresponding to column I, as shown in Fig 4.7.

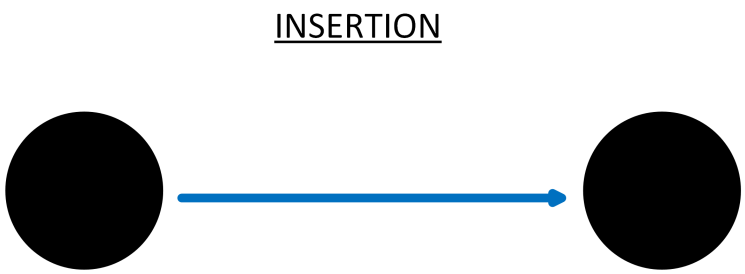


Figure 4.3: Lateral pointer connecting optimal nodes for an insertion operation.

If $L[I] - L[I-1] = 1$, the nodes are connected by a diagonal pointer, as illustrated in Fig. 4.4 and Fig. 4.5. A diagonal transition may represent either a match or a substitution operation. A match occurs when the query base corresponding to column I is identical to the reference base corresponding to row $L[I]$, as shown in Fig. 4.4. In the textual representation, the same nucleotide base is displayed in both the reference and query sequences, as illustrated in Fig. 4.7. Conversely, if the two bases are different, the diagonal transition represents a substitution, as shown in Fig. 4.5. Substitutions are represented in the same manner as matches for the textual representation; however, the corresponding query and reference positions contain the different nucleotide bases, as illustrated in Fig. 4.7.

Finally if $L[I] - L[I-1] > 1$, a vertical gap exists between the two adjacent optimal nodes. The nodes must therefore be connected using a combination of one diagonal pointer and one or more downward pointers. Examples of this case are shown in Fig. 4.6. If there is a match between any of the reference bases corresponding to the rows within the gap and the query base associated with column I, the diagonal pointer is of a match operation and will be placed in the appropriate position. The remaining transitions are represented by downward pointers, corresponding to deletion operations. If no such match exists, the

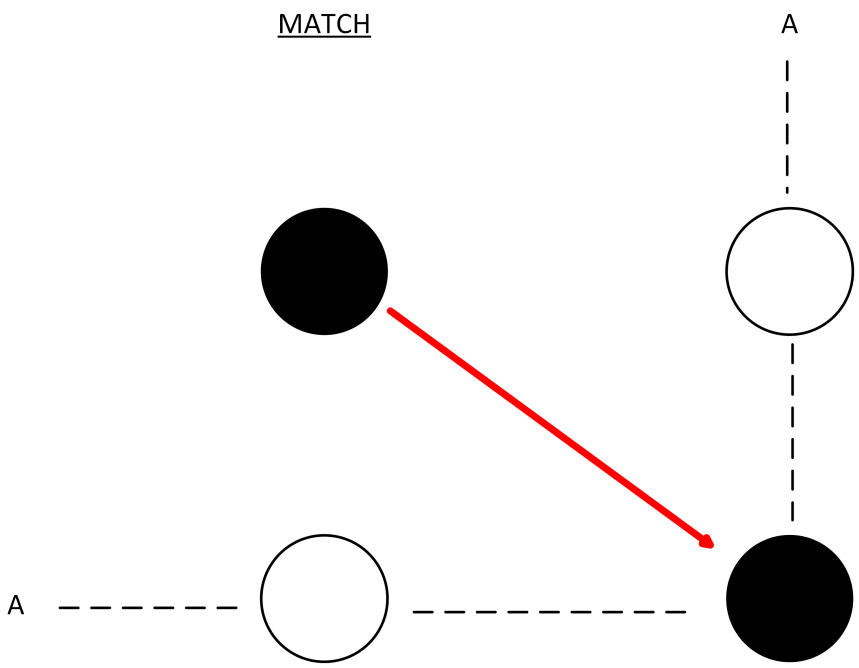


Figure 4.4: Diagonal pointer connecting optimal nodes for a match operation.

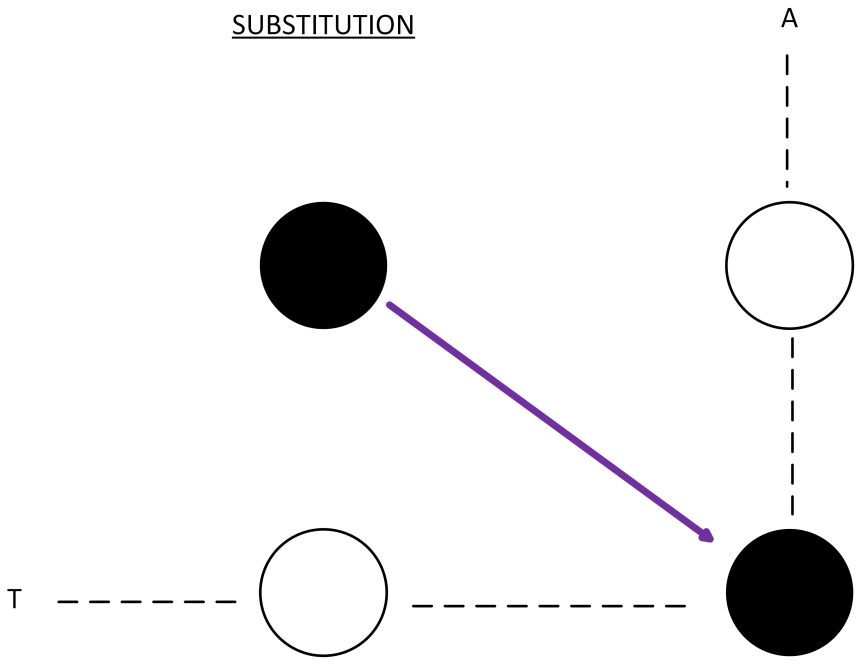


Figure 4.5: Diagonal pointer connecting optimal nodes for a substitution operation.

diagonal pointer is of a substitution operation. The diagonal and downward pointers may be arranged in any order. In the textual representation, a deletion is represented by a gap (-) in the query sequence, while the reference sequence contains the corresponding reference base, as illustrated in Fig. 4.7.

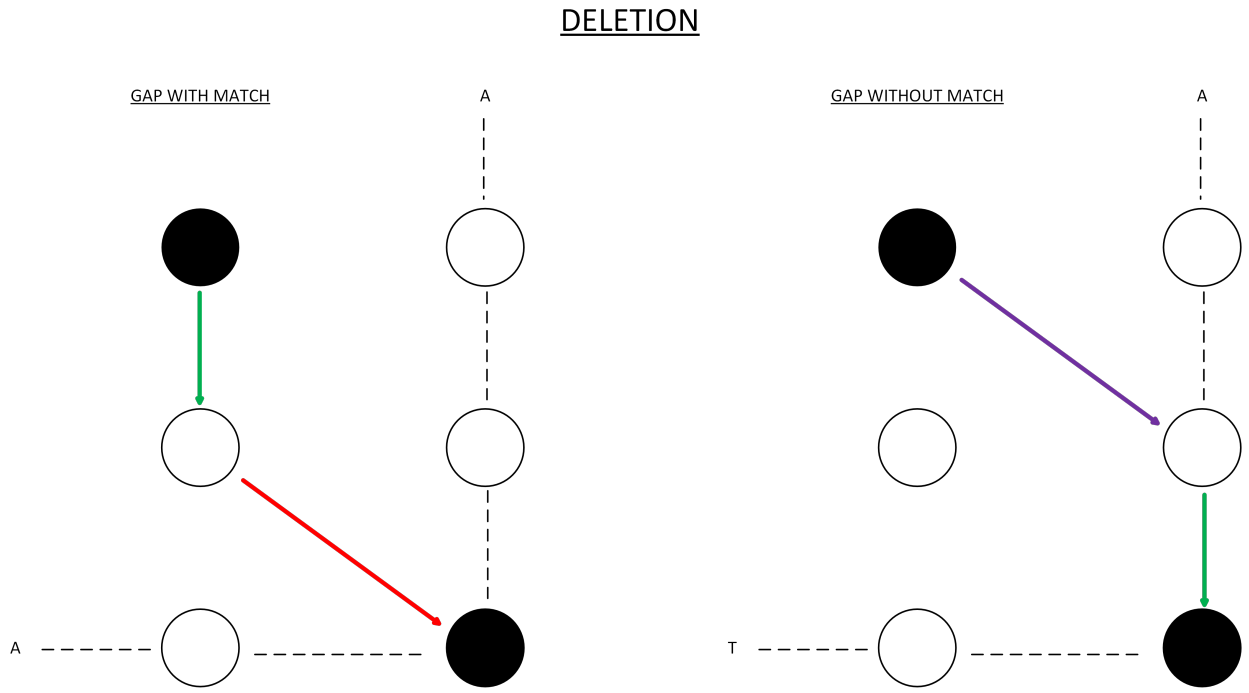


Figure 4.6: Downward and diagonal pointers used to connect optimal nodes with a vertical gap. A mixture of diagonal and downward pointers are always needed but the type of diagonal pointer and its position depends on the case. Red diagonal pointer are for match operations, purple diagonal pointer are for substitution operations

4.5 CIGAR Format

The Compact Idiosyncratic Gapped Alignment Report (CIGAR) format provides a concise representation of the alignment between a query sequence and a reference sequence and is a fundamental component of the Sequence Alignment/Map (SAM) format. A CIGAR string encodes the sequence of operations required to describe an alignment, including matches,



Figure 4.7: Base level textual representation of alignment. The reference sequence is at the top while the query sequence is at the bottom

2M1I2M1D1S1D1S

Figure 4.8: CIGAR Format of alignment depicted in Fig 4.7.

substitutions, insertions, and deletions. Operations are represented by an integer followed by a character. The integer indicates the number of consecutive occurrences of that operation, while the character identifying the operation type. `M` denotes an alignment operation involving either a match or mismatch, `I` denotes an insertion, and `D` denotes a deletion. Figure 4.8 presents the CIGAR representation corresponding to the alignment shown in Fig. 4.7.

By compactly representing the structure of an alignment, CIGAR strings enable efficient storage and facilitate downstream analysis of sequencing data. They are commonly used in applications such as variant detection, coverage analysis, genome assembly, and alignment-quality assessment. Once the traceback procedure has been completed, the software generates and stores both a textual representation of the alignment and its corresponding CIGAR representation in text files.

Chapter 5

Results

5.1 Methodology

5.1.1 Data

The DNA sequences used for performance measurements were randomly generated using a uniform probability distribution across the four nucleotide bases. The resulting data were stored in C header files as arrays of 64-bit integers, with each integer encoding 32 nucleotide bases in binary form. This representation approximates the storage of DNA sequences in virtual memory following basecalling and data cleanup. Each array entry contains 64 bits because the RoCC interface transfers data to the accelerator in 64-bit units. A total of 50,000 unique query sequences and three unique reference sequences were generated for experimental evaluation. For all measurements, the reference sequence was 50,000 bases in length, while each query sequence was 1000 bases in length.



Figure 5.1: VC707 Evaluation Board from AMD/Xilinx [7].

5.1.2 Setup

All results in this thesis were obtained from FPGA and software simulations. FPGA measurements were carried out on an Xilinx VC707 evaluation board (Fig 5.1) with bitstreams generated by Vivado using Eugene Tarassov's AMD RISCV repository [56]. This repository contains FPGA prototypes of fully a functional RISC-V SoC running a Debian Linux distribution. The board has a 40-Gb/s connectivity platform with PCIe Gen2x8 and memory interface with 1-GB DDR3 SODIM memory up to 800 MHz. Full system software simulations were performed using Berkeley's Chipyard SoC development framework [57]. FPGA power usage was measured using Texas Instrument's Fusion software via an attached adapter. Throughput measurements were obtained using performance counters that were accessed via C programs.

Like most DP algorithms, The fitting alignment algorithm implemented by KAOS is deterministic. Despite how well the sequences align the algorithm will always need to process the full DAG before returning an alignment score. This means that the only determining

factor for runtime becomes input size. For this reason, all performance measurements were conducted using the maximum supported input sizes: a 50,000-base reference sequence and 1000-base query sequences. Evaluating KAOS using these input sizes provides a representative worst-case workload for the system. Throughput was measured by recording the number of clock cycles and elapsed execution time required to compare 1000 queries consecutively against a reference sequence. Power consumption was measured as the average power observed during the execution of these 1000 query-to-reference comparisons.

5.2 Measurements

In this section, we present an extensive experimental evaluation of KAOS integrated into a RISC-V SoC, providing quantitative analyses of performance, scalability, power consumption, and energy efficiency. The SoC utilized a single Rocket core and a 16 kB L1 cache. All measurements and performance evaluations were conducted using the Xilinx VC707 evaluation board, offering a realistic hardware context for our results.

5.2.1 Throughput

Performance scaling of the SoC with a single engine KAOS configuration relative to the number of PEs is illustrated in Figure 5.2. The graph clearly demonstrates diminishing latency per alignment as the number of active PEs increases. Throughput, measured in alignments per second, significantly improves from modest initial levels to substantially higher levels with the addition of processing resources, reflecting efficient parallel hardware utilization and near-linear scalability.

A detailed analysis of the scalability of KAOS_Q with up to four engines, each with 64 PEs, across multiple frequencies (25 MHz, 50 MHz, 75 MHz, and 100 MHz) is depicted in Figure 5.3. This analysis indicates robust and consistent increases in throughput as frequency

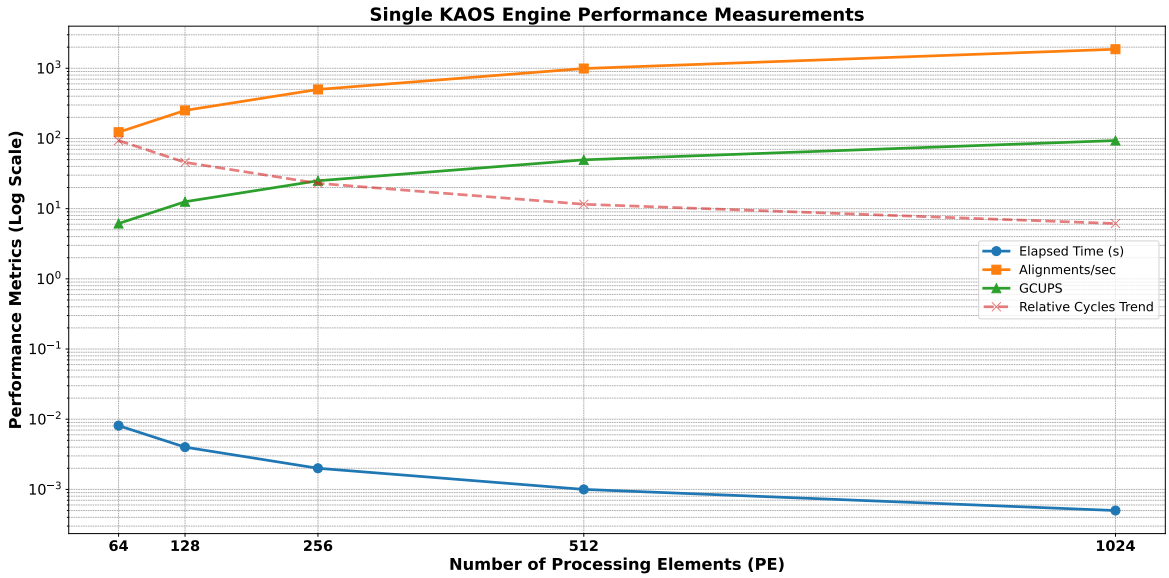


Figure 5.2: Performance trends of KAOS with only one engine and up to 1024 PEs activated, showing processing efficiency based on the number of PEs.

and engine count increases, with the system reaching peak performance at 100 MHz. These results emphasize the effectiveness of KAOS for rapid, high-performance genomic alignment tasks.

The figure also reports GCUPS (Giga Cell Updates Per Second), a popular metric for DP algorithms that quantifies the number (in the billions) of DAG cells updated per second. It is a key indicator of alignment processing efficiency. The steady increase in GCUPS alongside alignment per second measurements is juxtaposed with declining elapsed time and cycles per alignment to highlight the scalability of KAOS and its suitability for rapid, high-performance genomic alignment tasks.

Throughput for a four-engine configuration, each containing 64 PEs, further confirms the scalability and efficiency of the system. As frequency increases from 25 MHz to 100 MHz, alignments per second rise significantly, starting from 124.38 alignments per second at lower frequencies and reaching up to 496.18 alignments per second at maximum frequency, as shown

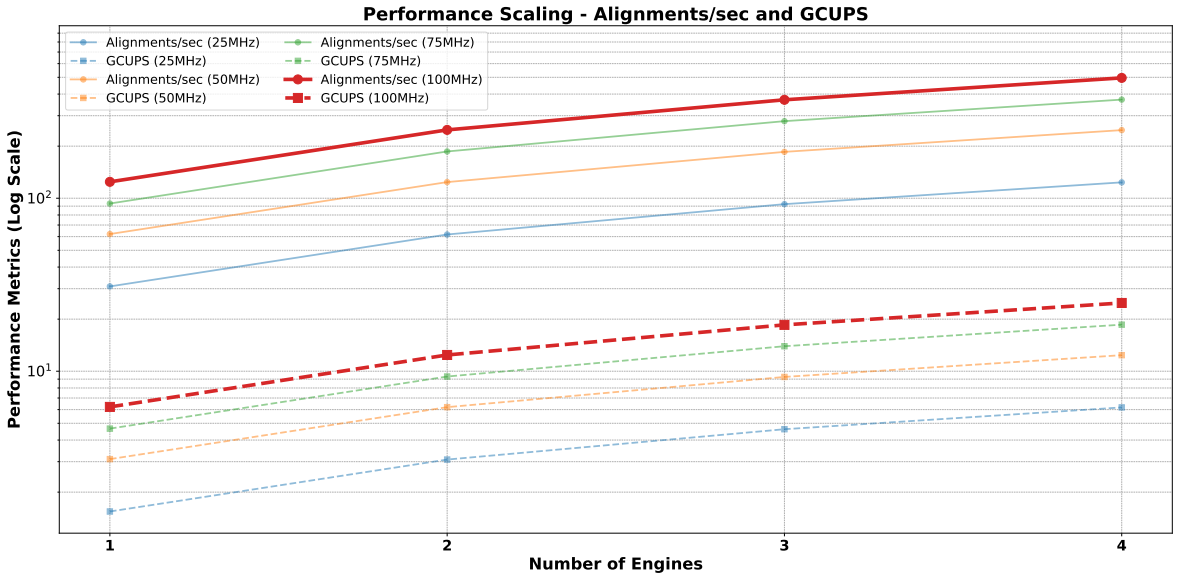


Figure 5.3: Performance scaling trends for up to 4 engines with 64-PE configuration running in parallel, highlighting alignments per second across different clock frequencies.

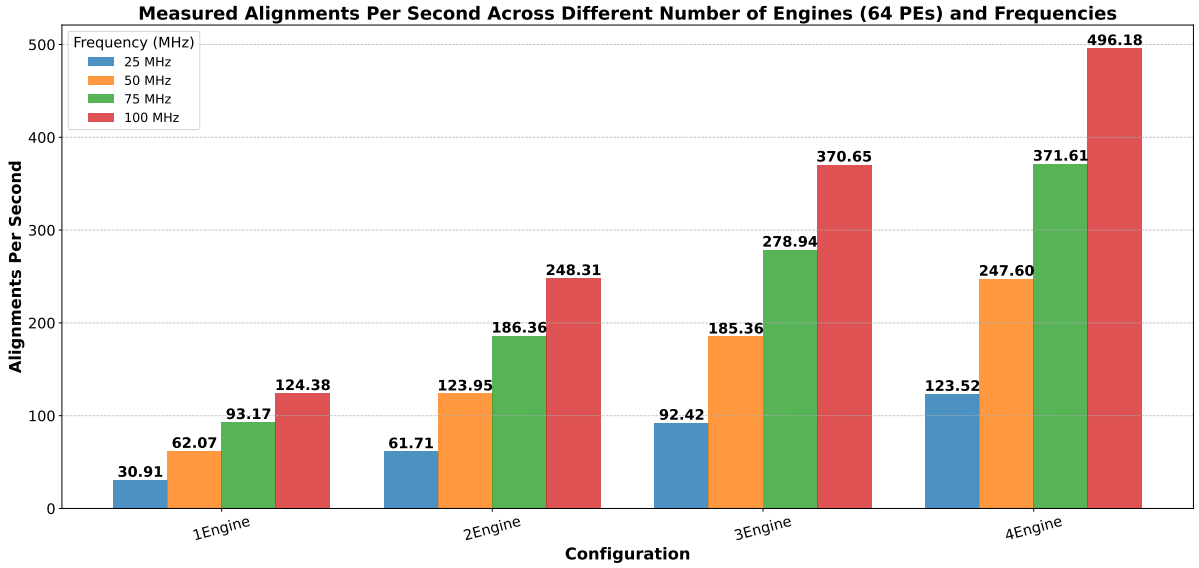


Figure 5.4: Alignments per second measurements with 4 engines each having 64 PEs at different frequencies.

in Figure 5.4. This consistent and linear growth illustrates excellent resource utilization and predictable performance behavior.

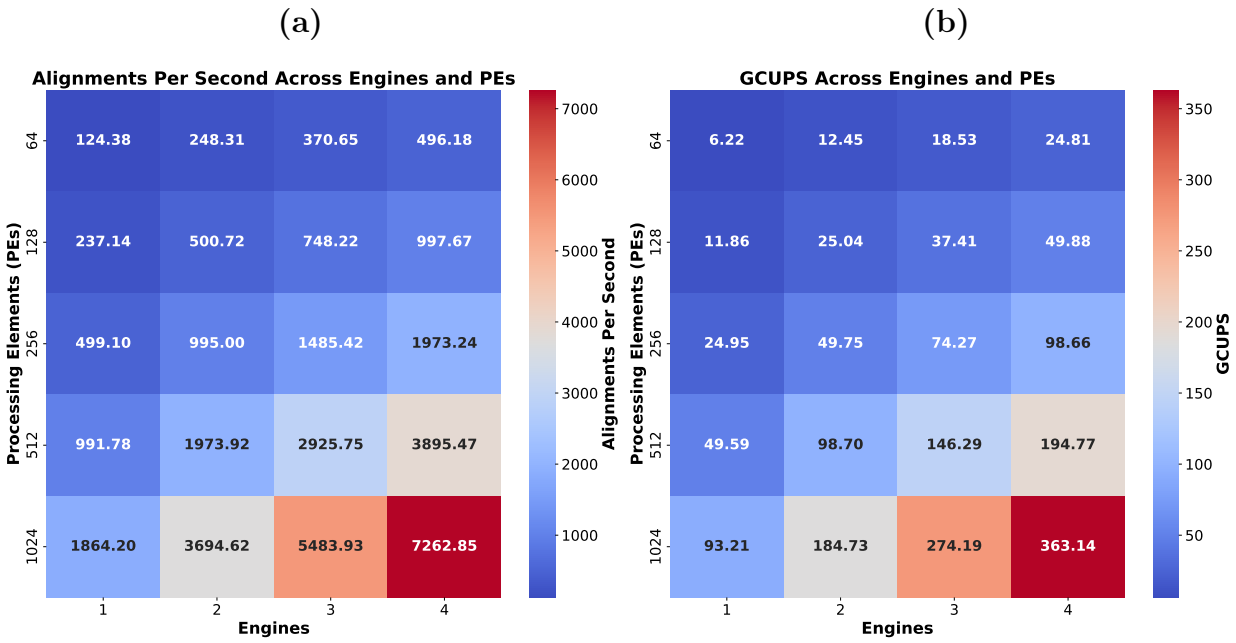


Figure 5.5: (a) Alignments Per Second Across number of Engines and PEs, (b) GCUPS Across number of Engines and PEs.

The heatmap analysis depicted in Figure 5.5 provides an insightful evaluation of the system's scalability across different configurations of engines and processing elements (PEs). The throughput significantly scales with additional hardware resources; notably, configurations range from a modest 124.38 alignments per second with one engine and 64 PEs, up to an impressive 7262.85 alignments per second using four engines each equipped with 1024 PEs, marking a remarkable increase of approximately 58-fold. Similarly, the GCUPS metric illustrates substantial computational capacity enhancements, starting from 6.22 GCUPS at minimal configuration to reaching a peak of 363.14 GCUPS at the maximum resource configuration. These quantitative results underscore the exceptional scalability and resource utilization efficiency of the architecture.

The quantitative evaluation presented in Figure 5.6 demonstrates substantial improvements

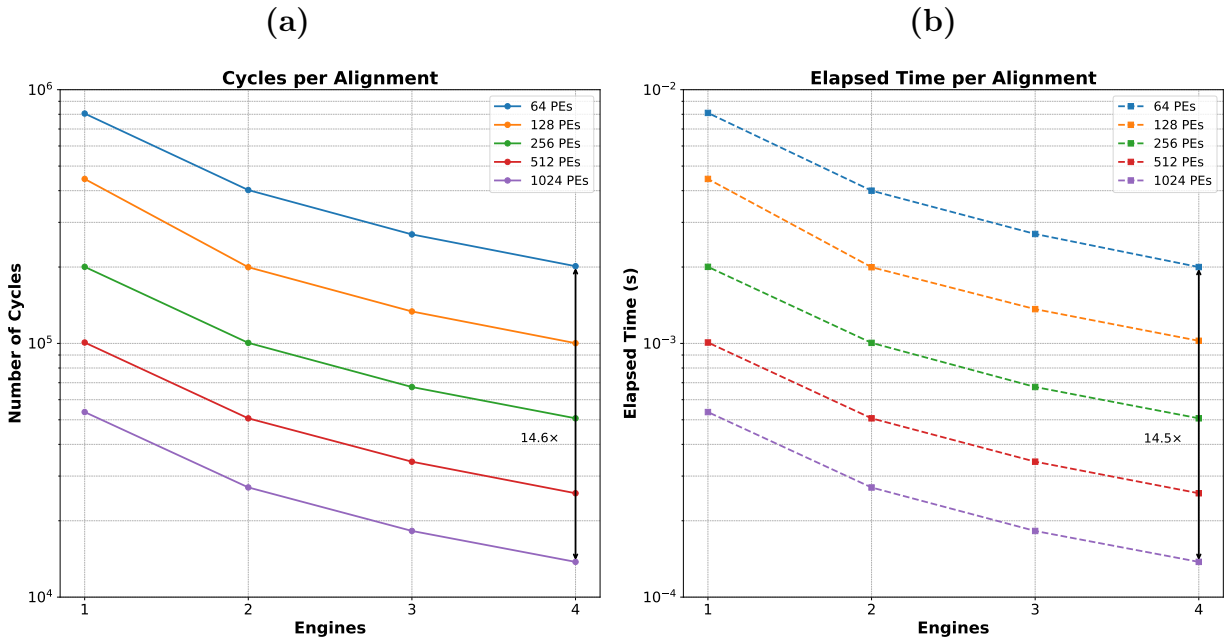


Figure 5.6: (a) Cycles per Alignment, (b) Elapsed Time per Alignment.

in cycles per alignment and elapsed time with increased computational resources. Specifically, the number of cycles per detection is significantly reduced from a maximum of approximately 1.6×10^6 cycles for a single-engine configuration with 64 PEs down to a minimum of roughly 1.1×10^5 cycles when using four engines each equipped with 1024 PEs, achieving a 14.6-fold improvement. Similarly, elapsed alignment times show remarkable reduction, decreasing from a maximum of about 8.1ms (minimal configuration) to a minimum of approximately 0.13ms (maximum configuration). These quantitative results highlight the architecture's effectiveness in leveraging parallel computational resources to achieve optimized pipeline execution and significantly reduced latency.

5.2.2 Power

Complementing these throughput results, power consumption across the same four-engine, 64-PE configuration at varying frequencies is shown in Figure 5.7. The power consumption

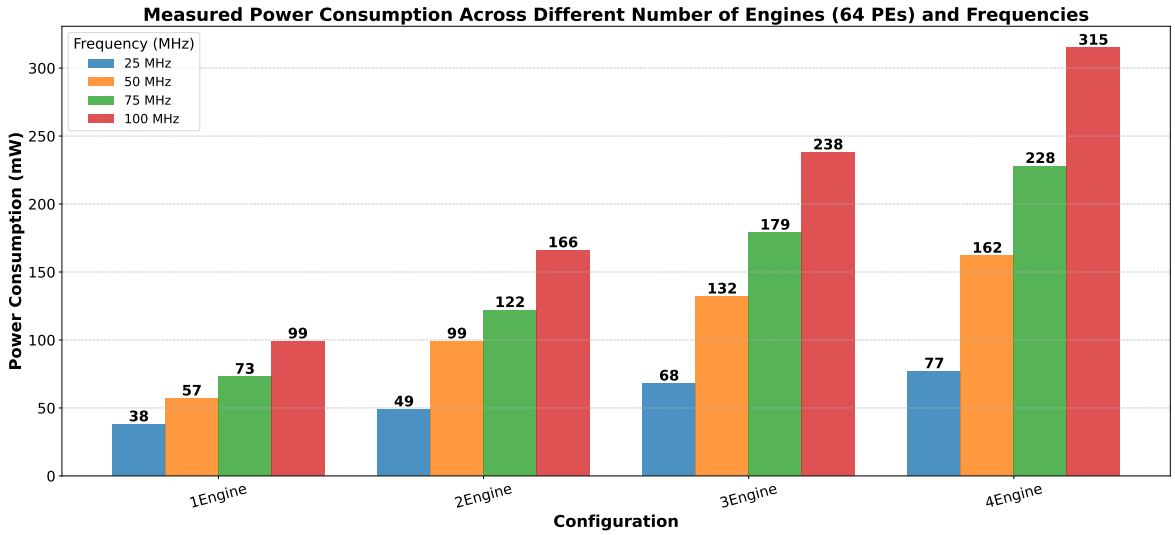


Figure 5.7: Power consumption measurements with 4 engines each having 64 PEs at different frequencies.

increases predictably from 38 mW under minimal load conditions (single engine at 25 MHz) to a maximum of 315 mW at peak throughput (four engines at 100 MHz). This controlled power scaling highlights the careful power management strategies implemented within the architecture.

Further assessing the efficiency, Figure 5.8 evaluates energy efficiency measured in alignments per second per watt. The SoC achieves optimal energy efficiency at higher operating frequencies, surpassing 1500 alignments per second per watt at 100 MHz, underlining the system's capability for effective energy usage.

Figure 5.9 addresses power overhead, illustrating a predictable and controlled increase in power consumption as additional engines and processing elements (PEs) are activated. Specifically, the power overhead ranges from approximately 89 mW for a single-engine configuration with 64 PEs, gradually increasing to around 910 mW when scaling up to four engines, each containing 1024 PEs. This incremental and well-defined power scaling, nearly an 18-fold

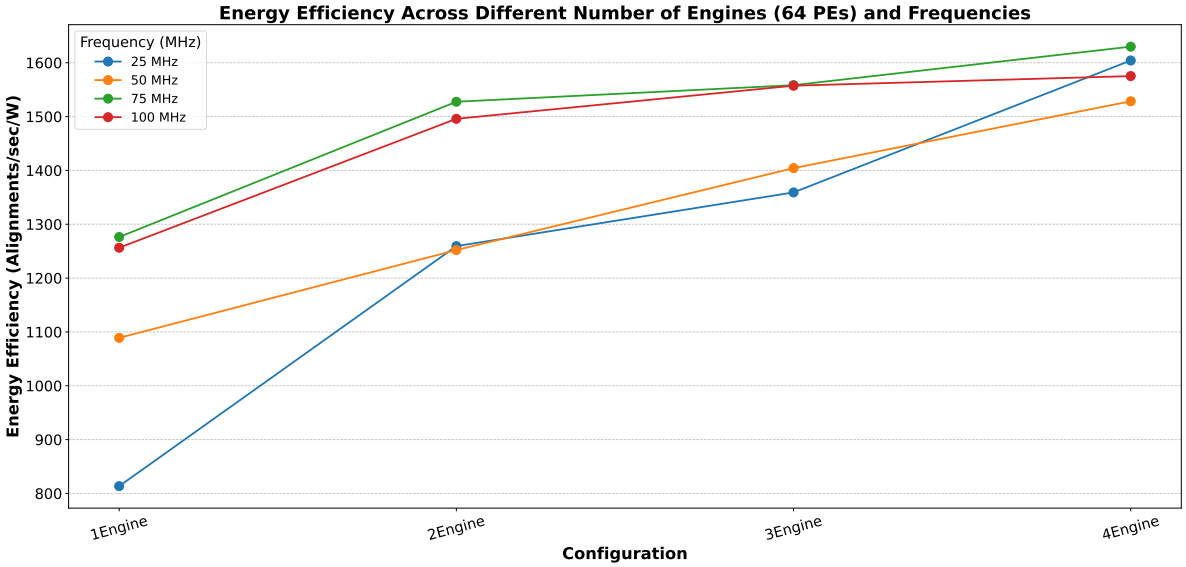


Figure 5.8: Energy efficiency measurements with 4 engines each having 64 PEs at different frequencies.

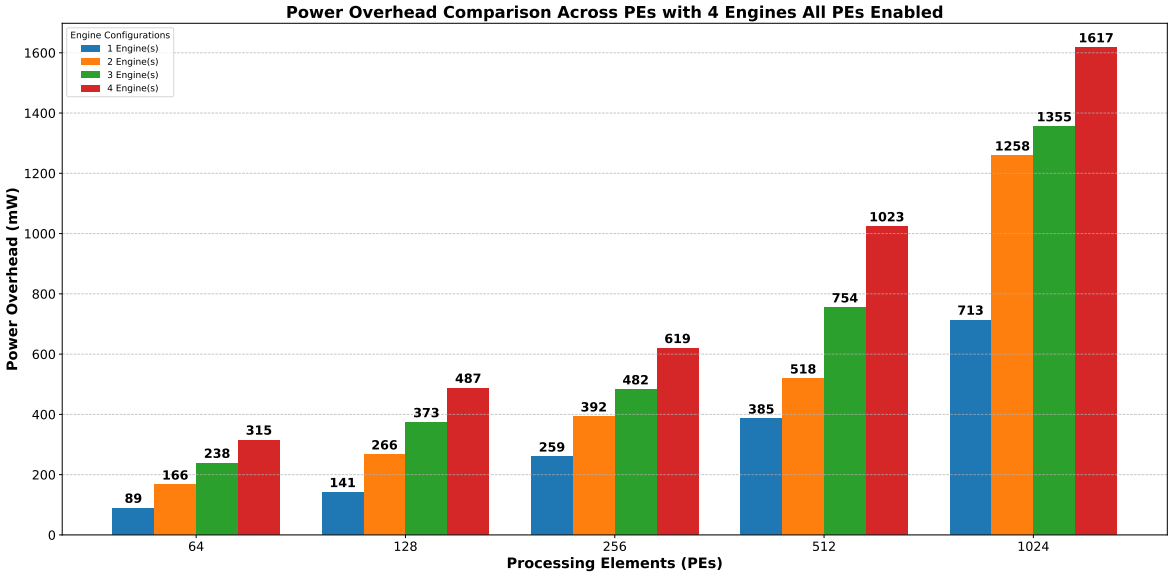


Figure 5.9: Power consumption across different number of engines and PEs.

increase from minimal to maximal resource configurations, facilitates precise and strategic decisions regarding hardware resource provisioning, enabling an optimal balance between performance demands and energy efficiency.

5.2.3 Cross-Platform Comparisons

Table 5.1: Performance, Runtime, and Power Comparison of Fitting Alignment Software Running Across Different Architectures, Including Our Proposed Accelerator.

Platform	**Accelerator @100MHz**	**RISC-V @100MHz**	**ARM @1.2GHz**	**x86 @4.4GHz**
GCUPS	6.2	7.8×10^{-4}	13.13×10^{-3}	89.39×10^{-3}
Runtime (ms)	8	64.13×10^{3}	3.93×10^{3}	0.59×10^{3}
Power (mW)	89	110	750	27.34×10^{3}
GCUPS/W	69.66	7.1×10^{-3}	1.75×10^{-2}	3.27×10^{-3}
Alignments/S	124	1.56×10^{-2}	2.62×10^{-1}	1.78

In our performance evaluation, we compared various fitting alignment executions on different computing architectures, namely, a Rocket-based RISC-V core, ARM Cortex A53 processor, Intel x86 i9-7900X CPU, and lastly RISC-V SoC with our accelerator in a single engine 64 PE configuration. All platforms ran a C implementation of the fitting alignment algorithm (in appendix A.1) on data specified in section 5.1.1. Table 5.1 summarizes the comparative analysis.

With regard to runtime analysis, our SoC design shows a significant speed up compared to other platforms. It achieves a speed up of 8016x, 491x, and 73x over RISC-V, ARM, and x86, respectively. Power consumption measurements reveal noticeable differences across architectures. The our SoC demonstrates the lowest power consumption compared to other architectures running the algorithm with 89 mW. The RISC-V core with operating frequency of 100 MHz (which has been implemented on a Virtex7 FPGA) consumes 110 mW, the ARM processor running at 1.2 GHz clock frequency consumes 150 mW, and Intel's CPU operating

at 2.4 GHz consumes 27.34 W.

Considering the results above, our accelerator is capable of performing 69.66 GCUPS/W while the x86 machine only processed 3.27×10^{-3} GCUPS/W. In other words, our accelerator is 21300x, 9811x, and 3980x more efficient than x86, RISC-V, and ARM architectures, respectively.

5.2.4 State-of-the-art Comparisons

Table 5.2: Comparison of SquiggleFilter, WFAasic, and Our KAOS variants

Metric	Squiggle Filter (2021) [58]	WFAasic (2023) [59]	KAOS Minimal	KAOS Full
Hardware Platform	ASIC	Xilinx Alveo U280	Xilinx VC707	Xilinx VC707
Runtime per Sample (ms)	0.027	176.74	**8**	**0.131**
Speedup over CPU	274× over GPU	778×	**66×**	**4284×**
Throughput (bases/s)	7.463 M	5.66 M	**124 K**	**7.26 M**
Power Consumption (W)	14.31	9.58	**0.089**	**1.6**
Power Efficiency (Mbases/s/W)	0.521	0.577	**1.39**	**4.53**
Memory Usage (KB)	500	490	**18.4**	**76.3**
Algorithm	Subsequence Dynamic Time Warping	**WaveFront**	**Fitting**	**Fitting**
Scalability (Long Sequences)	Yes	Yes	**Yes**	**Yes**
Frequency (MHz)	2500	50	**100**	**100**
Throughput Per Cycle (Bases/Cycle)	0.00298	0.1132	**0.00124**	**0.0726**

Table 5.2 presents a comparative evaluation of KAOS against state-of-the-art hardware accelerators for sequence comparison such as Squiggle Filter[58] and WFAasic[59]. Squiggle-Filter is a hardware accelerator developed to rapidly identify DNA reads of interest directly from the raw electrical signals produced by Oxford Nanopore sequencers, eliminating the need to first perform computationally intensive basecalling. Like KAOS it also uses a systolic array to implement a dynamic programming algorithm, but unlike KAOS it does not rely on the conversion of electrical signals into basecalled sequences. Instead it compares the measured incoming electrical signal of a read against the expected signal generated from a reference genome using the Dynamic Time Warping (DTW) algorithm. This enables real-time classification of DNA molecules and utilization of Oxford Nanopore's "Read Until" capability, allowing unwanted DNA molecules to be rejected during sequencing. By filtering

out irrelevant reads early, SquiggleFilter significantly reduces downstream computational workload and increases sequencing efficiency.

WFAsic is a hardware accelerator designed to accelerate DNA sequence alignment after basecalling has been completed, similar to KAOS. What makes it different is that it implements the Wavefront Alignment (WFA) algorithm, which exploits the observation that highly similar sequences only require computation around the optimal alignment path rather than across the entire dynamic programming matrix. By computing only the active wavefronts instead of every matrix cell, WFAsic substantially reduces both computation time and memory usage while still producing an exact sequence alignment and CIGAR string. This approach makes WFAsic particularly well suited for long-read sequence alignment, where traditional dynamic programming algorithms become computationally expensive.

The minimal KAOS configuration utilizes a single engine with 64 PEs, while the full configuration uses all four engines each with 1024 PEs. The minimal configuration already demonstrates a competitive balance between performance and resource efficiency. It achieves a runtime of 8 ms per sample, which significantly outperforms WFAasic's 176.74 ms and remains within an order of magnitude of SquiggleFilter's 0.027 ms, while consuming just 0.089 W, compared to 14.31 W for Squiggle Filter. Despite its low-power design, the minimal KAOS configuration sustains a throughput of 124K bases/s and exhibits a power efficiency of 1.39 Mbases/s/W, surpassing the 0.521 reported for SquiggleFilter. Its modest 18.4 KB memory usage is by far the lowest of any other accelerator in table 5.2. Further highlighting its suitability for ultra-low-resource environments.

The full KAOS configuration, enabled with all 4 engines and 1024 PEs, delivers a substantial performance leap, pushing throughput to 7.26 Mbases/s. A 58× improvement over the minimal configuration while maintaining a significantly lower power footprint of 1.6 W relative to SquiggleFilter's 14.31 W. With a runtime of just 0.131 ms per sample, the full configuration offers near real-time responsiveness and outpaces WFAasic's speed by a

factor of 1349×, achieving a speedup of 4284× over CPU. It also boasts the highest power efficiency among all designs at 4.53 Mbases/s/W, and maintains modest 76.3 KB memory usage, proving scalable for long-sequence processing. The full configuration also has the best energy efficiency out of all the designs. It is able to process 4.53 million bases per watt, better than the 1.39 million bases per watt of the minimum configuration and far greater than the 0.52 and 0.58 milion bases per watt of SquggleFilter and WFAsic respectively.

The results clearly demonstrate that both configurations of KAOS offer compelling trade-offs between power, runtime, and throughput, making them highly adaptable for genomic processing and time-sensitive pathogen detection pipelines. It is also important to point out that the FPGA-implemented version of WFAsic only performs well on reads of 300 bases or less. Our design can easily perform with 3X this size, allowing us to work with read sizes up to 1000 bases.

5.3 Fulfilling Requirements

As stated in the beginning of chapter 3, there are certain requirements that our accelerator has to meet. Unfortunately we were not able to perform measurements on the actual ASIC due to complications, so we had to extrapolate from the FPGA measurements instead.

5.3.1 Throughput Requirement

A primary design requirement was that KAOS, when operating at 200 MHz, should achieve a throughput equal to or greater than the approximately 230,000 bases/s output rate of the MinION. The minimum KAOS configuration, which represents the lowest-throughput configuration evaluated in this work, achieved approximately 124,000 bases/s when operating at 100 MHz. The measured results demonstrated an approximately linear relationship between clock frequency and throughput over the evaluated frequency range. Based on this measured

scaling behavior, extrapolating the operating frequency from 100 MHz to 200 MHz suggests a throughput of approximately 248,000 bases/s. Under the assumption that the architecture maintains the same scaling behavior at 200 MHz, this projected throughput would exceed the 230,000 bases/s system requirement.

5.3.2 Power Requirement

Another system-level requirement was that each hardware addition should consume approximately 100 mW or less. The minimum KAOS engine, with only 64 PEs, consumed approximately 89 mW when measured on the FPGA platform at an operating frequency of 100 MHz. This measured value is within the target power budget. However, power consumption would be expected to increase at a higher operating frequency, such as 200 MHz. Likely going over the 100 mW requirement.

FPGA tends to be far less energy efficient than ASIC for the same design, due to employing a lot more logic like LUTs in pursuit of being programmable [60]. The hope is that as the design is translated to ASIC the power usage of a single KAOS engine is still around 100 mW or less.

5.3.3 Accuracy Requirement

An implicit requirement of KAOS is functional accuracy. Because KAOS is intended to be a hardware implementation of the fitting alignment algorithm, it should produce the same alignment score as the software implementation when provided with identical input sequences and scoring parameters.

A software implementation of the fitting algorithm was developed as a functional reference model. During the design phase, an initial test suite consisting of 20 directed test cases was used to validate the KAOS architecture. These tests included queries of different lengths,

perfect alignments, partial alignments occurring at different positions within the reference sequence, and randomly generated sequences. All tested KAOS configurations produced alignment scores identical to those generated by the software implementation.

Following deployment of the design to the FPGA, the complete set of 50,000 generated query sequences was evaluated against a reference sequence. Every tested KAOS configuration produced results consistent with the software implementation across all 50,000 test cases. These experiments provide strong functional evidence that KAOS correctly implements the fitting alignment algorithm, meeting our standard of accuracy.

Chapter 6

Conclusion

6.1 Summary

This thesis proposes KAOS (K-Mer Alignment On Silicon), a hardware accelerator integrated within a RISC-V System-on-Chip (SoC) to perform sequence alignment directly on embedded hardware for real-time pathogen detection. KAOS is framed around third-generation nanopore sequencers, particularly the Oxford Nanopore MinION, which enable real-time DNA sequencing in a compact form factor. We envision a design where a MinION like sequencer and the SoC housing KAOS are under one chassis, allowing KAOS to do downstream bioinformatics processing on base-called reads. In this case KAOS implements a dynamic programming algorithm to see how well the reads fit in the genome of a pathogen. Pathogen detection represents a particularly valuable target application due to the relatively small size of viral genomes and the need for rapid diagnosis during infectious disease outbreaks.

There are many dynamic programming-based sequence alignment techniques that could be used as the computational foundation of the proposed accelerator. Classical approaches including Needleman–Wunsch /global alignment and Smith–Waterman /local alignment are examined, highlighting their respective strengths and limitations for pathogen detection

applications. We ultimately focus on the fitting (Shuffle-LAGAN) alignment algorithm, which combines properties of both global and local alignment by requiring complete alignment of the query sequence while permitting flexible placement within the reference genome. This characteristic makes it particularly suitable for determining whether a sequencing read originates from a known pathogen genome.

The heart of this thesis is chapter 3. It explains the architecture of the KAOS accelerator and its integration within a RISC-V-based SoC. The design is motivated by the throughput requirements of real-time nanopore sequencing, where software-only implementations are shown to be incapable of sustaining the data rates produced by a MinION flow cell. To address this limitation, KAOS employs a dedicated systolic architecture consisting of 64 processing elements operating in parallel to accelerate fitting alignment computations. The chapter describes the overall system organization, including the Rocket processor, RoCC interface, on-chip memories, FIFO buffers, and systolic engine. Detailed descriptions of the processing element architecture, wavefront computation strategy, data movement mechanisms, and score management infrastructure are provided. The chapter also introduces several enhanced accelerator variants designed to improve scalability, support multiple simultaneous alignments, and facilitate traceback generation.

Although alignment scores are sufficient for classification tasks, many bioinformatics applications require explicit reconstruction of the optimal alignment. Generating alignment visualizations while operating under the memory constraints of our embedded system was very challenging. A naive traceback implementation would require storing operation information for every cell in the DAG, resulting in memory requirements that are impractical. To overcome this limitation, we introduces a method for identifying a reduced sub-DAG that contains the optimal alignment path. By determining the start and end locations of the alignment within the reference sequence, the system significantly reduces the search space and memory footprint required for traceback. The system then executes a Divide and Conquer algorithm

that enables efficient alignment reconstruction while preserving the objectives of the overall design.

Lastly, we evaluate the performance of the KAOS accelerator and its architectural variants in terms of throughput, scalability, and resource efficiency. Results demonstrate that when it comes to throughput, KAOS substantially out performs software based executions of fitting alignment on a general-purpose processor. We investigate the effects of increasing the number of processing elements and employing multiple accelerator engines, showing that throughput scales predictably with available hardware resources. Measurements of performance, power and efficiency are used to compare KAOS to other state of the art hardware sequence aligners. Validating the design choices made throughout this thesis and demonstrating the feasibility of performing real-time pathogen detection within an embedded computing environment.

6.2 Future Works

Although the KAOS architecture demonstrates the potential of hardware acceleration for embedded DNA sequence comparison, several opportunities remain for further improvement. Future extensions could enhance the biological realism of the alignment model, improve the efficiency of alignment reconstruction and visualization, and integrate the accelerator into a more complete pathogen detection framework. The following subsections discuss these potential directions for future work.

6.2.1 Affine Penalties and Customizable Scoring matrix

When going over the hardware architecture for KAOS in chapter 3, it is shown that the scoring matrix values for each operation are hardwired into the PEs. In reality there is no one-size-fits-all scoring matrix for DNA sequence alignment. Scoring matrices are tailored for different use cases by researchers and professionals in the field. For this reason, the first

improvement to KAOS would be a customizable scoring matrix. Giving users full control of how much they want to reward or penalize certain alignment operations. This can be done by allowing users to set the values in software and passing those values through RoCC instructions to KAOS. KAOS would then set the registers in each PE to the appropriate values.

Another limitation of the current scoring model concerns the treatment of insertions and deletions (indels). In our design each indel is penalized the same, -1 per indel, regardless of if it is adjacent to other indels or far away. However, it is actually far more common for DNA molecules to experience indels in groups. To make alignments more biologically realistic an Affine Gap Penalty model can be used. The difference would be that for a group of indels, the initial indel would be penalized at the usual cost but the other indels in the group would each incur a much lower cost. KAOS could implement Affine Gaps into its systolic array but it would need more hardware resources, memory and maybe even a floating point design.

6.2.2 Traceback in Hardware

A hardware implementation of traceback represents another important direction for future work. The current approach relies on software-based processing to reconstruct and visualize sequence alignments. Moving portions of this process to a dedicated hardware accelerator could reduce the time required to generate alignment information and improve the overall responsiveness of the system.

A hardware traceback implementation would likely require a higher-level control module capable of coordinating the KAOS engines and identifying the relevant sub-DAG for traceback. Memory would remain one of the primary implementation challenges. In addition to the total volume of pointer information that must be stored, the storage architecture would need to accommodate the wavefront-parallel traversal of the sub-DAG.

The KAOS architecture would therefore require additional registers within and between PEs, as well as dedicated SRAM resources for storing traceback pointers or equivalent reconstruction information. Since storing the complete sub-DAG would be prohibitively expensive for memory in our SoC, an efficient implementation may require the sub-DAG to be partitioned and processed incrementally. A divide-and-conquer approach could therefore remain an attractive method for reducing memory requirements while enabling efficient hardware-assisted traceback.

6.2.3 Pathogen Detection Framework

The most significant potential extension of this work is the development of a complete pathogen detection framework built around KAOS. Pathogens have virulence genes/ pathogenicity islands (PAIs). These are regions of a pathogen's genome that contain genes responsible for causing disease, enabling host invasion, immune evasion, toxin production, nutrient acquisition, or other pathogenic functions[61][62]. Standard detection platforms like PCR works by targeting virulence genes that are unique to pathogenic strains or located within pathogenicity islands (PAIs). PCR plants markers on these regions and then amplifies them[63]. DNA amplification is the process of generating multiple identical copies of a specific DNA sequence from an initial minuscule amount[64]. After multiple rounds of amplification, the presence of the pathogen can then be confirmed by identifying if there is a significant amount of these markers. If the pathogen is not present then amplification cannot take place and the markers would not bind. Keep in mind that in any DNA sample, the amount of pathogen DNA is relatively tiny in comparison to the amount of non pathogen DNA, mostly human DNA[10]. Without substantial amplification looking for the markers would be like trying to find a needle in a haystack.

KAOS can implement its own framework by mapping reads to these virulent regions.

It can do this with a two stage pipeline. The first stage is essentially a filter. KAOS_B/ KAOS_C is used to get alignment scores from incoming reads. Those with high scores relative to the read's length are allowed to the 2nd stage. The threshold for a high score is determined by the user, presumably an expert in the field. For example, a user can set the threshold to be 90% of the query's length. For a 1000 base query that is a score of 900. The assumption is that the vast majority of reads from human DNA will fail to reach this threshold and will be filtered out immediately. However, suppose a 1000 base read from the pathogen experiences noise as it goes through the sequencing pipeline so instead of getting a perfect score of 1000, it gets 950. Even with the noise the threshold would allow sequences of high similarity to pass to the 2nd stage.

In the 2nd stage reads with high scores are sent to KAOS_T. It would proceed to return the starting index of alignment, ending index of alignment, and a CIGAR format representation of alignment via Divide and Conquer. Software can use these indexes to determine if the reads overlap or map directly to virulent regions. It can also use the CIGAR format output to show users a real time visual of the exact alignment of the reads to the pathogen genome. It is then up to the user to determine how much coverage of the virulent region is needed to declare a patient as positive.

Bibliography

[1] Dna: The body's instruction manual. https://www.ashg.org/discover-genetics/building-blocks/, 2019.

[2] A beginner's guide to dna sequencing. https://www.novogene.com/us-en/resources/blog/a-beginners-guide-to-dna-sequencing-blog/.

[3] Complex structural variants in mendelian disorders. https://alsanju.github.io/NGSchool2018/basecalling.html.

[4] Alexander S Mikheyev and Mandy MY Tin. A first look at the oxford nanopore minion sequencer. *Molecular ecology resources*, 14(6):1097–1102, 2014.

[5] Phillip Compeau and Pavel Pevzner. *Bioinformatics Algorithms:An Active Learning Approach*, volume 1. Active Learning Publishers, 2 edition, 2015.

[6] Phillip Compeau and Pavel Pevzner. *Bioinformatics Algorithms: An Active Learning Approach*, volume 1. Active Learning Publishers, 3 edition, 2023.

[7] Amd virtex™ 7 fpga vc707 evaluation kit. https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/ek-v7-vc707-g.html.

[8] Pierre-Edouard Fournier, Gregory Dubourg, and Didier Raoult. Clinical detection and characterization of bacterial pathogens in the genomics era. *Genome Med.*, 6(11):114, November 2014.

[9] Ian M. Mackay. Real-time pcr in the microbiology laboratory. *Clinical Microbiology and Infection*, 10(3):190–212, 2004.

[10] Charles Y. Chiu and Steven A. Miller. Clinical metagenomic next-generation sequencing for pathogen detection. *Annual Review of Pathology: Mechanisms of Disease*, 14:319–338, 2019.

[11] Rosanna W. Peeling, Mabey David, et al. Diagnostics for covid-19: moving from pandemic response to control. *The Lancet Infectious Diseases*, 17(9):e261–e272, 2017. General reference for importance of rapid diagnostics.

[12] Rachael Rettner. Dna: Definition, structure and discovery kernel description. `https://www.livescience.com/37247-dna.html`, 2023.

[13] Jeremy M. Berg, John L. Tymoczko, and Lubert Stryer. *Biochemistry 5ed.* W.H. Freeman, New York, NY, 5 edition, February 2002.

[14] Sara Goodwin, John D McPherson, and W Richard McCombie. Coming of age: ten years of next-generation sequencing technologies. *Nat. Rev. Genet.*, 17(6):333–351, May 2016.

[15] Artem E Men, Peter Wilson, Kirby Siemering, and Susan Forrest. Sanger DNA sequencing. In *Next Generation Genome Sequencing*, pages 1–11. Wiley-VCH Verlag GmbH & Co. KGaA, Weinheim, Germany, 2008.

[16] Joseph F Boland, Charles C Chung, David Roberson, Jason Mitchell, Xijun Zhang, Kate M Im, Ji He, Stephen J Chanock, Meredith Yeager, and Michael Dean. The new

sequencer on the block: comparison of life technology's proton sequencer to an illumina HiSeq for whole-exome sequencing. *Hum. Genet.*, 132(10):1153–1163, October 2013.

[17] Lloyd D Potts, Alex Douglas, Luis J Perez Calderon, James A Anderson, Ursula Witte, James I Prosser, and Cécile Gubry-Rangin. Chronic environmental perturbation influences microbial community assembly patterns. *Environ. Sci. Technol.*, 56(4):2300–2311, February 2022.

[18] Jay Shendure, Shankar Balasubramanian, George M Church, Walter Gilbert, Jane Rogers, Jeffery A Schloss, and Robert H Waterston. DNA sequencing at 40: past, present and future. *Nature*, 550(7676):345–353, October 2017.

[19] Mehdi Kchouk, Jean Francois Gibrat, and Mourad Elloumi. Generations of sequencing technologies: From first to next generation. *Biol. Med. (Aligarh)*, 09(03), 2017.

[20] Cliff Meldrum, Maria A Doyle, and Richard W Tothill. Next-generation sequencing for cancer diagnostics: a practical perspective. *Clin. Biochem. Rev.*, 32(4):177–195, November 2011.

[21] Carl W Fuller, Shiv Kumar, Mintu Porel, Minchen Chien, Arek Bibillo, P Benjamin Stranges, Michael Dorwart, Chuanjuan Tao, Zengmin Li, Wenjing Guo, Shundi Shi, Daniel Korenblum, Andrew Trans, Anne Aguirre, Edward Liu, Eric T Harada, James Pollard, Ashwini Bhat, Cynthia Cech, Alexander Yang, Cleoma Arnold, Mirkó Palla, Jennifer Hovis, Roger Chen, Irina Morozova, Sergey Kalachikov, James J Russo, John J Kasianowicz, Randy Davis, Stefan Roever, George M Church, and Jingyue Ju. Real-time single-molecule electronic DNA sequencing by synthesis using polymer-tagged nucleotides on a nanopore array. *Proc. Natl. Acad. Sci. U. S. A.*, 113(19):5233–5238, May 2016.

[22] David Deamer, Mark Akeson, and Daniel Branton. Three decades of nanopore sequencing. *Nat. Biotechnol.*, 34(5):518–524, May 2016.

[23] Clive G Brown and James Clarke. Nanopore development at oxford nanopore. *Nat. Biotechnol.*, 34(8):810–811, August 2016.

[24] Harrison S Edwards, Raga Krishnakumar, Anupama Sinha, Sara W Bird, Kamlesh D Patel, and Michael S Bartsch. Real-time selective sequencing with RUBRIC: Read until with basecall and reference-informed criteria. *Sci. Rep.*, 9(1):11475, August 2019.

[25] Hengyun Lu, Francesca Giordano, and Zemin Ning. Oxford nanopore MinION sequencing and genome assembly. *Genomics Proteomics Bioinformatics*, 14(5):265–279, October 2016.

[26] Xuan Lv, Zhiguang Chen, Yutong Lu, and Yuedong Yang. An end-to-end oxford nanopore basecaller using convolution-augmented transformer. *bioRxiv*, 2020.

[27] John D McPherson Sara Goodwin and W Richard McCombie. Coming of age: ten years of next-generation sequencing technologies. *Nature Reviews Genetics*, 17:333, 2016.

[28] Alexander L. Greninger, Samer N. Naccache, Sarah Federman, Gene W. Yu, Placide Mbala, Victor Bres, David Stryke, Jennifer Bouquet, Seshadri Somasekar, Jonathan M. Linnen, Roger Dodd, Patrick Mulembakani, Bradley S. Schneider, Joseph N. Fair, Jean Jacques Muyembe-Tamfum, Susan L. Stramer, and Charles Y. Chiu. Rapid metagenomic identification of viral pathogens in clinical samples by real-time nanopore sequencing analysis. *Genome Medicine*, 7:99, 2015.

[29] Xiang Deng, Anish Achari, Sarah Federman, Gene Yu, Seshadri Somasekar, Robert Mendoza, Julie Streithorst, Stephanie Miller, David Stryke, Beth Constantinides, Stephanie Goldstein, Jordon C. Hsu, Brian Miller, Glenn Simmons, Gregory Cloherty, Theodore Kupferman, James Bard, James Rogers, and Charles Y. Chiu. Metagenomic sequencing with spiked primer enrichment for viral diagnostics and genomic epidemiology. *Nature Microbiology*, 5:443–454, 2020.

[30] Yu Wang, Yanan Zhao, Aristeidis Bollas, Yang Wang, and Kin Fai Au. Nanopore sequencing technology, bioinformatics and applications. *Nature Biotechnology*, 39:1348–1365, 2021.

[31] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[32] Andrea Mira-Jover, Eva Graciá, Andrés Giménez, Uwe Fritz, Roberto Carlos Rodríguez-Caro, and Yann Bourgeois. Taking advantage of reference-guided assembly in a slowly-evolving lineage: Application to testudo graeca. *PLoS One*, 19(8):e0303408, August 2024.

[33] Jang-Il Sohn and Jin-Wu Nam. The present and future of de novo whole-genome assembly. *Brief. Bioinform.*, 19(1):23–40, January 2018.

[34] Scott McGinnis and Thomas L Madden. BLAST: at the core of a powerful and diverse set of sequence analysis tools. *Nucleic Acids Res.*, 32(Web Server issue):W20–5, July 2004.

[35] Yatish Turakhia, Gill Bejerano, and William J Dally. Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly. *ACM SIGPLAN Notices*, 53(2):199–213, 2018.

[36] Michael J. Mina and Kristian G. Andersen. COVID-19 testing: one size does not fit all. *Science*, 371(6525):126–127, 2021.

[37] Themoula Charalampous, Gemma L. Kay, Hollian Richardson, Alp Aydin, Rossella Baldan, and et al. Nanopore metagenomics enables rapid clinical diagnosis of bacterial lower respiratory infection. *Nature Biotechnology*, 37(7):783–792, 2019.

[38] Rowena A. Bull, Thiruni N. Adikari, James M. Ferguson, Jillian M. Hammond, Irina Stevanovski, and et al. Analytical validity of nanopore sequencing for rapid sars-cov-2 genome analysis. *Nature Communications*, 11(1):6272, 2020.

[39] Bert Bogaerts, An Van den Bossche, Bavo Verhaegen, Laurence Delbrassinne, Wesley Mattheus, Stéphanie Nouws, Maxime Godfroid, Stefan Hoffman, Nancy H. C. Roosens, Sigrid C. J. De Keersmaecker, and Kevin Vanneste. Closing the gap: Oxford Nanopore Technologies R10 sequencing allows comparable results to illumina sequencing for snp-based outbreak investigation of bacterial pathogens. *Journal of Clinical Microbiology*, 62(5):e01576–23, 2024.

[40] Laura E. Kafetzopoulou, Steven T. Pullan, Philippe Lemey, Marc A. Suchard, Daniele U. Ehichioya, and et al. Metagenomic sequencing at the epicenter of the Nigeria 2018 Lassa fever outbreak. *Science*, 363(6422):74–77, 2019.

[41] Samia N Naccache, Scot Federman, Narayanan Veeraraghavan, Matei Zaharia, Deanna Lee, Erik Samayoa, Jerome Bouquet, Alexander L Greninger, Ka-Cheung Luk, Barryett Enge, et al. A cloud-compatible bioinformatics pipeline for ultrarapid pathogen identification from next-generation sequencing of clinical samples. *Genome research*, 24(7):1180–1192, 2014.

[42] Derrick E Wood, Jennifer Lu, and Ben Langmead. Improved metagenomic analysis with kraken 2. *Genome Biol.*, 20(1):257, November 2019.

[43] Furong Tang, Lei Xu, Yusen Zhang, Zhen Li, and Yao Liu. Developments in algorithms for sequence alignment: A review. *Frontiers in Bioengineering and Biotechnology*, 10:857306, 2022.

[44] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[45] Valer Gotea, Vamsi Veeramachaneni, and Wojciech Makałowski. Mastering seeds for genomic size nucleotide BLAST searches. *Nucleic Acids Res.*, 31(23):6935–6941, December 2003.

[46] Jie Cheng, Lifu Hu, Wei Xu, Hanhua Chen, and Tian Xia. Hardware acceleration of minimap2 genomic sequence alignment algorithm. In *Proceedings of the 53rd International Conference on Parallel Processing*, pages 887–897, New York, NY, USA, August 2024. ACM.

[47] Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V Bzikadze, Alla Mikheenko, Mitchell R Vollger, Nicolas Altemose, Lev Uralsky, Ariel Gershman, Sergey Aganezov, Savannah J Hoyt, Mark Diekhans, Glennis A Logsdon, Michael Alonge, and Antonarakis et al. The complete sequence of a human genome. *Science*, 376(6588):44–53, April 2022.

[48] Fan Wu, Su Zhao, Bin Yu, Yan-Mei Chen, Wen Wang, Zhi-Gang Song, Yi Hu, Zhao-Wu Tao, Jun-Hua Tian, Yuan-Yuan Pei, Ming-Li Yuan, Yu-Ling Zhang, Fa-Hui Dai, Yi Liu, Qi-Min Wang, Jiao-Jiao Zheng, Lin Xu, Edward C Holmes, and Yong-Zhen Zhang. A new coronavirus associated with human respiratory disease in china. *Nature*, 579(7798):265–269, March 2020.

[49] Mohammed Alser, Jeremy Rotman, Kodi Taraszka, Huwenbo Shi, Pelin Icer Baykal, Harry Taegyun Yang, Victor Xue, Sergey Knyazev, Benjamin D. Singer, Brunilda Balliu, David Koslicki, Pavel Skums, Alex Zelikovsky, Can Alkan, Onur Mutlu, and Serghei Mangul. Technology dictates algorithms: recent developments in read alignment. *Genome Biology*, 22(1):249, 2021.

[50] Alexander Poliakov Michael Brudno1, Sanket Malde1 et al. Glocal alignment: finding rearrangements during alignment. *Bioinformatics*, 2000.

[51] Yunus Dawji, Zhongpan Wu, Abel Beyene, Karim Hammad, Ebrahim Ghafar-Zadeh, and Sebastian Magierowski. An SoC design for future mobile DNA detection. *IEEE Embed. Syst. Lett.*, 16(2):86–89, June 2024.

[52] The long view on sequencing. *Nat. Biotechnol.*, 36(4):287, April 2018.

[53] Peter Perešíni, Vladimír Boža, Broňa Brejová, and Tomáš Vinař. Nanopore base calling on the edge. *Bioinformatics*, 37(24):4661–4667, December 2021.

[54] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016.

[55] Scott Lloyd and Quinn O Snell. Hardware accelerated sequence alignment with traceback. *Int. J. Reconfigurable Comput.*, 2009(1):1–10, January 2009.

[56] Eugene Tarassov. Amd/xilinx vivado block designs for fpga risc-v soc running debian linux distro. `https://github.com/eugene-tarassov/vivado-risc-v`, 2020.

[57] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanovic, and Borivoje Nikolic. Chipyard: Integrated design, simulation, and implementation framework for custom SoCs. *IEEE Micro*, 40(4):10–21, July 2020.

[58] Tim Dunn, Harisankar Sadasivan, Jack Wadden, Kush Goliya, Kuan-Yu Chen, David Blaauw, Reetuparna Das, and Satish Narayanasamy. Squigglefilter: An accelerator for portable virus detection. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 535–549, 2021.

[59] Abbas Haghi, Lluc Alvarez, Jordi Front, Juan Miguel de Haro Ruiz, Roger Figueras, Max Doblas, Santiago Marco-Sola, and Miquel Moreto. Wfasic: A high-performance asic accelerator for dna sequence alignment on a risc-v soc. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 392–401, 2023.

[60] Amara Amara, Frédéric Amiel, and Thomas Ea. FPGA vs. ASIC for low power applications. *Microelectronics Journal*, 37(8):669–677, August 2006.

[61] Ohad Gal-Mor and B Brett Finlay. Pathogenicity islands: a molecular toolbox for bacterial virulence. *Cell. Microbiol.*, 8(11):1707–1719, November 2006.

[62] J Hacker and J B Kaper. Pathogenicity islands and the evolution of microbes. *Annu. Rev. Microbiol.*, 54(1):641–679, 2000.

[63] Scott Quainoo, Jordy P M Coolen, Sacha A F T van Hijum, Martijn A Huynen, Willem J G Melchers, Willem van Schaik, and Heiman F L Wertheim. Whole-genome sequencing of bacterial pathogens: The future of nosocomial outbreak analysis. *Clin. Microbiol. Rev.*, 30(4):1015–1063, October 2017.

[64] R A Gibbs. DNA amplification by the polymerase chain reaction. *Anal. Chem.*, 62(13):1202–1214, July 1990.

Appendix A

Appendix

A.1 C Implementation of Fitting Alignment

```
int fitting_alignment(const char *s1, const char *s2, size_t n, size_t m) {
    int *prev_row = (int *)malloc((m + 1) * sizeof(int));
    int *current_row = (int *)malloc((m + 1) * sizeof(int));
    int max_value = -1000;

    for (int k = 0; k <= m; k++) {
        prev_row[k] = -k;
    }

    for (int i = 1; i <= n; i++) {
        current_row[0] = 0;

        for (int j = 1; j <= m; j++) {
            int sub_cost = (s1[i - 1] == s2[j - 1]) ? 1 : -1;
```

```
            int deletion = prev_row[j] - 1;
            int insertion = current_row[j - 1] - 1;
            int substitution = prev_row[j - 1] + sub_cost;

            current_row[j] = maxi(deletion, insertion, substitution);
        }

        if (current_row[m] > max_value) {
            max_value = current_row[m];
        }

        // Swap rows
        int *temp = prev_row;
        prev_row = current_row;
        current_row = temp;
    }

    free(prev_row);
    free(current_row);

    return max_value;
}
```

A.2 C Implementation of Global Alignment

```
int global_alignment(char *s1, char *s2, size_t n, size_t m) {
    int *prev_row = (int *)malloc((m + 1) * sizeof(int));
    int *current_row = (int *)malloc((m + 1) * sizeof(int));
    int max_value = -1000;

    for (int k = 0; k <= m; k++) {
        prev_row[k] = -k;
    }

    for (int i = 1; i <= n; i++) {
        current_row[0] = -i;

        for (int j = 1; j <= m; j++) {
            int sub_cost = (s1[i - 1] == s2[j - 1]) ? 1 : -1;

            int deletion = prev_row[j] - 1;
            int insertion = current_row[j - 1] - 1;
            int substitution = prev_row[j - 1] + sub_cost;

            current_row[j] = maxi(deletion, insertion, substitution);
        }

        // Swap rows
        int *temp = prev_row;
```

```
        prev_row = current_row;
        current_row = temp;
    }
    max_value = prev_row[m];
    free(prev_row);
    free(current_row);

    return max_value;
}
```

A.3 Verilog Implementation of Processing Elements

```
module kaos_PE #(parameter score_nbits = 11, base_nbit = 2)
(
            input  clk,
            input  reset,
            input  enable,
            input  in_score_valid,
            output  logic score_valid,
            input  signed [score_nbits-1:0] left, // in
            input   signed [score_nbits-1:0] top, // del
            input   signed [score_nbits-1:0] top_left, // match/mismatch
            input   [base_nbit-1:0] que_in,
            input   [base_nbit-1:0] rf,

            output logic  [base_nbit-1:0] que_out,
            output logic  signed [score_nbits-1:0] score
           );

logic signed [score_nbits-1:0] max_score, ins, del, match;

assign match = (que_in == rf) ? top_left + 1 : top_left - 1; // match/mismatch score
assign ins = left - 1;          // insert score
assign del = top - 1;           // deletion score

// sequential elements of design
```

```
always@(posedge clk) begin//always@(*) begin
    if (reset) begin
        score <= 11'd0;
        score_valid <= 1'b0;
        que_out <= 2'b00;
    end else begin
        // only calculate and send outputs if the input from previous PE is valid and th
        if (enable && in_score_valid) begin
            score_valid <= 1'b1;
            que_out <= que_in;
            //score <= max_score;
            if (ins >= del && ins >= match) begin
              score <= ins;
            end else if (del >= ins && del >= match) begin
              score <= del;
            end else begin
              score <= match;
            end
        end else begin
            score_valid <= 1'b0;
        end
    end
end
endmodule
```