FPGA AND ASIC ACCELERATORS FOR GENOME DATA ANALYSIS

Abbas Haghi

Barcelona, 2023

Advisors: **Miquel Moretó Planas,**
Lluc Álvarez Martí

A thesis submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy

in the Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

Acknowledgements

This doctoral dissertation was carried out in the computer architecture research group of the Barcelona Supercomputing Center (BSC) and computer architecture department of the Polytechnic University of Catalonia (UPC) from 2019 to 2023.

First, I wish to express my deepest appreciation to my advisors, Dr. Miquel Moretó and Dr. Lluc Álvarez, for giving me the opportunity to pursue a PhD and for their help throughout this work.

I appreciate the support of my colleagues and friends in the computer architecture group, in particular, Santiago Marco-Sola for his guidance in most aspects of this thesis.

Lastly, the biggest thanks go to my family, especially my wife, for her endless love and support that enabled my decision to pursue a PhD.

Abstract

The continuous progress of Moore's Law in improving single-threaded performance (execution time) through clock frequency and process node improvements has slowed down due to physical limitations in silicon device physics. This has resulted in a shift towards using multicore processors to achieve performance gains. However, the use of multiprocessing is limited by Amdahl's Law, in which the sequential parts of the applications restrict the speedups in parallel systems. Consequently, the technology industry is now emphasizing specialization and developing domain-specific accelerators to enhance performance compared to general-purpose computers.

Genomics is a field that deals with the study of genes and their functions. Thanks to current and ongoing advancements, each stage of development in sequencing technologies is able to produce enormous amounts of data at an increasingly faster and cheaper way compared to its previous stage. However, the assembly and analysis of this data take extensive time on general-purpose processors, and the processor performance is growing at a much slower pace than the DNA sequencing speed. Hence, domain-specific accelerators are becoming increasingly essential in the genomics field, especially for DNA assembly. Specialized hardware computing devices designed for this specific domain can achieve significant performance improvements compared to general-purpose computers. Thus, with domain-specific accelerators, the speed of DNA assembly can keep up with the pace of DNA sequencing, allowing for quicker analysis and discoveries in the genomics field.

The main goal of this thesis is to accelerate two critical applications of DNA assembly, k-mer counting and pairwise read alignment, using FPGAs and ASICs. The first contribution of this thesis targets accelerating the k-mer counting application using FPGAs and its adaptation in a genomics application called SMUFIN, a Somatic MUtation FINder. The second and third contributions focus on exploiting FPGAs for accelerating the Wavefront Alignment (WFA), a novel pairwise read alignment algorithm for aligning DNA sequences generated by different sequencing technologies. The accelerator in the second contribution is customized for short DNA sequences of up to 300 bases, which are generated by next generation sequencing

technologies, while the accelerator in the third contribution is customized for long DNA sequences of up to 50K bases, which are generated by third generation sequencing technologies. The fourth contribution proposes an ASIC accelerator of the WFA algorithm and its integration in a RISC-V SoC. In all contributions, we analyze different parts of the application and port the most time-consuming parts to the accelerator. We also modify and re-design the remaining CPU parts to better adapt them to the accelerator code and finally propose efficient co-designed accelerated designs.

In our first contribution, the integration of our k-mer counting accelerator improves the SMUFIN k-mer counting performance by 2.14× while consuming 2.93× less energy and 1.57× less memory compared to the baseline multi-threaded software implementation. The performance of the WFA accelerators in the second and third contributions is evaluated using one and two FPGAs. Compared to the baseline multi-threaded software implementation of the WFA running on a IBM POWER9 high-performance processor, our WFA accelerator for short reads reaches performance improvements of up to 8.8× and 13.5× with one and two FPGAs, respectively, and energy efficiency improvements of up to 9.7× and 14.6× with one and two FPGAs, respectively. The WFA accelerator for long reads reaches performance improvements of up to 5.6× and 9.9× with one and two FPGAs, respectively, and energy efficiency improvements of up to 7.5× and 10.9× with one and two FPGAs, respectively. In our fourth contribution, our ASIC WFA accelerator integrated in the RISC-V SoC reaches performance improvements of up to 1076× compared to the single-threaded software implementation of the WFA on Sargantana, the RISC-V CPU of the chip.

Resumen

El continuo progreso de la Ley de Moore en la mejora del rendimiento de un solo hilo a través de la frecuencia de reloj y las mejoras en el nodo de proceso se ha visto obstaculizado debido a limitaciones físicas. Esto ha resultado en un cambio hacia el uso de procesadores multinúcleo para lograr ganancias de rendimiento. Sin embargo, el rendimiento de los procesadores multinúcleo se ve limitado por la Ley de Amdahl, que limita cualquier aceleración por las partes secuenciales del software. En consecuencia, la industria tecnológica está enfocándose en la especialización y el desarrollo de aceleradores para mejorar el rendimiento de aplicaciones específicas.

La genómica es el campo que se ocupa del estudio de los genes. Con los desarrollos actuales en tecnologías de secuenciación se produce una cantidad enorme de datos de manera más rápida y económica de lo previsto por la Ley de Moore. Sin embargo, el ensamblaje y análisis de estos datos llevan mucho tiempo en procesadores de propósito general. Por lo tanto, los aceleradores son esenciales en el campo de la genómica, especialmente para el ensamblaje de ADN. Al utilizar hardware especialmente diseñado para este dominio, como FPGAs o ASICs, los investigadores en genómica pueden lograr mejoras significativas en el rendimiento en comparación con los procesadores de propósito general. Gracias a los aceleradores, la velocidad de ensamblaje de ADN puede mantenerse al ritmo de la secuenciación de ADN, lo que permite un análisis y descubrimientos más rápidos en genómica.

El objetivo principal de esta tesis es acelerar dos aplicaciones críticas del ensamblaje de ADN, el conteo de k-mers y la alineamiento de secuencias, utilizando FPGAs y ASICs. La primera contribución se centra en acelerar la aplicación de conteo de k-mers utilizando FPGAs y su adaptación en una aplicación de genómica llamada SMUFIN. La segunda y la tercera contribución se centran en acelerar con FPGAs el novedoso algoritmo de alineamiento de secuencias WFA. El acelerador de la segunda contribución está diseñado para secuencias cortas de ADN de hasta 300 bases, que son generadas por tecnologías de secuenciación de próxima generación. El acelerador de la tercera contribución está diseñado para secuencias largas de ADN de hasta 50.000 bases, que son generadas por tecnologías de secuenciación de

tercera generación. En la cuarta contribución diseñamos un ASIC para acelerar el alinemiento de secuencias con WFA y lo integramos en un SoC RISC-V. En todas las contribuciones analizamos diferentes partes de la aplicación y trasladamos las partes más lentas al acelerador. También modificamos y rediseñamos las partes restantes de la CPU para adaptarlas al código del acelerador.

En nuestra primera contribución, nuestro acelerador de conteo de k-mers mejora el rendimiento respecto al conteo de k-mers de SMUFIN en un factor de 2,14×, mientras consume 2,93× menos energía y 1,57× menos memoria en comparación con la implementación software de referencia. El rendimiento de los aceleradores de alineamiento de secuencias de la segunda y la tercera contribución se evalúa utilizando una y dos FPGAs. En comparación con la implementación software de referencia del algoritmo WFA, nuestro segundo acelerador alcanza mejoras de rendimiento de hasta 8,8× y 13,5× con una y dos FPGAs, respectivamente, y una mejora en eficiencia energética de hasta 9,7× y 14,6× con una y dos FPGAs, respectivamente. Nuestro tercer acelerador alcanza mejoras de rendimiento de hasta 5,6× y 9,9× con una y dos FPGAs, respectivamente, y una mejor en eficiencia energética de hasta 7,5× y 10,9× con una y dos FPGAs, respectivamente. En la cuarta contribución nuestro acelerador ASIC integrado en el SoC alcanza mejoras de rendimiento de hasta 1076× en comparación con la implementación software del algoritmo WFA en Sargantana, la CPU RISC-V del chip.

Resum

El progressiu desenvolupament de la Llei de Moore per a millorar el rendiment de fils individuals mitjançant la freqüència de rellotge i les millores en el node de procés s'ha vist obstaculitzat a causa de limitacions físiques. Això ha resultat en un canvi cap a l'ús de processadors multinucli per aconseguir guanys de rendiment. No obstant això, el rendiment dels processadors multinucli està limitat per la Llei d'Amdahl, que limita qualsevol acceleració per les parts sequencials del programa. En conseqüència, la indústria tecnològica està posant l'accent en la especialització i el desenvolupament d'acceleradors per millorar el rendiment d'aplicacions específiques.

La genòmica és el camp que es dedica a l'estudi dels gens. Amb els desenvolupaments actuals en tecnologies de seqüenciació es produeix una enorme quantitat de dades més ràpidament i econòmicament que el previst per la Llei de Moore. No obstant això, l'assemblatge i l'anàlisi d'aquestes dades és molt lent en processadors de propòsit general. Per tant, els acceleradors són essencials en el camp de la genòmica, especialment per a l'assemblatge d'ADN. Utilitzant hardware dissenyat per a aquest domini concret, com ara FPGAs o ASICs, els investigadors en genòmica poden aconseguir millores significatives en el rendiment en comparació amb els processadors de propòsit general. Gràcies als acceleradors, la velocitat de l'assemblatge d'ADN pot mantenir-se al ritme de la seqüenciació d'ADN, el que permet un anàlisi i descobriments més ràpids en genòmica.

L'objectiu principal d'aquesta tesi és accelerar dues aplicacions crítiques de l'assemblatge d'ADN, el recompte de k-mers i l'alineament de seqüències, utilitzant FPGAs i ASICs. La primera contribució es centra en accelerar l'aplicació de recompte de k-mers utilitzant FPGAs i la seva adaptació en una aplicació de genòmica anomenada SMUFIN. La segona i la tercera contribució es centren en accelerar amb FPGAs el nou algoritme d'alineament de seqüències WFA. L'accelerador de la segona contribució està dissenyat per a seqüències curtes d'ADN de fins a 300 bases, que són generades per tecnologies de seqüenciació de pròxima generació. L'accelerador de la tercera contribució està dissenyat per a seqüències llargues d'ADN de fins a 50.000 bases, que són generades per tecnologies de seqüenciació de tercera generació. En la quarta contribució dissenyem un ASIC per accelerar l'alineament de seqüències amb

WFA i l'integrem en un SoC RISC-V. En totes les contribucions analitzem diferents parts de l'aplicació i portem les parts més lentes a l'accelerador. També modifiquem i redisenyem les parts restants de la CPU per adaptar-les al codi de l'accelerador

En la nostra primera contribució, el nostre accelerador de recompte de k-mers millora el rendiment del recompte de k-mers de SMUFIN en un factor de 2,14×, mentre que consumeix un 2,93× menys energia i un 1,57× menys memòria en comparació amb la implementació software de referència. El rendiment dels acceleradors d'alineament de seqüències de la segona i la tercera contribució s'avalua utilitzant una i dues FPGAs. En comparació amb la implementació software de referència de l'algoritme WFA, el segon accelerador millora el rendiment fins a 8,8× i 13,5× amb una i dues FPGAs, respectivament, i millora la eficiència energètica fins a 9,7× i 14,6× amb una i dues FPGAs, respectivament. El tercer accelerador arriba a millores de rendiment de fins a 5,6× i 9,9× amb una i dues FPGAs, respectivament, i una millor eficiència energètica de fins a 7,5× i 10,9× amb una i dues FPGAs, respectivament. En la quarta contribució el nostre accelerador ASIC integrat en el SoC millora el rendiment fins a 1076× en comparació amb la implementació software de l'algoritme WFA a Sargantana, la CPU RISC-V del xip.

List of Figures

1.1 50 years of microprocessor trend data [1]. 2
1.2 The cost of sequencing a human-sized genome [17]. 3

2.1 DNA structure [21]. 14
2.2 NGS shotgun sequencing and read production [25]. 15
2.3 An example of a Hamiltonian and Eulerian *de Bruijn* graphs [40]. 18
2.4 Calculating DP-matrix of edit distance between sequences *a* and *b*. 23
2.5 a) Distance scoring matrix; b) Similarity scoring matrix. 24
2.6 Calculating the DP-matrix of local alignment between sequences *a* and *b*. . . 26
2.7 Memory hierarchy in GPUs [74]. 30
2.8 (a) The von Neumann architecture in CPU and GPU, (b) PIM architecture [80]. 32
2.9 FPGA structure. Derived from [101]. 34
2.10 FPGA design flow. 35
2.11 RISC-V based SoC with cryptography accelerators [112]. 37

3.1 POWER9 system. 42
3.2 CAPI infrastructure [194]. 43
3.3 CPU and FPGA direct memory access. 44

4.1 SMUFIN mandatory and optional (light color) phases. 51
4.2 Procedure of the count step. 53
4.3 Proposed procedure of the count step. 54
4.4 Block diagram of FPGA k-mer generator modules. The connections are labeled with their width, in bits. 56
4.5 Example of partitioning lookup table. 58
4.6 An example of k-mer and its frequency layout after data compaction. 59
4.7 Execution time for different designs and configurations. 61

4.8 Memory and disk usage of the count step. 63
4.9 Power consumption of the count step over time. 64

5.1 Dependencies between previous wavefronts to compute one element of the new wavefront [19]. 68
5.2 Alignment of two sequences using penalties $(x, g_o, g_e) = (4, 6, 2)$. Left) SWG DP-matrix highlighting the cells that are computed by the WFA. Right) Calculation of the necessary wavefront vectors by the WFA. 69
5.3 Example of error rate, error score, and penalties. 72
5.4 Steps in the WFA co-designed accelerator of short reads and example of the compact and full CIGAR. The compact CIGAR is computed in the FPGA and only returns differences between sequences. Then, the CPU recovers the full CIGAR inserting matches by comparing both sequences in the CPU. 73
5.5 Structure and different modules of the FPGA design of the WFA accelerator for short reads. 74
5.6 a) An example WFA wavefront matrix with $k = 8$. Valid cells for penalties $(x, g_o, g_e) = (4, 6, 2)$ are marked with an X. Same colored cells of each column are the parallel inputs of the Extend and Compute modules at each clock cycle. The appropriate Extend and Compute inputs are selected using multiplexers shown in (b). 76
5.7 Architecture of the Aligner module and its sub-modules in the FPGA design of the WFA accelerator for short reads. 78
5.8 Co-design steps in the WFA accelerator for long reads. The FPGA aligns reads in batches while the CPU checks the results and performs the backtrace. The CPU Rescue computes the alignments that the FPGA has failed to compute. . 80
5.9 Synchronization of different threads. 81
5.10 Structure and different modules of the FPGA design of the WFA accelerator for long reads. 83
5.11 Implementation of the $\widetilde{M}$ wavefront matrix using RAMs in the WFA accelerator for long reads. 86
5.12 Data required by each Compute sub-module and how it is accessed as cells of a matrix (left) or words of a RAM (right). Two accesses in two consecutive clock cycles are required to read the data required by the Compute sub-modules. For simplicity, the right figure only shows connections and addresses for the first access. 87

5.13 Parallel structure of an Aligner using input RAMs and wavefront RAMs in the WFA accelerator for long reads. 89
5.14 Extend sub-module in the FPGA design of the WFA accelerator for long reads. 90
5.15 The format of writing the alignment and backtrace data in memory for the design of long reads. 93
5.16 ASM chart and equations for calculating compact CIGAR. 94
5.17 Speedup of the FPGA designs of the WFA accelerator for short reads with respect to WFA-CPU. 98
5.18 Speedup of the FPGA designs for short reads with WFA-CPU (top) and one FPGA (bottom) for multi-threaded runs over single-threaded WFA-CPU. . . . 100
5.19 Energy improvement of the FPGA designs for short reads with respect to WFA-CPU. 100
5.20 Speedup of the FPGA designs for long reads with respect to WFA-CPU. . . . 101
5.21 Speedup of the FPGA designs for medium reads with respect to WFA-CPU. . 102
5.22 Speedup of the FPGA design 13 for long reads with different number of Aligners and with one and two FPGAs with respect to the same design with one Aligner and one FPGA. 102
5.23 Speedup of the FPGA designs for long reads with WFA-CPU (top) and one FPGA (bottom) for multi-threaded runs over single-threaded WFA-CPU. . . . 103
5.24 Energy improvement of the FPGA designs for long reads with respect to WFA-CPU. 104
5.25 Energy improvement of the FPGA designs for medium reads with respect to WFA-CPU. 105
5.26 Speedup (left) and energy improvement (right) of the FPGA designs of long reads with respect to WFA-CPU when applying real input sets. 107

6.1 SoC architecture including the RISC-V CPU, the WFAsic accelerator and their connections. 112
6.2 WFAsic structure and different modules. 114
6.3 The format of writing the alignment and backtrace data in memory for the WFAsic. 118
6.4 Accelerator layout. The size is 1330um×1200um with all the connectivity on the right side. 119
6.5 Speedup with respect to the CPU-scalar code. 121
6.6 Speedup of adding Aligners with respect to one Aligner. 122

6.7 Performance comparison between WFAsic with one Aligner of 64 parallel sections (64PS) performing data separation (Sep), two Aligners of 32 parallel sections (32PS) performing data separation (Sep), and one Aligner of 64 parallel sections (64PS) without performing data separation (No Sep). 123

List of Tables

3.1 Input sets of second, third and fourth contributions. 47

4.1 FPGA resources utilization (%) for the CAPI-related IP cores and for the proposed k-mer counting accelerator. 60
4.2 Execution time and disk space requirements of the count and unify steps with the prune step enabled and disabled. 62

5.1 FPGA designs, resource utilization, number of Aligners in each FPGA and synthetic inputs used in the evaluation. 96
5.2 Duration (in clock cycles) of alignment, backtrace and extracting reads of one sequence pair and maximum efficient Aligners in each FPGA. 99
5.3 Real input sets specifications. 106
5.4 FPGA design configurations evaluated with real input sets. 106
5.5 Computed and equivalent GCUPS achieved by our co-designs for different inputs. 108
5.6 Peak GCUPS of different exact SWG FPGA accelerated methods. 109

6.1 Maximum number of Aligners for each input based on the execution cycles of reading and aligning one pair of reads. 122
6.2 GCUPS and area comparison of different platforms/methods aligning reads of 10Kbp. 125
6.3 GCUPS comparison of WFA-1FPGA and WFAsic per Aligner. 125

Table of contents

Acknowledgements i

Abstract iii

Resumen v

Resum vii

List of Figures ix

List of Tables xiii

Contents xvii

1 Introduction 1

1.1 Thesis Objectives and Contributions 4

1.1.1 Accelerating K-mer Counting 5

1.1.2 Accelerating Pairwise Read Alignment 7

1.2 Thesis Outline 10

2 Background 13

2.1 Genomics 13

2.1.1 DNA Sequencing 15

2.1.2 DNA Assembly 17

2.2 Hardware Accelerators 28

2.2.1 GPU 29

2.2.2 PIM 31

2.2.3 FPGA 33

2.2.4 ASIC 36

2.2.5 Genomics Hardware Accelerators 38

3 Experimental Methodology **41**
3.1 Platforms 41
3.1.1 POWER9 Platform 41
3.1.2 ASIC Platform 45
3.2 Baselines and Input Sets 45
3.2.1 K-mer Counting in SMUFIN 45
3.2.2 WFA for Pairwise Read Alignment 46

4 K-mer Counting FPGA Accelerator **49**
4.1 Introduction 49
4.2 Background 50
4.2.1 DNA Reads, K-mers and K-mer Counting 50
4.2.2 SMUFIN Overview 50
4.2.3 SMUFIN K-mer Counting Structure 51
4.3 Acceleration Method of K-mer Counting in SMUFIN 53
4.3.1 Prune Step 53
4.3.2 Count Step 54
4.3.3 Unify Step 59
4.4 Evaluation and Results 59
4.4.1 Experimental Setup 59
4.4.2 Results 60
4.5 Conclusions 64

5 WFA FPGA Accelerator **65**
5.1 Introduction 65
5.2 Background 67
5.3 WFA Accelerator for Short Reads 73
5.3.1 Extractor Module 74
5.3.2 Collector Module 74
5.3.3 Aligner Module 74
5.4 WFA Accelerator for Long Reads 79
5.4.1 Hardware/Software Co-design Structure 80
5.4.2 Extractor Module 84

5.4.3 Aligner Module . . . 84
5.4.4 Collector Module . . . 91
5.4.5 Backtrace in CPU . . . 92
5.5 Evaluation and Results . . . 95
5.5.1 Experimental Setup . . . 95
5.5.2 Results of Short Reads for Synthetic Input Sets . . . 98
5.5.3 Results of Long Reads for Synthetic Input Sets . . . 101
5.5.4 Results of Long Reads for Real Input Sets . . . 105
5.5.5 Performance Comparison . . . 107
5.6 Conclusions . . . 109

6 WFA ASIC Accelerator **111**
6.1 Introduction . . . 111
6.2 System on Chip Architecture . . . 112
6.3 WFAsic Accelerator . . . 113
6.3.1 Memory implementations . . . 113
6.3.2 Extractor Adaptation . . . 115
6.3.3 Collector Adaptation . . . 115
6.3.4 Backtrace . . . 117
6.4 Evaluation . . . 119
6.4.1 ASIC Synthesis and Place and Route . . . 119
6.4.2 FPGA Prototype Performance Results . . . 120
6.5 Conclusions . . . 125

7 Conclusions **127**
7.1 Goals, Contributions and Main Conclusions . . . 127
7.2 Future Work . . . 130
7.3 Publications . . . 131
7.4 Financial and Technical Support . . . 133

Bibliography **135**

Chapter 1
Introduction

The progress of Moore's Law is being impeded by physical limitations in silicon device physics. According to the data collected and plotted by Horowitz et al. [1] in Figure 1.1, since 2004 the frequency has not scaled anymore as the Dennard Scaling was no longer viable due to increasing leakage power and chip temperature in the advanced technology nodes, where the size of transistors has shrunk to the point where physical effects have a significant impact on power consumption. As a result, improving single-threaded performance (execution time) through clock frequency and process node improvements is no longer effective. Slight increases in single-thread performance after 2004 are achieved with developments in computer architecture [2] and clever power management by dynamic clock frequency adjustments [3]. Therefore, instead of increasing frequency, the focus has shifted to using multicore processors to achieve performance gains. In this trend, cores work in parallel at lower frequencies and share resources, and so consume less power. However, the use of multiprocessing is limited by Amdahl's Law, which states that any speedup will be restricted by the sequential parts of the software. Hence, instead of relying solely on general-purpose processors, the technology industry is now emphasizing specialization and developing domain-specific accelerators [4].

A domain-specific accelerator refers to a specialized hardware computing device designed for a particular domain of applications. These accelerators are capable of significantly enhancing the performance/cost and performance/W ratios when compared to general-purpose computers. To accelerate domains of applications, different types of accelerators such as Graphics Processing Units (GPUs), Processing in Memories (PIMs), Field Programmable Gate Arrays (FPGAs), and Application Specific Integrated Circuits (ASICs) are available, each with varying degrees of development cost, programmability, and efficiency trade-offs. These accelerators have been developed for various tasks, genomics being one of them [5].

Genomics is a field of biology that deals with the study of genes and their functions. Genomics is changing our understanding of humans, evolution, diseases, and medicine. The

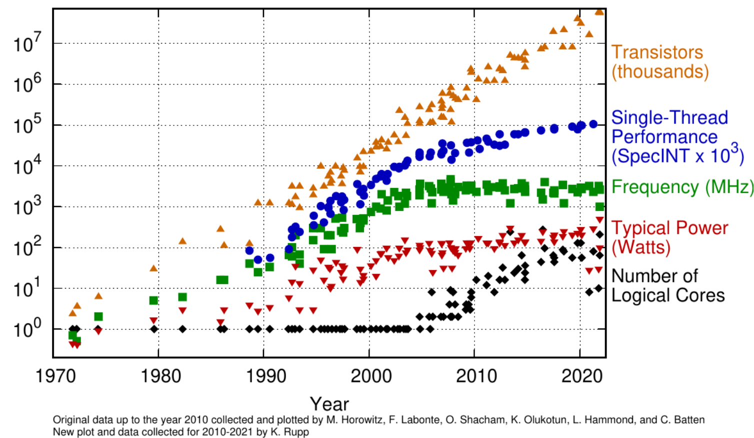


Figure 1.1: 50 years of microprocessor trend data [1].

genomics data is not only valuable to find the DNA mutations causing cancer [6], Alzheimer [7], genetic disorders [8], and autism [9], but also to better understand the human biology and the differences between individuals [10–12], which opens the door to personalized medicine in which each patient takes specific and effective medical treatments regarding to their individual characteristics or genetic information [13, 14].

Genomics is divided to two very general categories of *DNA sequencing* and *DNA assembly*. DNA sequencing is a chemical process, used to obtain the DNA in a format that can be analyzed by scientists. Today, the growth of DNA sequencing using new technologies is significantly outpacing the growth of computation power predicted by the Moore's Law and, as shown in Figure 1.2, have drastically reduced the cost of DNA sequencing compared to just a few years ago by several orders of magnitude. The Illumina NovaSeq 6000 system can sequence about 48 human whole genomes at 30× genome coverage (the average number of times a base is sequenced) in about two days [15, 16].

After DNA sequencing, the first step to make it usable for scientific analysis consists of DNA assembly which is reconstructing the DNA sequence. Advances in genomics technology have allowed researchers to analyze large amounts of genomics data quickly and accurately. However, the analysis of genomic data is computationally intensive, and general-purpose processors are often not powerful enough to handle the processing requirements. For example, reconstructing the sequenced data of a single human genome, which consists of 90GB of data, requires over 32 hours on a 48-core Intel Xeon processor [16]. Analyzing this amount of data using CPU based platforms is time-consuming and falls the speed of DNA assembly far behind that of DNA sequencing.

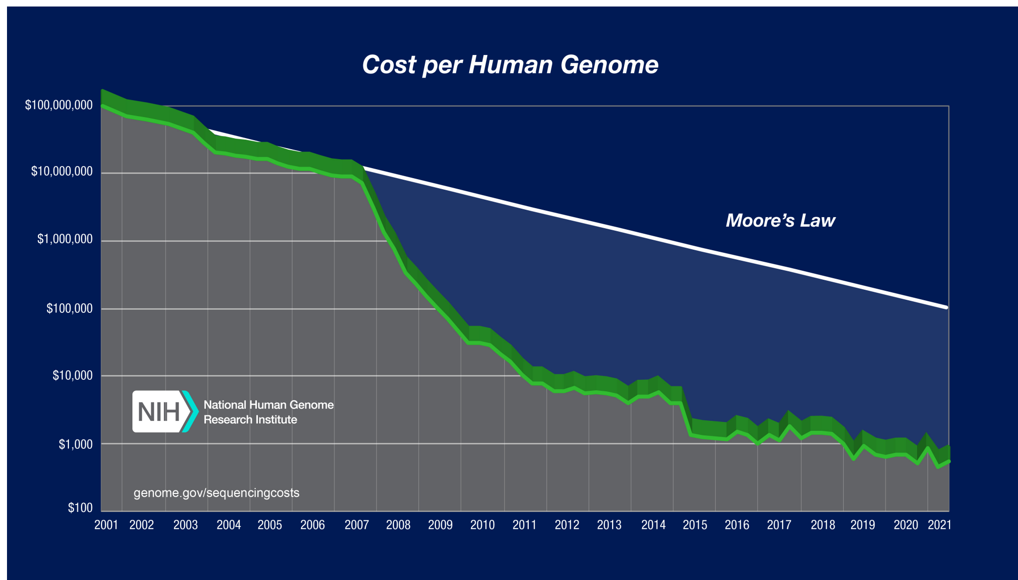


Figure 1.2: The cost of sequencing a human-sized genome [17].

Hence, as mentioned earlier, domain-specific accelerators in genomics domain for DNA assembly are considered as promising solutions for improving performance as well as reducing power consumption. Among the available accelerators, FPGAs are of more interest due to their performance, scalability, energy efficiency, and reconfigurability. FPGAs are integrated circuits that can be programmed to perform specific tasks, making them highly versatile and efficient for a wide range of applications. One of the main advantages of using FPGAs in genomics applications is their high performance. FPGAs can achieve processing speeds (throughput) that are significantly higher than those of CPUs and GPUs. This means that FPGAs can significantly reduce the time required to process large genomic datasets, making it possible to perform complex genomics analyses in a timely manner. Also, FPGAs are highly scalable, which means they can be configured with multiple processing units, which makes them highly efficient for parallel processing. This means that FPGAs can be used to scale up genomics applications to handle large datasets, making them ideal for large-scale genomics projects. Another advantage of using FPGAs is their low power consumption. Although FPGAs can be more expensive than traditional computing platforms, they can be more cost-effective in the long run. FPGAs consume less power than traditional computing platforms, which can result in significant cost savings over time. This is particularly important for large-scale genomics projects, where the energy costs of traditional computing platforms can be prohibitively high. Additionally, FPGAs can be customized to meet the specific requirements of a particular genomics application, which can result in increased efficiency and cost savings. FPGAs can be reprogrammed to perform

different tasks, making them highly flexible. This is useful for genomics applications, which often involve developing new algorithms or modifying existing ones.

While FPGAs are reconfigurable, ASICs are custom-built integrated circuits that are designed and optimized to perform a specific function. ASICs are also good options for accelerating genomics applications. ASICs are more energy efficient compared to the FPGAs and reach higher working frequencies, however, they are more costly and have more complex design flow.

Despite the fact that FPGAs and ASICs have many advantages in genomics applications, there are some challenges associated with their use. One of the main challenges of using them in genomics applications is the complexity of designing them. In addition, due to the hardware resource restrictions of FPGAs and ASICs, they may not have enough resources (i.e. memory) to handle some of the most complex applications. This means that they may not be suitable for all genomics applications. Hence, wise decisions should be made to select the genomics applications or a part of them which are good targets for the FPGA/ASIC implementation.

Hardware/software co-design using FPGAs and ASICs can offer several benefits for genomics applications. FPGA and ASIC hardware/software co-designs provide the required flexibility, in terms of performance, resources/memory usage and design complexity, for implementing genomics applications, resulting in highly optimized systems that can handle large-scale genomics data processing. Additionally, hardware/software co-design can result in reduced power consumption, as the custom hardware can be designed to operate at low power.

1.1 Thesis Objectives and Contributions

The objective of this thesis is accelerating important genomics applications. To do this, we select important genomics applications, we identify their most time-consuming steps as best candidates for acceleration, we evaluate the feasibility and the potential performance improvement, and we propose hardware/software co-designs for them. After that, the software codes are modified and customized for the accelerators integration in a hardware/software co-designed approach.

The selected target applications cover k-mer counting application and its implementation in a *de novo* approach, SMUFIN [18], and a novel pairwise read alignment algorithm, WFA [19]. These applications are computationally intensive and require significant amounts of memory and processing power to complete in a reasonable time. Traditional computing architectures, such as CPUs, are often not efficient enough to handle these tasks in a reasonable amount

of time. However, using FPGAs and ASICs which offer high speed, power efficiency and scalability are promising solutions for accelerating these applications.

This thesis does a total of four contributions; one contribution to accelerate the k-mer counting and its integration within the SMUFIN, and three contributions to accelerate WFA. More specifically, the four contributions of this thesis are:

- Accelerating the k-mer counting.
- Accelerating pairwise read alignment.
  - Aligning short reads using FPGAs.
  - Aligning long reads using FPGAs.
  - Aligning reads using ASICs.

1.1.1 Accelerating K-mer Counting

The first contribution of this thesis, in general, accelerates the genomics k-mer counting application through an FPGA-based hardware/software co-design, and in particular its usage in the SMUFIN algorithm [18]. K-mer counting involves counting the occurrence of all possible substrings of length k in a sequence. K-mers are used in a variety of genomics applications, including genome assembly and gene expression analysis. SMUFIN is a state-of-the-art algorithm for identifying somatic mutations in DNA samples. The algorithm consists of separate and sequential steps, with the first step being k-mer counting. This step reads DNA sequences stored in several files, generates all possible k-mers of all reads, and then counts the number of appearances of each distinct k-mer in the whole dataset. Since a huge amount of data should be processed, k-mers are processed in batches, and the results of each step are stored on the disk as intermediate data. These intermediate data are later merged to generate a single histogram table.

The k-mer counting step is considered data-intensive, and generates and consumes more than 500GB of intermediate data. One of the main challenges in software implementation of data-intensive applications like k-mer counting is the high computational requirements and the large amount of memory resources required. The sequential execution of the algorithm on traditional CPUs can be slow, and parallelizing the code can introduce overheads and dependencies that limit scalability. Moreover, the memory requirements for k-mer counting can be substantial, which can lead to issues with memory allocation and management in software implementations. Additionally, the use of high-performance computing resources can lead to high energy consumption, making the analysis of genomics data unsustainable in the long run. Offloading some portions of these applications to the FPGAs can overcome the limitations of

software implementation, as FPGAs are capable of performing parallel computations at a much higher speed and with much lower energy consumption. This leads to faster and more efficient data analysis, enabling researchers to analyze larger datasets and ultimately accelerate the pace of genomics research. However, optimizing the performance of the algorithm requires a deep understanding of the underlying hardware and software architectures.

In this contribution, we investigate the bottlenecks of the k-mer counting software implementation of the SMUFIN algorithm and redesign it to improve its performance. Additionally, we explore portions of the code that are relatively slow in software and could be parallelized using FPGA. Finally, we propose an efficient hardware/software co-design for k-mer counting in SMUFIN that improves performance, reduces power consumption, and requires less memory while producing the same output. To achieve these goals, we make several contributions. Firstly, we present an register-transfer-level (RTL) design for the FPGA that accelerates the processes of extracting reads and generating k-mers. Secondly, we modify the software algorithm, where it counts the occurrence of all k-mers, to remove dependencies between parallel threads, reduce overheads, and use less memory. Finally, we introduce data compaction mechanisms to minimize memory and disk requirements of the intermediate data. We integrate the co-designed accelerator into an adapted version of SMUFIN, and evaluate it on a high-performance computing node with a dual-socket POWER9[1] processor and an FPGA.

Our results show that the co-design of the k-mer counting of SMUFIN, using one Xilinx Virtex UltraScale+ (XCVU3P) FPGA running at 250MHz, outperforms the 160-threaded CPU-only design, running on 40 cores of the POWER9 High Performance Computing (HPC) machine, by 2.14×, while consuming 2.93× less energy and 1.57× less memory. This co-design is not only applicable to the SMUFIN algorithm but can also be adapted for use in other algorithms that require k-mer counting. Our work has important implications for genomics research and for the development of more efficient and effective algorithms for identifying somatic mutations in DNA samples. By improving the performance and reducing the power consumption and memory requirements of the SMUFIN algorithm, our work contributes to the development of more efficient and sustainable genomics research.

[1]IBM, POWER9 and OpenCAPI (Open Coherent Accelerator Processor Interface) are registered trademarks. Other product or service names may be trademarks or service marks of IBM or other companies. A current list of IBM trademarks is available on ibm.com/trademark.

1.1.2 Accelerating Pairwise Read Alignment

Pairwise read alignment involves aligning two sequences of DNA reads to identify regions of similarity and differences. This technique is used in genome assembly and variant detection. This thesis makes three contributions towards accelerating pairwise read alignment, which are detailed in the following sub-sections.

Aligning Short Reads using FPGAs

The second contribution of this thesis aims to accelerate the pairwise read alignment of short DNA reads (up to 300 bases) through an FPGA-based hardware/software co-design. In most DNA sequence analysis pipelines, the first step is to locate each read in the reference genome. This process is known as read mapping and involves two main steps. The first step, *seeding*, involves filtering and minimizing the potential locations of the reads in the reference genome. The second step, *seed extension*, entails aligning the reads with the potential candidate locations of the reference genome. This contribution specifically focuses on the seed extension step of read mapping which is also known as *pairwise read alignment*.

Next Generation Sequencing (NGS) technologies generate massive amounts of short reads, typically ranging from 50 to 300 base pairs in length. These short reads pose a significant challenge in terms of aligning them to a reference genome. Variants of Smith-Waterman (SW) algorithm are commonly used for the pairwise read alignment, but unfortunately it is computationally intensive and requires significant amount of memory. This is because these algorithms are based on Dynamic Programming (DP) and require $O(n^2)$ execution time and memory proportional to the sequence length n. Hence, by increasing the sequence length, the computational requirements of the SW increases drastically. Running SW alignment for millions of short reads can be time-consuming and resource-intensive, making it a bottleneck in the overall analysis pipeline. Recently, the WaveFront Alignment (WFA) [19] algorithm has been proposed as a breakthrough solution, which runs in $O(n{\cdot}s)$ time, proportional to the sequence length n, and the error score between sequences s. It utilizes a novel approach that only computes a reduced number of DP-matrix cells to find the optimal alignment.

In this contribution, we examine the software code of the WFA algorithm, conduct profiling, and identify functions that could be parallelized and moved to the FPGA. We distribute the workload across two available accelerators and modify the software code to schedule tasks between them and verify accelerator results. In addition, some alignment tasks still need to be performed by the CPU and cannot be moved to the FPGA. As a result, we propose the first

FPGA-based hardware/software co-designed accelerator for the wavefront alignment algorithm. This accelerator computes sequence alignments for pairs of sequences and produces compacted results that make communication between the CPU and FPGA more efficient.

Our WFA accelerator design which is suitable for aligning short length DNA sequences, using Xilinx Virtex UltraScale Plus (XCVU37P) FPGAs running at 200MHz, achieves speedups of 4.5× to 8.8× with one FPGA and 8.2× to 13.5× with two FPGAs, compared to the 64-threaded CPU-only implementation of the reference WFA running on 16 cores of the POWER9 HPC machine. Additionally, the FPGA accelerator reduced energy-to-solution by 6.1× to 9.7× with one FPGA and by 11.4× to 14.6× with two FPGAs. The accelerator is also scalable, with multiple aligner cores that can work in parallel to compute sequence alignments, depending on available resources. Furthermore, the design is parameterizable, with configurable maximum supported read length and error score values between the reads, making it adaptable to different input sets with varying characteristics.

Aligning Long Reads using FPGAs

The third contribution of this thesis aims to accelerate the pairwise read alignment of long DNA reads (more than 10K bases) through an FPGA-based hardware/software co-design. While NGS technologies have revolutionized the field of genomics, there are still some limitations that have led to the emergence of third generation sequencing technologies, which provide reads with lengths on the order of a few thousand base pairs.

NGS technologies have difficulties in sequencing long DNA fragments, calling large structural variants, and impose difficulties in assembling repetitive genome regions. On the other hand, third generation sequencing technologies can produce ultra-long reads, which makes it easier to sequence these previously difficult-to-sequence regions. Moreover, the ability of third generation sequencing technologies to produce longer reads and real-time sequencing data, could reduce the time and cost associated with DNA analysis. This could make third generation sequencing technologies more attractive for large-scale genomics projects.

Long read aligners are critical tools for accurately mapping long reads generated by third generation sequencing technologies to reference genomes. However, the implementation of long read aligners, particularly on FPGAs, can be challenging due to the large amounts of data involved. One of the main difficulties in implementing long read aligners on FPGAs is the high memory requirements. This often results in the need for off-chip memory access, which can significantly slow down the alignment process. In addition, implementing computations of the

long reads alignment algorithms on FPGAs can be challenging, particularly when trying to balance the trade-off between performance and resource usage.

To address these challenges, many researchers have turned to heuristic methods for implementing long read aligners on FPGAs. Heuristic methods involve using simplified algorithms that sacrifice some accuracy in exchange for improved computational efficiency. While these methods may not produce optimal results, they can still provide high-quality alignments that are suitable for many downstream analysis.

In this contribution, we propose a modified design for the WFA accelerator, which extends its functionality to medium and long reads. Our approach involves intelligent usage of FPGA RAMs to store sequences and data structures of the WFA algorithm, as well as a re-organization of tasks between the FPGA and CPU in hardware/software co-design. Unlike many long read aligners, our design implements the optimal WFA alignment algorithm, rather than relying on heuristics.

Our WFA accelerator design which is suitable for aligning long length DNA sequences, using Xilinx Virtex UltraScale Plus (XCVU37P) FPGAs running at 200MHz, achieves significant speedups, ranging from 2.6× to 5.6× with one FPGA, and 2.7× to 9.9× with two FPGAs, compared to the 64-threaded CPU-only implementation of the reference WFA running on 16 cores of the POWER9 HPC machine. We also reduce energy-to-solution by 3.6× to 7.5× with one FPGA, and 3.7× to 10.9× with two FPGAs. Similar to our design for aligning short reads, this design is also scalable and parameterizable. The parameters of this design allow users to balance resource utilization between FPGA RAMs and lookup tables. This provides efficient and even utilization of resources, which consequently maximizes parallelization.

Aligning Reads using ASICs

The fourth contribution of this thesis aims to accelerate the pairwise read alignment of DNA reads using an ASIC implementation of WFA algorithm in a RISC-V SoC. Having genomics ASIC accelerators inside a RISC-V SoC offers several benefits over a system with FPGA. Firstly, ASICs offer higher performance and energy efficiency compared to FPGAs. This is because ASICs are specifically designed to perform a particular task, whereas FPGAs are general-purpose devices that can be reconfigured to perform different tasks. Therefore, ASICs can provide higher computational power and lower energy consumption for genomics applications, making them a more cost-effective solution in the long run. Moreover, integrating genomics ASIC accelerators into a RISC-V SoC can improve system-level integration and reduce the overall system complexity. By having ASICs integrated into the SoC, the communication

between the different components can be optimized, leading to faster and more efficient data transfer. Additionally, having an ASIC-based solution can also lead to a reduction in the form factor of the overall system, which can be beneficial in applications where space is a constraint.

However, porting FPGA code to ASIC can be a challenging task due to the fundamental differences between the two technologies. FPGAs are reconfigurable devices, and the design flow involves programming the logic and routing resources of the device. In contrast, ASIC design involves a process of physical implementation, which involves the design of custom logic cells and interconnects. As a result, porting FPGA code to ASIC can require significant modifications to the original design, making it a complex and time-consuming process.

In this contribution, we modify our FPGA-based long-reads design to be implemented on an ASIC. We achieve this by exploring various design configurations to ensure that the design meets the budget constraints of the ASIC while maintaining the highest possible level of parallelization. We also make changes to the previous design to make it compatible with the requirements of the SoC. Additionally, we modify the hardware modules and components to optimize the ASIC operating frequency. Moreover, we adapt the C code running on the RISC-V processor to work with the new hardware implementation, with the ultimate goal of achieving the best single-thread performance for the entire co-design.

Our ASIC design after synthesis and Place and Route (PnR) in GlobalFoundries 22nm technology fits in an area of 1.6mm^2 and reaches a frequency of 1.1GHz. The ASIC accelerator is configured using a standard Linux driver and Application Programming Interface (API). In addition, the ASIC accelerator runs as an independent process in parallel to other CPU processes. Based on the performance analysis using an FPGA prototype, the integrated ASIC accelerator provides performance improvements of up to 1076$\times$ compared to the CPU implementation of the WFA running on Sargantana [20], the in-order single-threaded RISC-V CPU of the chip which also runs at 1.1GHz.

1.2 Thesis Outline

The contents of this thesis are organized as follows:

- Chapter 2 presents the background and definitions in genomics, different hardware accelerators and their architectures, and related works in accelerating genomics with hardware accelerators.

- Chapter 3 presents the experimental methodology used within the four contributions of this thesis.
- Chapter 4 presents the first contribution of this thesis, which provides an FPGA-based hardware/software co-design for accelerating k-mer counting and its implementation in the SMUFIN application.
- Chapter 5 presents the second and third contributions of this thesis, which provides FPGA-based hardware/software co-designs for accelerating pairwise read alignment of short and long reads using WFA algorithm.
- Chapter 6 presents the fourth contribution of this thesis, an ASIC implementation of WFA for pairwise read alignment in a RISC-V SoC.
- Finally, Chapter 7 concludes by summarizing the contributions of this thesis, listing the publications resulting from it and considering what future potential research directions it suggests.

Chapter 2
Background

This chapter presents the background and outlines the related work in the topics addressed in this thesis. This chapter is divided in two main sections. First, in Section 2.1 we explain the required background on genomics and the necessary definitions to understand this thesis. It includes Section 2.1.1, which is about DNA sequencing, and Section 2.1.2, which is about DNA assembly methods and their fundamental operations which are the focus of this thesis, i.e., k-mer counting and pairwise sequence alignment. Then, in Section 2.2 we describe different hardware accelerators and their architectures, in which Section 2.2.1 provides context for GPUs, Section 2.2.2 describes processing in memory, and Sections 2.2.3 and 2.2.4, respectively, discuss FPGA and ASIC accelerators, which are the accelerators we used in this work. Finally, Section 2.2.5 reviews the main hardware accelerators for genomics applications which are most related to this thesis.

2.1 Genomics

The DNA of each living species and virus holds the genetic instructions (codes) which determine the characteristics of that species. DNA is made of two twisted and connected strands of chained nucleotides. Nucleotides are made of sugar and phosphate molecules, and one of four types of nitrogen bases, i.e., adenine (A), cytosine (C), guanine (G) and thymine (T). The chain of each DNA strand is formed by alternating covalent bounds between sugars and phosphates of consecutive nucleotides [21, 22]. Then, the two strands are connected by forming hydrogen bonds between complementary bases of the strands, i.e., A with T and C with G and vice versa. A pair of those bonded bases is called a base pair (bp). Figure 2.1 shows the structure of a DNA [23].

The sequence of bases in the DNA strands determines the biological instructions of the DNA. For example, the sequence ATCGTT might instruct for blue eyes, while ATCGCT might

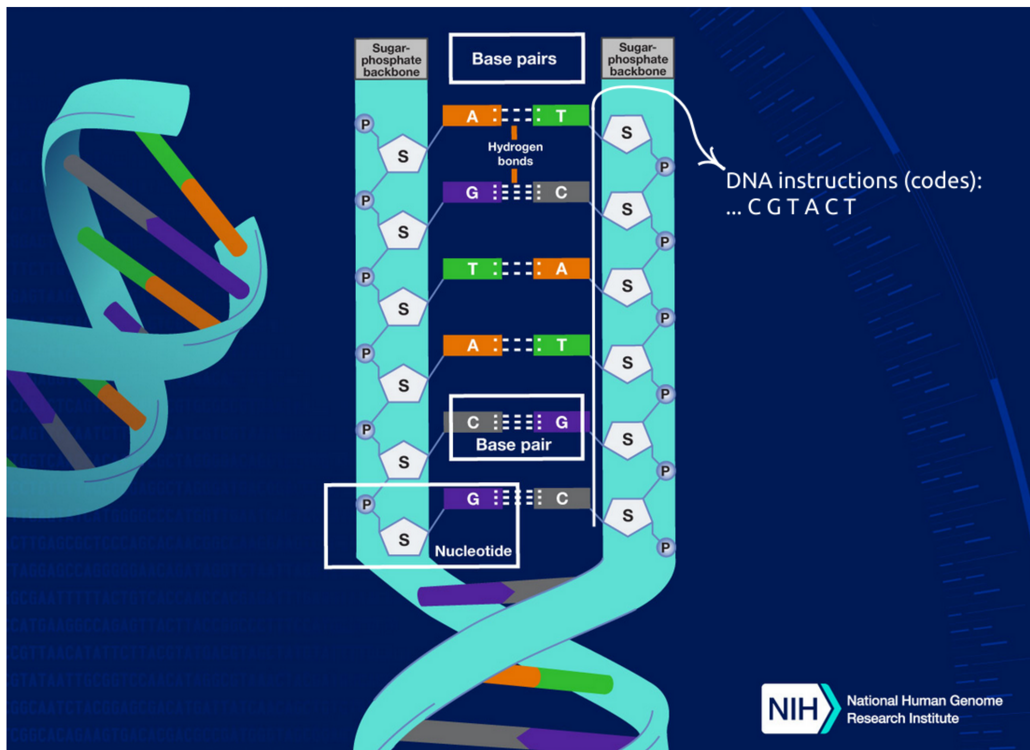


Figure 2.1: DNA structure [21].

instruct for brown. The complete human instruction genome is made of 3 billion bases and 20,000 genes on 23 pairs of chromosomes [21]. The human reference genome is developed and maintained by the Genome Reference Consortium and the current version is called the GRCh38 [24].

In order to study and analyze DNA, it is necessary to decode its instructions. Instructions are actually DNA sequences, and each instruction is related to a section of the DNA sequence, called gene. Decoding instructions refers to identifying which instruction affects which organs and in what manner. However, before decoding, the initial step is to obtain DNA sequences, i.e., for a human, the 3 billion bases of the DNA in order. The techniques of obtaining DNA sequences are developing every day, and are getting faster, more accurate, and more cost-effective.

Obtaining DNA sequences is a two-step approach. In the first step, *DNA sequencing*, the DNA is sampled and sequenced in short or long lengths. In the second step, *DNA assembly*, the sequenced data is put together to reconstruct the whole genome.

2.1.1 DNA Sequencing

Nowadays, DNA sequencing is the most widely used way to obtain the DNA sequences in a format that can be analyzed by scientists. DNA sequencing is the process of sampling molecules of the DNA into a large number of fragments, called *reads*, and reading the sequence of bases in each read. The size of the reads depends on the sequencing technology. Apart from the first generation DNA sequencing technique, there are two DNA sequencing technologies in the market, known as *next generation DNA sequencing* and *third generation DNA sequencing*.

First Generation DNA Sequencing

In 1977, the complete genome of an organism was sequenced for the first time using the first generation of sequencing technologies. Sanger sequencing, the widely preferred method at the time, eventually produced a reference human genome in 2003. Sanger reads fragments of the DNA sequence with a few thousand base pairs long, which are useful for sequencing unknown genomes where no reference genome is available. However, this process requires high cost and long time to sequence DNA, thereby limiting the usage of these sequencers [14].

Next Generation DNA Sequencing

Since 2005, next generation sequencing technologies have proliferated. Instead of producing the long and slow-to-extract reads, NGS technologies use shotgun sequencing, in which the DNA is cut in many small fragments and, from those fragments, reads are extracted in parallel. Thus, this technique makes genome sequencing much easier, faster and cheaper. A sequencing run produces many overlapping reads [25, 26] because first the DNA is replicated and then each copy is cut somewhat randomly, resulting in many overlapping fragments [27]. Figure 2.2 shows the NGS shotgun sequencing and the read production process.

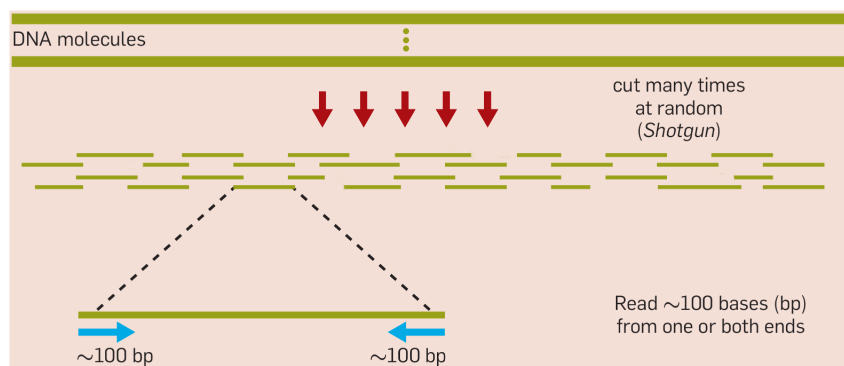


Figure 2.2: NGS shotgun sequencing and read production [25].

NGS technologies are typically capable of producing output reads with short lengths of between 50 and 300 base pairs, so they are also know as Short Read Sequencing (SRS) technologies [26]. As sequencing technologies are not 100% accurate, sequencing machines report a quality score for each sequenced base of the DNA [25]. The higher the quality score is, the more accurate the base call is. A quality score of 20 means that 1 error might occur in every 1000 sequenced bases. The accuracy of a base with a quality score of 20 is 99% and that of 30 is 99.9% (1 error in 10,000 bases) [28]. Low quality bases could be neglected or substituted with other bases depending on the application.

NGS sequencing machines fragment DNA into small pieces, amplify them, and then sequence the resulting fragments simultaneously to achieve a high coverage of the genome. *Coverage* or *depth* refers to the number of times a given base in a genome is sequenced by high-quality reads. In other words, it is the number of times that a given position in the genome is covered by a high-quality read [29]. The quality of the sequenced bases is measured using a Phred quality score, which is a logarithmic scale that reflects the probability of an error in base calling. A base with a quality score of at least 20 is considered a high-quality base [30]. NGS coverage level for whole human DNA sequencing is recommended to be between 30$\times$ and 50$\times$ depending on the application [31].

NGS sequencing technologies usually provide sequenced reads in FASTQ file format. Each read in a FASTQ file is represented by four lines. The first line determines the sequence identifier and information about the sequencing run and the cluster, such as machine ID, run session, lane, etc. The second line contains the sequence data (sequenced bases). The third line is a separator between the data of the second and the fourth lines. It starts with the character "+", and is optionally followed by the same sequence identifier. The fourth line includes the quality scores for each base in line 2 [32].

Illumina, Ion Torrent, Roche 454 and SOLID are some of the available and widely used NGS sequencing technologies nowadays. Although NGS has very good output quality (error rates around 0.1%), it fails to generate sufficient overlapping sequences from the DNA fragments. This constitutes a major challenge for reconstructing a highly complex and repetitive genome like the human genome without using a reference genome (*de novo*). Also, the detection of large sequence changes is another difficulty encountered by NGS [26].

Third Generation DNA Sequencing

To overcome the limitations of NGS technologies, third generation sequencing technologies produce long reads with lengths of about 10Kbp. This new technology, which entered the

market in 2011, is also called long read sequencing (LRS) technology. As third generation sequencing machines provide longer reads, they make DNA assembly significantly faster, easier and more accurate. More importantly, the long reads also enable the identification of long insertions and deletions in the genome. However, LRS technologies have higher error rates than the ones produced by NGS technologies, so the assembly of long reads becomes more complicated. Nowadays, a big effort is being made to lower the error rates of LRS technologies, which is currently around 5% in most cases but can reach less than 1% in the most advanced and costly LRS approaches. Despite the fact that LRS technology is racing to overtake NGS technology in industry, it is not yet fully accepted by the market and, hence, NGS technology is still dominating the market due to the aforementioned reasons [26].

2.1.2 DNA Assembly

After DNA sequencing, the first step to make it usable for scientific analysis consists of reconstructing the DNA sequence. There are two general methods to perform this step. In the first one, called reference-guided method, reads are aligned against a reference genome. In other words, the reads are assigned to locations in the DNA sequence based on the similarity between the reads and the reference genome. The second method, called *de novo*, avoids using a reference genome. In other words, genome is reconstructed from the scratch. In this method, to reconstruct the genome, reads which are next to each other in the genome are identified and assembled like a puzzle. *De novo* genome sequence assembly is important to reconstruct the genome of uncharacterized genomes and also to identify the genome sequence of individuals in a reference-unbiased way [33].

De novo Read Assembly

Existing *de novo* sequence assembly algorithms can be categorized in five branches: greedy, Overlap Layout Consensus (OLC), *De Bruijn* Graph (DBG), string graph, and hybrid algorithms [34]. OLC and DBG are two most common methods for *de novo* assembly algorithms. The string graph algorithm is a variant of OLC that makes it more suitable for assembling long reads. The hybrid algorithm mixes various algorithms to reduce the number of contigs (sets of DNA segments or sequences that overlap in a way that provide a contiguous representation of a genomic region) and errors produced by other algorithms [35].

In the OLC method, the first step is to detect overlaps greater than a certain threshold between all the reads, followed by the construction of an overlap graph. Overlaps can be

detected by pairswise read alignment techniques, which are explained later in this chapter. Then, overlapping reads are merged to form contigs. After that, according to the consensus of all reads which form a contig, the sequence is inferred. The human genome was primarily constructed using OLC algorithms [33]. Popular OLC-based assembly algorithms include PCAP [36], AMOS [37], Arachne [38] and Celera [39].

A *de Bruijn* graph is a representation of all the k-mers of all the reads of the genome. *K-mers* of a read are all the sub-strings of length k of that read. Each read contains $(n-k+1)$ k-mers, where n is the read length, and k is the k-mer length. The *de Bruijn* graph is a directed graph which is used to find overlaps between reads. To do so, first all the k-mers of all the reads are generated. Then, the nodes are connected if they share an overlap of a $(k-1)$-mer. There are two types of *de Bruijn* graphs, Hamiltonian and Eulerian. In the Hamiltonian *de Bruijn* graph, k-mers are nodes, and edges are the connections between the nodes having an overlap of $(k-1)$-mer. While, in the Eulerian *de Bruijn* graph, k-mers are edges and the overlapped $(k-1)$-mers are the nodes [40]. An example of Hamiltonian and Eulerian de Bruijn graphs is shown in Figure 2.3.

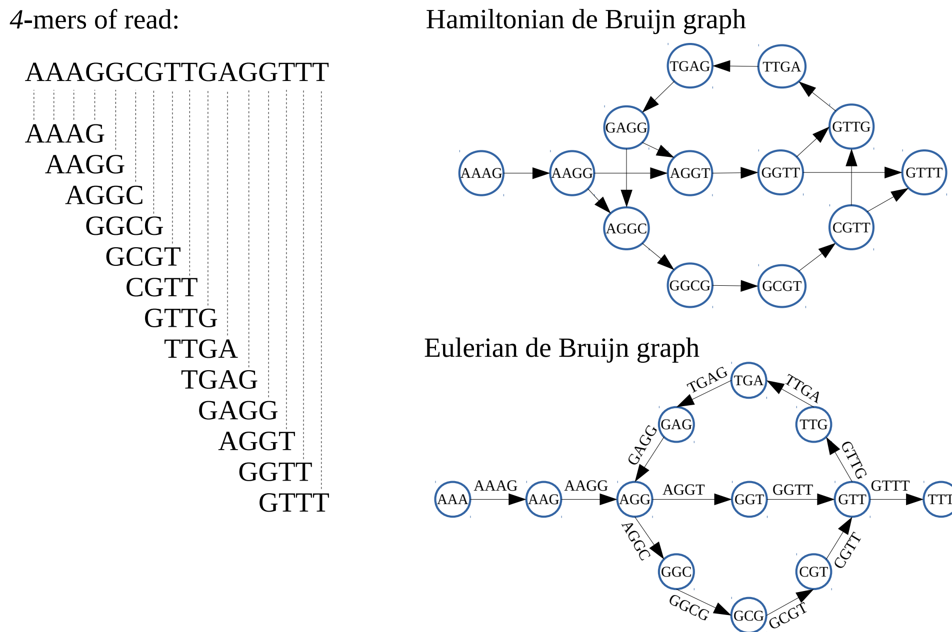


Figure 2.3: An example of a Hamiltonian and Eulerian *de Bruijn* graphs [40].

Once the graph is formed, the optimal path(s) which reassembles the genome is identified. Wisely choosing the length of the k-mer (k) is important in the formation of *de Bruijn* graph

and the results it yields. A very large k removes some short repetitive regions and hence, leaves more unconnected sub-graphs, which leads to having more gap regions. In contrast, a very small k reduces some gap regions and hence, increases the number of nodes and edges, which leads to increasing the number of short repetitive regions [33]. The *de Bruijn* graph data structure is widely used in next generation sequencing [41]. Popular DBG-based assembly algorithms include Euler assembler [42], Velvet [43], Euler-USR [44], AllPaths [45], ABySS [46], ABySS2 [47] and IDBA [48].

Some of the OLC-based genome assemblers and mainly DBG genome assemblers use k-mer counting in their algorithms. The k-mer counting problem consists of building a histogram of occurrences of the k-mers of all reads in the dataset. K-mer counting plays a crucial role in genome analysis, allowing for the detection of overlaps and estimation of genome size [49]. It is also valuable for other bioinformatics applications, such as fast multiple sequence alignment and repeat detection [50], and for creating multiple protein sequence alignments (for fast distance estimation) [51]. K-mer counting is a simple yet time-consuming phase of many genomics applications. One of the main challenges of k-mer counting is the huge amount of memory it requires, specially when dealing with large datasets like a complete human genome.

Most of the k-mer counters in the literature use either hash tables or sorting mechanisms for their counting algorithm. Jellyfish [52], BFCounter [53], DSK [54], KMC [50] and MSPKmerCounter [55] are among the first memory efficient k-mer counting algorithms. Jellyfish uses an algorithm using an efficient lock-free hash table. Jellyfish writes hash tables to disk when the system is out of memory and merges them later. BFCounter proposes using a Bloom filter to discard unique k-mers before putting them into the hash table. In a dataset, a huge number of k-mers are unique and generated mostly due to sequencing errors. Hence, by accepting a low rate of false positives, Bloom filters can be used to reduce memory usage significantly. The concept of partitioning is introduced in DSK, KMC and MSPKmerCounter, which all are disk based algorithms. DSK generates all the k-mers of the whole input in a loop and, based on the loop iteration, only some k-mers are stored on disk. K-mers are partitioned based on a hash function and written to their specific files. The total number of loop iterations is determined by the size of the input and the specified disk and memory space. Later, each file (partition) is loaded in memory separately and processed using a hash table. KMC uses a sorting mechanism instead of hash tables and compacts data before writing it to disk. In this approach, all the k-mers are generated and partitioned based on k-mer prefixes with variable lengths and are put into specific bins (partitions). When a bin is full, its data is compacted and written to a file.determining the relationship between each gene and its corresponding

instruction, as well as Later these files are read back to memory, sorted, and the frequency of each k-mer is counted. KMC uses multiple threads for k-mer generation, compaction, writing data to disk, and sorting, and the communication channel between each pipeline stage uses shared queues, which often become a congested synchronization point. MSPKmerCounter uses super k-mers generated using minimizers, which reduces the required disk space. The file (partition) of each super k-mer is determined based on its minimizer. Later, each time a partition is loaded in memory, k-mers are extracted from super k-mers and are processed using hash tables, and the resulting hash table is sorted and written back to disk.

The aforementioned algorithms use different approaches to reduce their memory footprint. Some works in the literature have mixed these ideas to improve performance of k-mer counting. Turtle [56] generates k-mers and partitions them using hash tables. Then, a consumer thread responsible for a partition creates one Bloom filter for that partition and, when the number of k-mers reaches a threshold, the thread sorts the k-mers and calculates their frequency. Thus, at the end of each frequency calculation, the table only contains distinct k-mers with their associated frequency. The next time that the number of k-mers reaches the threshold, in addition to sorting them, their histogram is calculated by comparing the new k-mers with the ones from the previous step. KMC2 [57] is the modified version of KCM in which partitioning is done using minimizers and, instead of saving k-mers, it stores super k-mers on disk, like MSPKmerCounter.

In this thesis we accelerate the k-mer counting step of the SMUFIN [18] application. SMUFIN is a reference free algorithm which identifies somatic mutations just by comparing normal and tumor DNA samples of the same patient. This algorithm is explained in detail in Section 4.

Reference guide assembly

The first step in most DNA sequence analysis pipelines based on a reference genome is to determine the location of each read in the reference genome. Modern read mappers such as BLAST [58], BWA-MEM [59], Minimap2 [60], and GEM [61, 62] avoid comparing every read against the whole reference genome, which can be very time-consuming and inefficient. Instead, modern mapping tools implement a seed-and-extend approach. The seeding step consist in finding candidate locations in the genome where a read can map to. Doing this, each read is only compared with a few regions from the genome, which depict some similarities with the input read. Afterwards, during the seed extension step, also known as *pairwise read alignment*, each candidate location is aligned against the input read. This process determines the degree

of the similarity (i.e., matches, mismatches, insertions, and deletions) between the input read and the candidate region of the reference genome. The main focus of the current thesis is to accelerate the pairwise read alignment algorithm leveraging FPGA and ASIC accelerators. This way, we discuss this topic in more detail in the following sections.

Pairwise Read Alignment

An essential problem in computational biology is the comparison and alignment of two DNA sequences. Pairwise sequence alignment identifies similarities between the elements of a pair of sequences which may share a common characteristic, revealing mutations, insertions or deletions of bases and demonstrating the functional or structural relationships between two DNA sequences. Sequence aligners are used to find the homology between DNA sequences and to align sequences in a way that the similarity is maximized according to a given distance/score function. There are three main types of pairwise sequence alignment: global alignment, semi-global alignment, and local alignment.

Global alignment compares both sequences through their entire length, end to end. This way, deletions and insertions at beginning or at the end of both sequences are also penalized during the alignment. Global alignment is used when it is known that both sequences should be the same entirely, and it is done to find out the differences caused by sequencing errors or single nucleotide polymorphisms (SNPs) [63]. This type of alignment is used in the most common alignment algorithms and it is considered to be the most natural form of pairwise sequence alignment [22].

Semi-global alignment is a variant of global alignment in which the gaps at the beginning or at the end of one of the sequences are not penalized. Semi-global alignment is used when one of the sequences is much larger than the other and it is known that the entire length of the smaller sequence is related to the region of the larger sequence where they overlap. Hence, the entire smaller sequence, end to end, is aligned to a local region of the larger sequence. In semi-global alignment, gaps at both ends of the larger sequences are not considered in the alignment [22].

In contrast, local alignment does not compare none of the sequences end to end. Instead, it detects a sub-string in sequence a which has the most similarity with a sub-string in sequence b. In other words, it finds the most similar regions in the sequences. This way, the gaps before and after the similar regions of both sequences are not penalized. Local alignment is used in applications such as detecting homology, finding protein structure and function, deciphering evolutionary relationships, etc [64].

The most efficient way of performing pairwise sequence alignment is using dynamic programming techniques. The following sections explain these techniques for global and local alignment.

Dynamic Programming for Global Alignment

Levenshtein introduced the edit distance (Levenshtein distance) [65] between two strings as the minimum number of required operations, i.e., substitutions, insertions and deletions, to change one string to the other one. It is important in bioinformatics as the similarity between two DNA sequences can be measured and presented by the edit distance between them.

The edit distance between two sequences can be calculated using dynamic programming techniques. In this method we need to calculate a matrix (DP-matrix), in which one of dimensions corresponds to sequence *a* (or query, usually in the vertical dimension) and the other dimension corresponds to sequence *b* (or reference, usually in the horizontal dimension). In the DP-matrix we add an empty character at the beginning of each sequence. Figure 2.4 shows an example of calculating the edit distance using a DP-matrix, which explanation follows.

The value of each cell of the matrix determines the edit distance between the prefix of sequences until the coordinates of that cell. For example, the value in cell *(i,j)* shows the edit distance between prefix [0:i] of sequence *a* and prefix [0:j] of sequence *b*. Hence, the most right and the most bottom cell of the matrix determines the final edit distance between the sequences. The edit distance of each cell is calculated by Equation 2.1.

$$ed(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0 \\ min \begin{cases} ed(i-1,j-1) + \begin{cases} 0 & \text{if } a_i = b_j \\ 1 & \text{Otherwise} \end{cases} \\ ed(i-1,j)+1 \\ ed(i,j-1)+1 \end{cases} & \text{Otherwise} \end{cases} \tag{2.1}$$

The first case of the equation initializes the cells of the first row and the first column of the matrix from 0 to *L* in ascending order. This is because the edit distance between an empty character and a sequence of length *L* is *L*. The second case of the equation determines how the value of the rest of cells is calculated. This value is achieved by comparing three previously calculated adjacent cells (top, left and diagonal). According to the dependencies between cells,

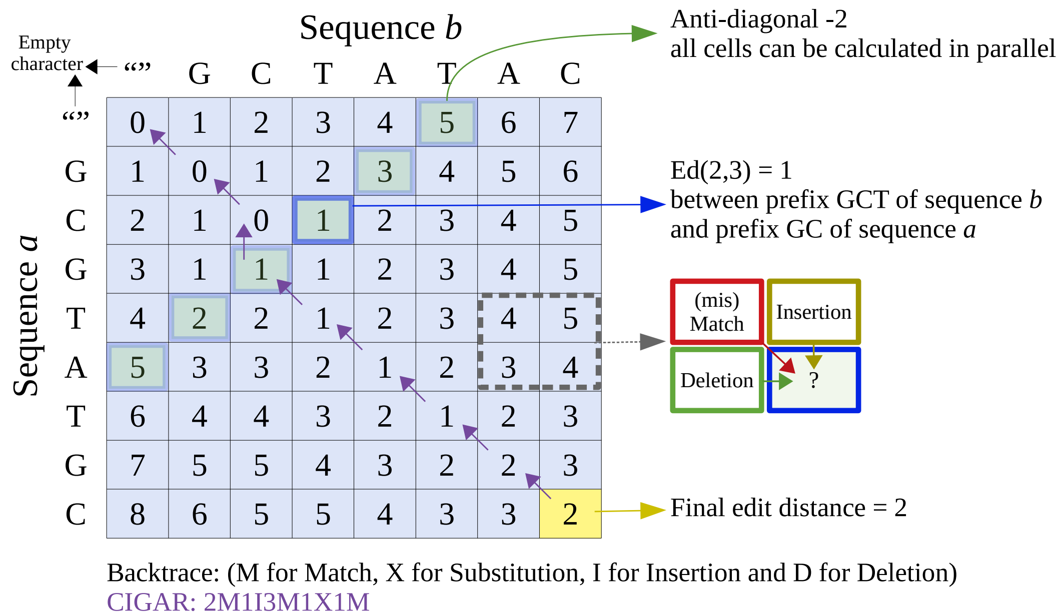


Figure 2.4: Calculating DP-matrix of edit distance between sequences *a* and *b*.

and by looking at the matrix in Figure 2.4, we can see that the algorithm can be parallelized. In particular, all the cells in each anti-diagonal can be calculated in parallel.

As mentioned earlier, the last cell of the matrix holds the edit distance between the two sequences. However, in order to identify how different the sequences are, we need to perform the backtrace step. To do the backtrace, when we calculate the value of each cell, we also save the direction (top, left or diagonal) from which the value is achieved. Using this data we trace back the directions from the last to the first cell, i.e., from $ed(8,7)$ to $ed(0,0)$ in the example of Figure 2.4. The backtrace provides the Compact Idiosyncratic Gapped Alignment Report (CIGAR) which is the set of operations that illustrates all of the differences and similarities between sequence *a* and sequence *b*. CIGAR string has several operators, each preceded by a number indicating consecutive occurrences of that operator. Operator ’M’ indicates a match, ’X’ indicates a mismatch, ’I’ indicates an insertion and ’D’ indicates a deletion in sequence *a* (compared to the reference genome). A diagonal direction indicates a match or a mismatch. If $a(i) = b(j)$ it is a match, otherwise it is mismatch. A vertical direction indicates an insertion and a horizontal direction indicates a deletion in sequence *a*.

When calculating the edit distance between two sequences, each difference (mismatch, insertion and deletion) is penalized by a unit of 1. However, depending on the nature of the

a) Distance Scoring Matrix (Penalty Function)

	A	G	C	T	“ ”
A	0	x	x	x	g
G	x	0	x	x	g
C	x	x	0	x	g
T	x	x	x	0	g
“ ”	g	g	g	g	

b) Similarity Scoring Matrix (Score Function)

	A	G	C	T	“ ”
A	a	-x	-x	-x	-g
G	-x	a	-x	-x	-g
C	-x	-x	a	-x	-g
T	-x	-x	-x	a	-g
“ ”	-g	-g	-g	-g	

Figure 2.5: a) Distance scoring matrix; b) Similarity scoring matrix.

bioinformatics analyses, some substitutions might be more common than other substitutions or gaps (insertion and deletion). For this reason, more complex distance scoring or penalty functions are introduced. Figure 2.5 (a) shows a distance scoring matrix where a match is not penalized, a mismatch penalty is x and a gap penalty is g. To use this distance scoring matrix, the edit distance computation previously shown in Equation 2.1 evolves into Equation 2.2 [63]. Using distance scoring, the lesser the distance, the more similar the sequences are.

$$H(i,j) = \begin{cases} max(i,j) \times g & \text{if } min(i,j) = 0 \\ min \begin{cases} H(i-1,j-1) + \begin{cases} 0 & \text{if } a_i = b_j \\ x & \text{Otherwise} \end{cases} \\ H(i-1,j) + g \\ H(i,j-1) + g \end{cases} & \text{Otherwise} \end{cases} \tag{2.2}$$

A similar metric to distance scoring is similarity scoring. In similarity scoring, the bigger the similarity score (or simply the score), the more similar the sequences are. As shown in Figure 2.5 (b), in the similarity scoring method a match between characters is rewarded by a positive score a and differences are penalized by negative scores, i.e., -x for mismatches and -g for gaps. When using the similarity scoring method, the DP-matrix is computed using Equation 2.3. This algorithm is known as the Needleman–Wunsch algorithm [66].

$$H(i,j) = \begin{cases} max(i,j) \times (-g) & \text{if } min(i,j) = 0 \\ max \begin{cases} H(i-1,j-1) + \begin{cases} a & \text{if } a_i = b_j \\ (-x) & \text{Otherwise} \end{cases} \\ H(i-1,j) + (-g) \\ H(i,j-1) + (-g) \end{cases} & \text{Otherwise} \end{cases} \tag{2.3}$$

Neither of the scoring methods change the characteristics of the DP-matrix, i.e., anti-diagonal cells can be calculated in parallel, the final distance/score is kept in last cell of the matrix and the backtrace starts from last cell backwards to the first cell of the matrix.

Dynamic Programming for Local Alignment

The Smith-Waterman algorithm [67] is a variant of the Needleman–Wunsch algorithm to perform local alignment instead of global alignment. Local alignment is used when a local region of sequence *a* is related to a local region of sequence *b*, but they are not related end to end. Thus, in local alignment the dynamic programming technique is modified in a way that it allows the alignment to start and end at any position of the sequences without penalizing gaps at both ends of the sequences. To do that, the local alignment equation shown in Equation2.4 includes a term that sets the score to zero whenever a difference in the alignment makes a negative score. This equation uses the scoring matrix of Figure 2.5 (b) and initializes the first row and column of the DP-matrix to 0. This allows the alignment to start from any position of the sequences [63].

$$H(i,j) = \begin{cases} 0 & \text{if } min(i,j) = 0 \\ max \begin{cases} H(i-1,j-1) + \begin{cases} a & \text{if } a_i = b_j \\ (-x) & \text{Otherwise} \end{cases} \\ H(i-1,j) + (-g) \\ H(i,j-1) + (-g) \\ 0 \end{cases} & \text{Otherwise} \end{cases} \tag{2.4}$$

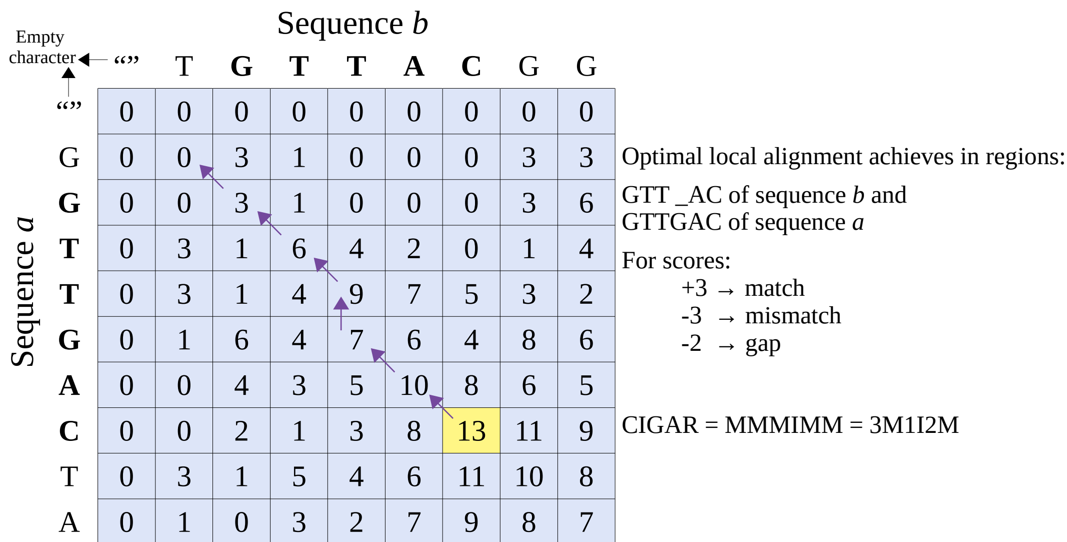


Figure 2.6: Calculating the DP-matrix of local alignment between sequences *a* and *b*.

Figure 2.6 illustrates an example of calculating the DP-matrix for local alignment with a sample score function. Please note that different score functions produce different alignment results. In the DP-matrix, cells with lower values show less relevant regions, while higher values indicate areas with higher similarities. The calculation of the DP-matrix of the local alignment follows the same steps as of the global alignment, the only difference is the starting and ending positions of the backtrace. The backtrace for the local alignment starts from the cell of the matrix which holds the maximum value, and traces back until a cell with a value of zero is reached. This way, not only we detect the most similar regions in the sequences, but we also identify the differences between these regions.

Both the Needleman–Wunsch algorithm for global alignment and the Smith-Waterman algorithm for local alignment have the same nature, and both implement the *gap-linear* scoring system. The gap-linear scoring system does not differentiate the penalty of a gap-opening from a gap-extension (continuous gap), that is, it penalizes a gap proportional to its length. However, in biological analyses, the penalty of a gap may not increase linearly with its length, and it is preferred that a gap-opening (first gap) is penalized more compared to the gap-extension(s) following the first gap [22]. To this end, the Smith-Waterman-Gotoh (SWG) algorithm [68] is a variant of the Smith-Waterman algorithm that implements the *gap-affine* scoring model. This scoring model penalizes more the gap-opening than the gap-extensions, so it is highly preferred by biologists as it provides a more realistic computation where a genetic mutation generally

causes an insertion/deletion of a large block of bases [69]. However, using this scoring system makes the alignment algorithm more complicated, and thus lowers the performance of the sequence aligners. Yet, modern sequence alignment tools tend to implement alignments based on the gap-affine scoring model.

In the gap-affine scoring model there are three DP-matrices to be solved, M, I and D, which track the scores of alignments ending with a match/mismatch, an insertion, and a deletion, respectively. Equation 2.5 indicates how each matrix is calculated.

$$Initialization \begin{cases} M(0,j) = M(i,0) = 0 \\ I(0,j) = D(i,0) = -\infty \\ I(i,0) = D(0,j) = NotUsed \end{cases}$$

$$M(i,j) = max \begin{cases} M(i-1,j-1) + \begin{cases} a & \text{if } a_i = b_j \\ (-x) & \text{Otherwise} \end{cases} \\ D(i,j) \\ I(i,j) \\ 0 \end{cases} \tag{2.5}$$

$$D(i,j) = max \begin{cases} M(i-1,j) + (-g_o - g_e) \\ D(i-1,j) + (-g_e) \end{cases}$$

$$I(i,j) = max \begin{cases} M(i,j-1) + (-g_o - g_e) \\ I(i,j-1) + (-g_e) \end{cases}$$

In the equation, a is the reward of a match, -x is the penalty of a mismatch, and $-g_o$ and $-g_e$ are the penalties of gap-opening and gap-extension, respectively. Please note that a first gap (opening) is penalized both for gap-opening and gap-extension, while a continuous gap (extension) is only penalized for gap-extension. Basically, a gap-opening is more costly than a gap-extension. In this method, as the values of the cells of each matrix are calculated based on the values of the cells of the other matrices, the backtrace also has to potentially traverse up to

three matrices. However, the concept remains the same as when using the gap-linear scoring model.

All the aforementioned algorithms use DP techniques to find the optimal alignment, whether local or global. A common characteristics of these algorithms is that they need to calculate all cells of the DP-matrix, which is time and memory consuming. To solve this problem, many heuristic variants of these algorithms have been proposed, in which a reduced number of cells of the DP-matrix are calculated. The main problem of the heuristics methods is that they do not provide the exact (or optimal) alignment. As an example, the pairwise read alignment step of BLAST limits the computation of cells to bases that are likely to be highly similar. Many other heuristic methods only compute a band of the matrix, known as banded Smith-Waterman, or adaptively banded Smith-Waterman. In these kind of algorithms, the computation of the matrix is limited to a fixed-size W wide band of diagonal cells, i.e., W/2 of diagonals after, and that of diagonals before the main diagonal. This is done based on the assumption that the alignment of a high quality sequenced read should not go far from the main diagonal. These heuristic methods, in general, are more desired to be implemented in hardware, specially if the target is to align long sequences. This is because such methods require much less memory compared to the optimal methods.

Recently, the WFA algorithm [19] has been proposed. The WFA algorithm uses a novel approach which only computes a reduced number of the DP-matrix cells. However, it is not a heuristic method and finds the optimal alignment. Three of the contributions of this thesis are about the acceleration of the WFA algorithm, which are discussed in Sections 5 and 6.

2.2 Hardware Accelerators

Nowadays, a variety of software-based applications, ranging from trading to medicine and aerospace, take advantage of accelerators such as GPUs, PIM, FPGAs, and ASICs to primarily increase the speedup and, in some cases, consume less power. These accelerators can increase the parallelism of the application at the cost of decreasing the flexibility of it. Therefore, usually it is more desirable to use them beside the CPU implementation, and constitute a co-design in order to provide both the flexibility and the degree of parallelism which is required by the application.

2.2.1 GPU

GPUs were initially designed in the 1970s and 1980s to offload the CPU task of 2D graphics rendering computations. Later in the 1990s the support of 3D rendering was added to the GPUs [70]. In that time GPUs were not programmable and the rendering operation was implemented in the GPU hardware. From these early designs, the architecture of the GPUs has developed over the time. First they offered limited programmability only for operating on graphical data (e.g., pixels and vertices) and then, in 2009, the release of Fermi architecture General Purpose GPU (GPGPU) allowed to solve non-graphics-related problems [71]. By adding more and more programmability capabilities and flexibility to the GPUs, programming models and environments were developed allowing to run non-graphical application written in C-like languages on the GPUs. Compute Unified Device Architecture (CUDA) and OpenCL are the most popular programming models for GPUs. CUDA is developed by NVIDIA only for NVIDIA GPUs, while OpenCL can be used generally for different vendors (e.g., Intel, AMD, NVIDIA, Altera, IBM, Samsung, Xilinx) [72].

In comparison with CPUs which can be used for any kind of applications running on several processing cores and software threads, GPUs are massively parallel devices used in highly parallelized computational applications as they include tens of processing cores and can run thousands of threads. A GPU has multiple processing units called streaming multi-processors (SM). The SMs are responsible for executing the GPU functions known as kernels. Each SM is capable of running multiple threads, known as a thread block, and executing hundreds of instructions in parallel. GPUs have a Single Instruction Multiple Threads (SIMT) architecture. The execution unit in SIMT is called warp which typically consist of 32 threads. In other words, the thread block of the SM is divided into batches of 32 threads, called warps. The A100 GPU has 108 SMs, in which each SM is able to run 64 warps. Having 32 threads per warp, the A100 when running with full capacity could have 221,184 threads (108 SMs $\times$ 64 warps per SM $\times$ 32 threads per warp) in flight [73].

GPUs have different types of memories. Some of them are shared and some other are private to each SM or thread. The memory hierarchy of a GPU is shown in Figure 2.7. Each SM includes registers (of 256KB in A100) which are private to each thread and are not visible to other threads. The registers resource utilization depends on the compiler decisions. In addition to registers, each SM has a fast, on-chip scratchpad memory (of 192KB in A100) that can be used as L1 cache and shared memory. All threads in a SM can share shared memory. Also all threads of a SM can share the physical memory resource provided by the SM. Each

SM contains a read-only memory to kernel code which includes instruction cache, constant memory, texture memory and RO cache. The global memory which is of several GB (40GB in A100) and the L2 cache which is of several MB (40MB in A100) are shared memories among SMs. The global memory contains the frame buffer of the GPU and includes the DRAM of the GPU [74].

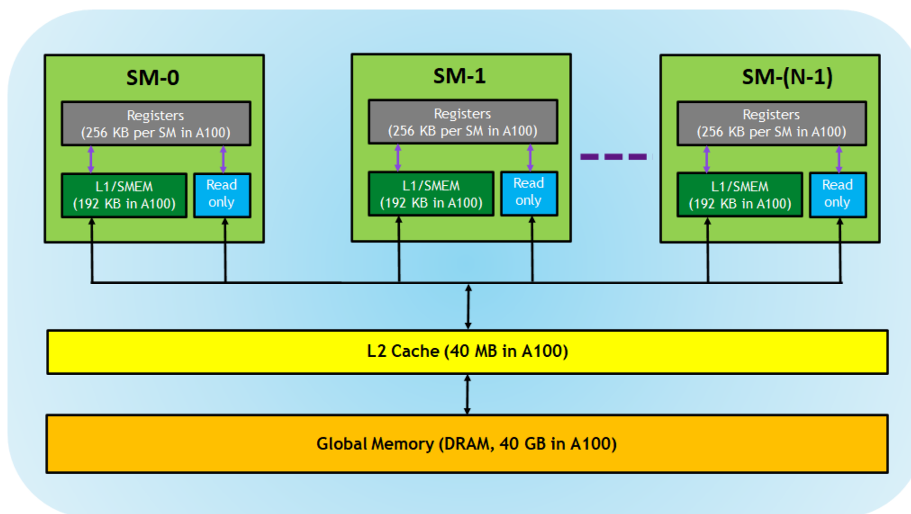


Figure 2.7: Memory hierarchy in GPUs [74].

The architecture of GPUs continuously improves in order to make them faster or more efficient for specific applications. For example very recently NVIDIA has introduced their new Hopper GPU architecture which is able to accelerate dynamic programming techniques using an instruction set built into NVIDIA H100 GPUs, called DPX instructions [75]. New DPX instructions in H100 GPU are claimed to improve the performance of DP algorithms by 7× compared to the A100 GPU, and by 40× compared to traditional dual-socket CPU-only servers [76]. Accelerating algorithms based on dynamic programming are of importance as they are used in different scientific areas including genomics.

GPU applications, in order to obtain high performance executions, should design kernels with thousands of threads all doing the same instruction at the same time. To increase the level of parallelism, GPUs use a latency-hiding mechanism in which several warps of the SM thread block are dynamically scheduled to be executed in the SM. GPUs achieve a very high degree of parallelism specially from instruction level parallelism, as they can execute multiple independent instructions at the same time. Consequently, they should be used for applications capable of being parallelized, otherwise the performance drops significantly, even less than that of on a CPU. For example, applications which need a huge number of vector operations are good

targets for GPU implementations. They can reach performance improvements of several orders of magnitude compared to a CPU implementation [77, 78]. On the other hand, applications with many conditional branches degrade the level of parallelism due to the instruction dependencies they make and warp divergence they cause. This problem makes some of the threads idle, and only a few remain active. The other GPUs drawback is the slow access to the global memory in comparison with the fast access to the on-chip memory (i.e., shared memory and registers). Thus, the access to the global memory should be limited by performing coalesced accesses in order to increase the performance. This means that the shared memories should be exploited. However, the amount of required shared memory by CUDA block negatively affects the number of concurrent CUDA blocks running on the same SM (i.e., reduces the total number of concurrent active threads per SM). Having high thread occupancy is critical for hiding memory latency and compute operations [78].

2.2.2 PIM

In data intensive applications, such as genomics applications, a huge amount of data should be regularly read from the memory, processed and written back to the memory. These data transfers between memory and processor cause latency and power consumption.

The idea of processing in memory (sometimes referred as processor in memory) is to do some or all the operations on data in memory (e.g., in RAM) by embedding logic units or by integrating the whole processor inside the memory or near memory in a single chip. The latter one sometimes is called near-memory processing [79, 80]. PIM makes it feasible to perform the necessary computations and data processing inside the memory of a computer or a server [81]. Hence, all or lots of data accesses to the memory are eliminated, and so the performance is not degraded due to the off-chip data transfer latency [82]. It also helps in saving energy. In addition, by processing in memory we can benefit from the inherent parallel computing mechanism and utilize wide internal memory bandwidth [80]. Consequently, it increases the speed up of processing tasks, and so the whole application by processing data inside the memory. The application software running on one or more CPUs manages the processing and the data in memory [81].

Processing in memory is of interest nowadays not only due to its performance improvements compared to data processing methods requiring multiple data readings and writings to the memory but also due to the performance improvements over readings and writings from/to slower media. Reads and writes to a slower media usually results in a data access bottleneck, known as Input/Output (I/O) bound. Processing in memory is considered to be real-time or

at least is used in real-time application. As RAM prices continuously decrease, processing in memory opportunities are becoming more economical [82].

Not only data intensive applications but also today's standard von Neumann computer architectures have a latency problem. This is because in such architecture there are two separate computing and memory sections which are connected through buses [81]. This leads to memory access latency, limited memory bandwidth, energy-hungry data transfer, and huge leakage power for holding data in volatile memory [83, 84]. This is because processing data is only the task of processor and memory is not taking a part in that. Hence, a huge number of instruction fetch and data transfer is needed between both sections [81]. Processing in memory is a promising technique to tackle the above mentioned issues, especially the latency. Figure 2.8 compares the von Neumann architecture in CPU and GPU with PIM architecture.

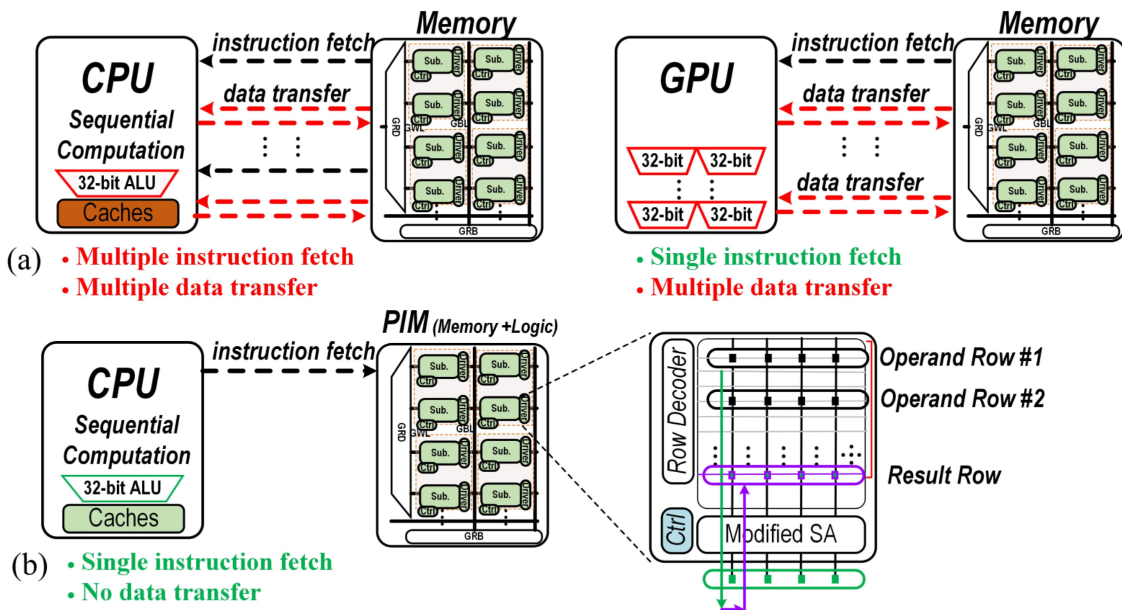


Figure 2.8: (a) The von Neumann architecture in CPU and GPU, (b) PIM architecture [80].

The PIM architecture, preferably should have the ability to perform bulk bit-wise operations as they are required in data intensive applications [85, 86]. Normally a PIM has operand rows ready to operate on the data of rows of the memory. When PIM's row decoder receives the instruction from the CPU, it detects the rows of the two operands and performs bit-wise logic function (required computations) between bit-cells of the target rows which are storing the two operands (operand row 1 and 2 are activated in Figure 2.8(b)) [80]. In the literature it is doable by performing different modifications on memory components at Sense Amplifiers (SA) level [85], on memory bit-cell level [87, 88], or even adding combinational circuits after SA [89–91].

There are works suggesting PIM architectures for SRAMs [92, 93], DRAMs [94, 95] and even Non-Volatile Memory (NVM) technology, such as Phase Change Memory (PCM) [96] and resistive RAM (ReRAM) [83]. Processing in DRAMs has been a trend in recent years mainly because of huge memory capacities and off-chip data transfer reduction opportunities it delivers. Although the current processing in memory architectures designed to perform a range of tasks from simple logical operations [85, 97] to more complex ones [80, 98–100], they suffer from serious drawbacks such as high refresh/leakage power, multi-cycle logic operations, operand data overwritten, operand locality, etc.

2.2.3 FPGA

FPGAs, which are the main focus of this research, are integrated circuits which allow users to configure them over and over and design digital circuits. Simply, an FPGA is composed of an array of Configurable Logic Blocks (CLBs). Designers can design their digital logic using low level Hardware Description Language (HDL) or high level OpenCL programming language. The two most popular HDLs are VHDL and Verilog. Programming using HDLs gives more control to the designer in terms of implementation, resource utilization and optimization.

FPGAs support lower frequencies compared to the CPUs and GPUs. With current technologies an FPGA design barely reaches a frequency above 300MHz and usually less than 200MHz. In contrast, CPUs easily work with frequencies of orders of gigahertz. This makes FPGAs more energy efficient but not slower and even faster if they are used wisely for the appropriate algorithms. FPGAs, unlike the CPU instructions which may need multiple clock cycles to be executed, can execute multiple operations in parallel and in just one clock cycle or two. This is because with FPGAs we implement the operations with physical resources. In other words, we configure the available FPGA resources to implement the desired circuit which performs the required functionality. Therefore, if in an algorithm, multiple data can be analyzed independently, they could be easily parallelized in the FPGA.

Depending on the FPGA manufacturer, the FPGA resources may differ or be handled differently. Normally they consist of Look-Up Tables (LUTs), Flip-Flops (FFs), DSPs, memory blocks and programmable I/O blocks. LUTs are truth tables inside the FPGA, which can have different number of inputs depending on the FPGA and one output. Essentially, this element is a truth table in which different combinations of the inputs implement different functions to yield output values. LUTs can act as logic gates in various combinations. LUTs also can be treated as a small piece of RAM. The address of the RAMs are as the inputs of the LUTs. LUTs are building blocks of combinational circuits. FFs are the smallest memory elements

which keeps the state of a signal for one clock cycle. FFs are the building blocks of sequential circuits. DSP block is an Arithmetic Logic Unit (ALU) and includes dedicated multiplication resources and also some other supporting operations. On-chip memory blocks of FPGAs are usually in orders of tens of megabits. They can be used as dual or single port RAMs, ROMs and as FIFOs. The limitation in using blocks of memories is that the minimum size of the memory you can use is the minimum size of a physical block (i.e. 18Kb for Xilinx virtex ultra scale plus FPGAs). In other words, if you use less than 18Kb of a block of memory, it allocates the entire block of 18Kb. The general I/Os could be configured as inputs, outputs or inouts. Some special I/Os are for high speed communications, differential signals or clocks of the system. I/O groups also can be configured to work with different voltage levels. It makes it easy to connect FPGAs to peripherals of different voltage levels. A basic structure of an FPGA is shown in Figure 2.9.

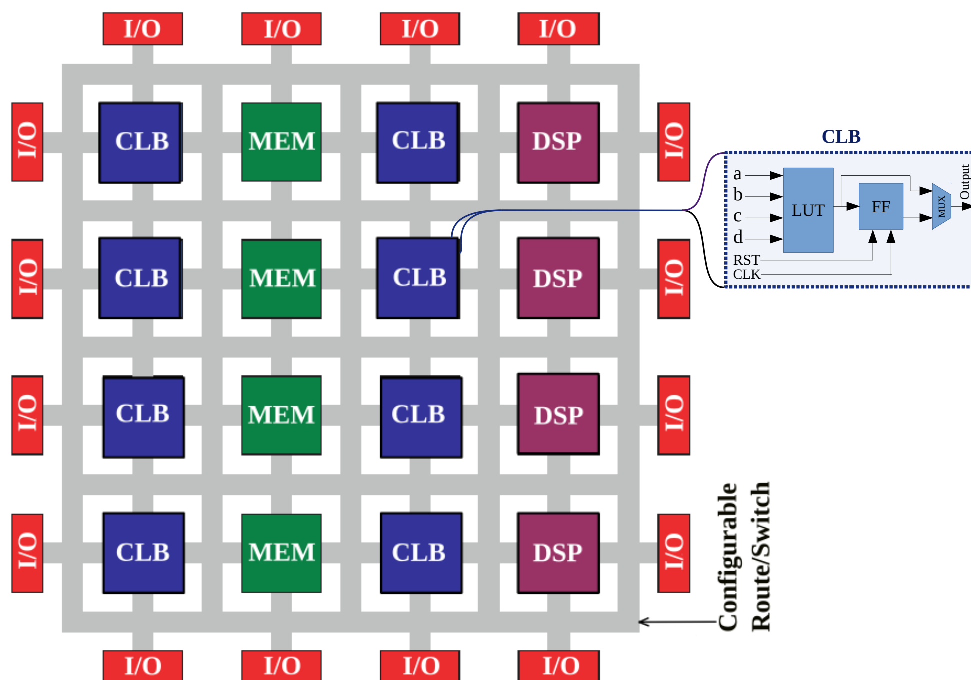


Figure 2.9: FPGA structure. Derived from [101].

The FPGA design flow is shown in Figure2.10. First, we need to evaluate the algorithm we want to implement in the FPGA, define the design functionality and estimate the performance regarding the FPGA resources and the frequency expectation. To have a modular design we need to separate different functions of the design and implement them in different modules. Then the functionality of each module is described in HDL as the source codes of the design. Next, a behavioral simulation is performed to make sure the design functions as expected. In

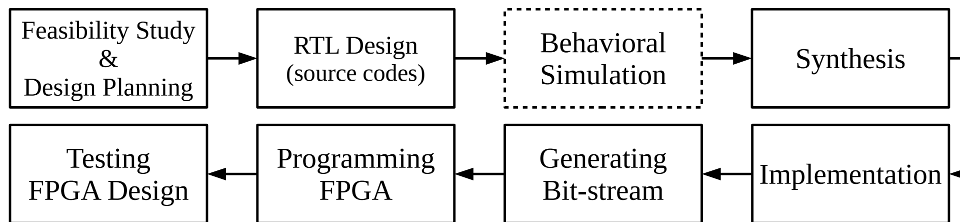


Figure 2.10: FPGA design flow.

this step the frequency and timings are not considered and only the functionality of the design is of importance. Some of the HDL simulator tools are ModelSim/QuestaSim from Siemens and Xcelium from Cadence. Some of the EDA tools also have embedded simulators. After verifying the design functionality, using an EDA tool, a project is build and source codes are added to that. The most common tool for targeting Xilinx FPGAs is Vivado and for targeting Intel FPGAs is Quartus.

After adding source files and constraint files to the project, the tool performs the synthesis step by transforming the RTL code into a lower level of gate-level description. Usually after this step an optimization step is performed in order to reduce resource utilization, power consumption and critical path (a trade-off between them). Then the logic gates are mapped to the existing technology libraries and the output is a netlist of the synthesized modules which is understandable for the next step. Synthesis tool also reports a fine estimation of resource utilization, power, timing and etc. Regarding the resource utilization report we can re-think the design and increase or decrease the level of parallelization if the resource utilization is low or high, respectively. At this point also we should check the timing and maximum frequency the design can reach. If it does not reach the expected frequency, whether the RTL code should be modified and the synthesis step runs again, or we should lower our frequency expectation if possible.

The next step after synthesis is implementation. Translation, mapping, placing and routing is commonly referred to as the implementation step [102]. In this step the netlist is divided into sub-blocks and is mapped and placed to actual FPGA blocks. Then it routes between the blocks. The placement and routing is done by considering time constraints. Optimizations may perform after placement and after routing to decrease resources and to decrease critical path delay which improves the frequency. Implementation reports consist of final resource utilization, power and timing reports. We should check again the resource utilization and timing reports to make sure everything is as expected.

The final step is generating the bitstream. In this step the implemented design is converted into a bitstream understandable for the FPGA. A bitstream includes the description of the hardware logic, routing, and initial values for both registers and on-chip memory (e.g., LUTs). By streaming it to the FPGA configuration port, we can program the FPGA and test the FPGA design.

There are several technologies to transfer the configuration bitstream to the FPGA. The most widely-used one is the SRAM-based FPGAs technology. In this method, FPGA keeps the configuration data in the static memory. In this method the FPGA should be programmed on the start and every time that FPGA is powered on, as SRAM is volatile. This method has two modes of master and slave. In master mode FPGA reads configuration data from an external flash memory. While in slave mode, FPGA is configured by an external master device, such as a processor. This is usually done using a JTAG interface [103]. SRAM cells are distributed throughout the design in form of an array and mainly are used to program: (1) the routing interconnects (2) used CLBs (see Figure 2.9) [104]. SRAM cell could be programmed an indefinite number of times.

2.2.4 ASIC

ASICs are hardware circuits which are only designed to have a specific application, and unlike FPGAs are not programmable. In fact FPGAs are used as ASIC prototypes. In the design of an ASIC the HDLs are used. The design of an ASIC should be tested on FPGAs, because after ASIC fabrication it cannot be changed. ASICs can reach frequencies above 1GHz which is much faster than FPGAs.

Multiple ASICs which each of them has a dedicated function can be connected together on a single ASIC chip and create a System-on-Chip (SoC). Typical SoCs may include a microcontroller or microprocessor core, DSPs, memories such as RAMs or ROMs, peripherals such as timers/counters, communication interfaces such as USB/UART, analog interfaces such as ADCs/DACs or other hardware [105]. Communication between SoC modules is a challenge in SoC design. However, it may be facilitated by a simple bus if the number of modules is small or they do not need to communicate together. A commonly used bus in SoC designs is the AXI (Advanced eXtensible Interface) [106, 107] interface introduced by ARM.

RISC-V based SoCs

Since the emerge of the open source Instruction Set Architecture (ISA) of RISC-V many research centers and companies started developing their own custom RISC-V based processors for their SoCs without the need of paying for that. This is the main reason of RISC-V popularity, i.e. an open source publicly available ISA standard [108] which could be used and developed by everyone.

In 2010 the research project of RISC-V started in UC Berkeley, and soon after, in 2011, the first RISC-V chip in 28nm FDSOI taped out [109]. In the same year, the first report describing the RISC-V instruction set was published [110]. The first RISC-V workshop was held in 2015, and later that year, the RISC-V Foundation established by 36 founding members. These company members include NVIDIA, Google, IBM, and Qualcomm. In 2020, the RISC-V Foundation became RISC-V International [111].

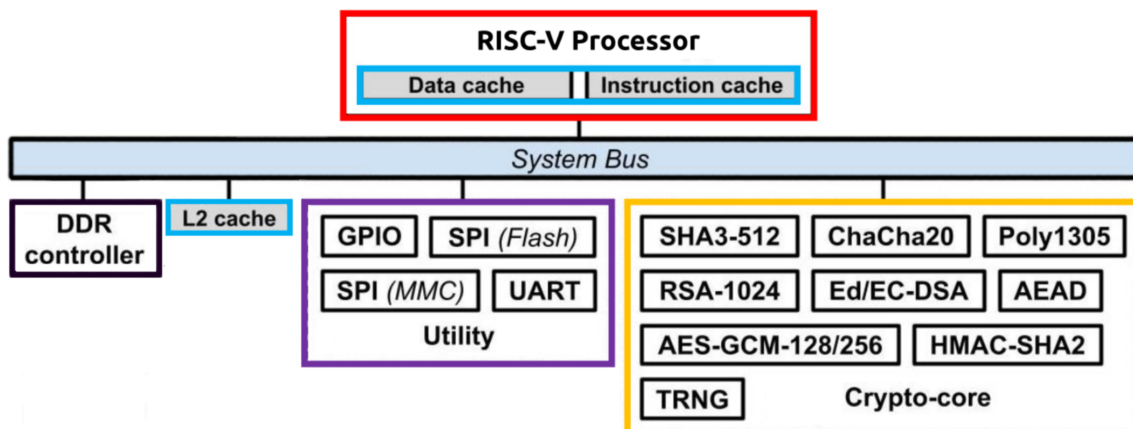


Figure 2.11: RISC-V based SoC with cryptography accelerators [112].

A sample RISC-V based SoC with a custom IP (ASIC) as cryptographic accelerator is shown in Figure 2.11. In the literature several works have been implemented application-specifies IPs in the RISC-V based SoCs to accelerate different applications such as cryptography [112], deep neural network [113] and, Internet-of-Things (IoT) [114].

Adding ASIC accelerators to the SoC is not the only solution for accelerating specific applications when there is a RISC-V CPU in the chip. RISC-V ISA has different groups of instructions (ISA extensions), and is designed to be able to activate or deactivate some of these groups of instruction extensions based on the system requirements. This is an efficient practice to avoid area and power increases by only keeping the necessary elements in the design. Among these groups one empty group exists which belongs to the designer customized instructions. This enables designers to invent their required application-specific instructions complying with

the RISC-V standards without violating the main specification. Hence, based on the designer needs, it is possible to have a RISC-V processor with minimum instructions for basic integer operations (I/E extension), with all extensions enabled, or even with designer own-specific extensions [115]. Many projects in the literature have designed instruction extensions to accelerate their target applications in cryptography [116], and driving assistance [117].

However, adding customized extensions is not a trivial implementation. It not only is about implementing the instructions in RTL code, but also is about the work to add them to the compiler, simulators, debuggers and etc, and verify the functionality of all of those tools after the changes. Not to mention the RTL verification step. There are many companies providing RISC-V related services. For example Codasip provides an EDA tool that enables customizing off-the-shelf processors that Codasip also offers [115].

2.2.5 Genomics Hardware Accelerators

In this thesis we are targeting two genomics applications, k-mer counting and pairwise read alignment. Both are critical applications which have been targeted for acceleration in many works. In the following sections we review some of them.

K-mer Counting Accelerators

Gerbil [118] is one of the most well-known works on k-mer counting. It introduces an open source k-mer counter software with optional GPU threads The work evaluates different algorithms for different datasets and k-mer lengths and shows that its GPU enabled design outperforms other algorithms. Li [119], another GPU accelerator for k-mer counting, reports a significant speed up by porting some parts of KCM2 to GPU. Cadenelli [120] proposes some modifications to the k-mer counting application to decrease its memory requirements and proposes a GPU accelerator for it.

K-mer counting is a memory-bound application. Hence, using in/near memory processing could be a suitable acceleration method for this application. NEST [121] is a Near Data Processing (NDP) accelerator for k-mer counting. It is built by modifying the Load-Reduced Dual-Inline Memory Module (LRDIMM). NEST adds a Near Memory Computing (NMC) module to each rank within each LRDIMM to perform k-mer counting. In another work, Joardar [122] propose a Network on Chip (NoC) architecture that uses the Monolithic 3D (M3D) integration to implement multiple logic layers in PIM architectures with 3D-stacked memory for k-mer counting.

In another work of Cadenelli [123], he re-designs the GPU part of his previous work [120] for an FPGA implementation and makes an FPGA accelerator for k-mer counting. Mcvicar [124] ports Bloom filter generation and calculation of k-mer counting to a cloud of 4 FPGAs attached to a Hybrid Memory Cube (HMC). The paper shows how taking the advantages of parallel FPGA designs together with high random access speed of HMCs can yield performance improvements.

Pairwise Read Alignment Accelerators

In recent years, many hardware accelerators have been proposed to improve the performance of read mappers. GPUs can be leveraged to accelerate read mappers. Pham [125], Sadasivan [126], Rani [127] and Muller [128] has used GPUs to improve, BWA-MEM, minimap2, BLAST and Anyseq [129], respectively. Other works has focused on accelerating pairwise read alignment algorithm of the read mapper, the WFA algorithm [78, 130, 131] ,the SW algorithm [132–135], or both the SW and the NW algorithms [136, 137]. In-memory computation is also a promising technique to accelerate read alignment. Many works have suggested to improve the performance of the read mappers [138–142] or the pairwise read alignment part either using the SW algorithm [143, 144], the NW algorithm [145], the WFA algorithm [146], or all of them [147].

Many FPGA ASIC-based methods have been proposed to accelerate the SW algorithm. As a result, some surveys [14, 148, 149] have been published summarizing the many contributions done over the years. In addition, many production-ready bioinformatics tools already incorporate custom FPGA accelerators [150–156].

Some of the earlier works on sequence alignment tried to fully optimize the FPGA LUTs by doing custom optimizations using simple cost models, like the edit-distance, the Levenshtein distance [157–159] or being limited to compute the longest common subsequence (LCS) [160]. Despite their good performance, these solutions do not fulfill the requirements of modern bioinformatics tools due to algorithmic limitations.

Other FPGA-based proposals tackled the problem of accelerating the SW algorithm using linear gap penalties [156, 161–171]. The design of these accelerators has improved from optimized designs based on systolic architectures [156, 162, 166, 169] and custom FPGA designs to large-scale accelerators running on supercomputing infrastructures [167, 168]. Nevertheless, these solutions lack the flexibility to meet the requirements of many biological applications. For this reason, FPGA accelerators that fully implement the gap-affine model (i.e., the SWG algorithm) are usually preferred [154, 155, 172–187].

Other works propose to accelerate SWG approximate methods such as banded SWG. These heuristic methods are usually employed to align long reads [173–175, 188], although some works also apply them to short reads [155, 189]. The main limitation of these approaches is that the result of the algorithms is not guaranteed to be the optimal one, so they trade speed for accuracy.

Another problem of the gap-affine model is the complexity of producing the full alignment. For this reason, some accelerators proposed in the literature are limited to compute the alignment score [177, 178], but not the CIGAR. In contrast, other designs offer more flexibility and allow the computation of the full CIGAR alignment at the expense of lower performance [179, 186, 190].

Chapter 3

Experimental Methodology

This chapter describes the experimental methodology used in this thesis. The platforms used for thesis contribution are explained in Section 3.1, while Section 3.2 outlines the baselines and input sets of each contribution.

3.1 Platforms

The platform used for the first three contributions of this thesis is the IBM POWER9 system with attached FPGAs which is explained in Section 3.1.1. The last contribution of this thesis, is an ASIC design in a RISC-V SoC. In Section 3.1.2 the technology node, tools of synthesis, place and route, and simulations of the ASIC design are explained.

3.1.1 POWER9 Platform

IBM POWER9 is an HPC machine which ranks among the top 5 computer systems as of the time of conducting this thesis [191]. IBM has also developed an infrastructure to take advantage of FPGAs in addition to their CPU-based systems. They have developed a customized Coherent Accelerator Processor Interface (CAPI) [192] to connect the POWER9 to the FPGA boards, providing a platform with processors and FPGAs as a complete package. Figure 3.1 shows a POWER9 system with the attached FPGA boards.

CAPI

CAPI provides a coherent access to the host memory for accelerators (FPGAs) eliminating the need of a driver and hence avoiding multiple data copies. In other words, CPU and FPGA can access to the same shared memory equally and directly by giving memory addresses. Shared memory is constituted of DDR4s, shown in Figure 3.1 in purple. Three versions of CAPI have

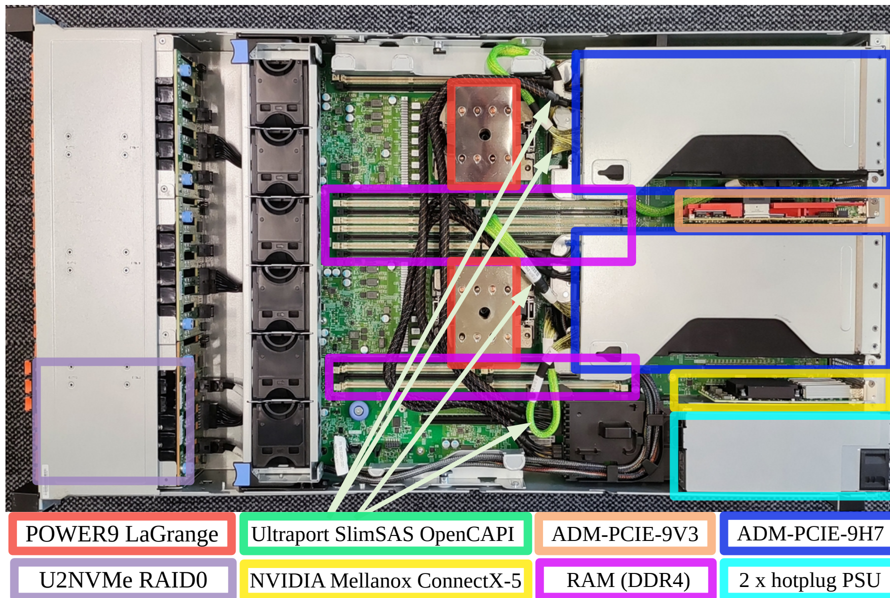


Figure 3.1: POWER9 system.

been released up to now. CAPI 1 and 2 protocols are on top of the PCIe communication BUS. While, OpenCAPI (CAPI3) [193] has a customized communication link (in Figure 3.1 marked by green). The practical bandwidth between host memory (shared memory) and FPGA for data sizes more than 100MB are approximately 3.3, 13 and 20GBps, respectively for CAPI 1, 2 and OpenCAPI.

OpenCAPI utilizes high-frequency differential signaling technology (BlueLink) that provides high bandwidth in addition to low latency inteconnection links. IBM introduced the Bluelink interconnect in POWER9 processors to provide NUMA interconnects across multiple POWER9 processors, and across POWER9 processors and GPU accelerators (e.g. Nvidia NVLink for Volta GPUs), and also to provide more generic memory coherent links to other kinds of accelerators, like FPGAs.

IBM has developed two C libraries, SNAP and libcxl, for the CPU side and one IP core, PSL, for the FPGA side. The SNAP library serves as the interface between the user application written in C and the libcxl. Libcxl, on the CPU side, is the CAPI translator, while PSL, on the FPGA side, translates CAPI to AXI and vice versa. Figure 3.2 illustrates the structure of a CAPI-equipped system.

AXI is a widely used bus protocol in modern computer systems. It provides a standard interface for IP cores and enables easy integration of different hardware components such as processors, memory controllers, DMA engines, and other IP blocks. The AXI bus protocol is widely used in SoC designs. The newest version of AXI, AXI4, has three types of AXI4-Full

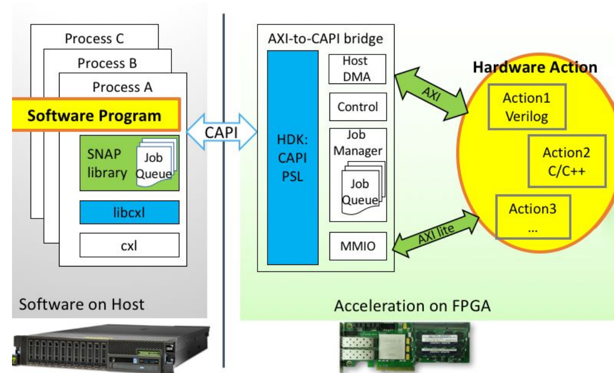


Figure 3.2: CAPI infrastructure [194].

(or AXI or AXI-mm), AXI4-Lite and AXI4-Stream. AXI4-Full is used for high performance memory mapped transactions between components in a system. AXI4-Lite is a simplified version of the AXI4-Full, designed for simple memory mapped control and status register access. AXI4-Stream is used for high-speed, continuous data transfers.

Figure 3.3 shows, in a CAPI-equipped system, how CPU and FPGA are communicating. First, in the C application two memory sections are allocated as the shared memory between CPU and FPGA. These memory sections are called *source memory* and *destination memory*. Source memory holds the data to be analyzed by the FPGA, while, destination memory keeps the final analyzed data. Then, the application puts the raw data on the source memory. After that, the C application should determine, for the FPGA, the size of the data, the address of the source memory and the address of the destination memory. To do this, it uses SNAP functions to write the starting addresses of source and destination memories and also the size of the data in MMIO (Memory-Mapped Input/Output). MMIO is mapping device registers to memory addresses. Then, by setting the start bit in the MMIOs, it triggers the FPGA to start processing data. Finally, the C application reads the idle bit of the MMIOs in a polling manner to check for the completion of the FPGA process. After job completion, the C application can use the results of the FPGA which are written in the destination memory. Checking/Setting control registers of the MMIO from the FPGA side is done using AXI4_Lite. While reading and writing from/to source and destination memory is done using AXI4-Full [195].

SNAP and CAPI setup and simulation

In order to work with SNAP and CAPI, the most important requirement is to install Xilinx Vivado. IBM provides all the necessary scripts and tcl files to make an FPGA project, simulate, synthesis, place and route and generate bitstream using Xilinx Vivado. The codes are open

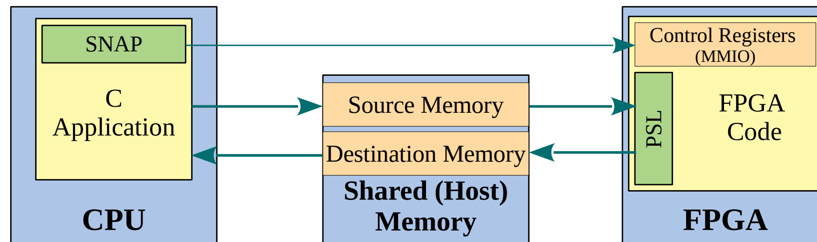


Figure 3.3: CPU and FPGA direct memory access.

source and publicly available for CAPI2 [194] and OpenCAPI [196]. The software platform that IBM provides to the users includes a folder for the user software code and one for hardware code. The software code uses SNAP libraries to communicate with FPGA through CAPI. The SNAP folder includes a make file to compile and link all libraries needed by user C code.

IBM also provides a platform for simulating PSL and user FPGA code and its communication with the C code. In other words, it simulates the whole system. The simulator is called Power Service Layer Simulation Engine (PSLSE). Through an interface user is able to choose the target FPGA board, the FPGA frequency and other configuration. For simulating the hardware part user can select the preferred simulator from the user interface. It also includes the Vivado simulator, xsim. The simulator shows the output of the user C code, but all of the FPGA waveforms are also saved automatically and can be seen by the user when the simulation is finished. Opening waveforms is done using provided commands in the scripts.

POWER9 configurations of the thesis contributions

POWER9 systems could have different CPU sockets, amount of RAMs and FPGA boards. In this sub-section we explain the configurations of the POWER9 for each of our contributions.

For our first contribution of this thesis in Chapter 4 the experiments are done on a POWER9 with two CPU sockets, each with 20 cores running at 2.3GHz. Each core has four threads so the platform provides 160 threads in total. The RAM of the system is 512GB and consists of 16 32GB DDR4 DIMMs running at 2666MHz. This POWER9 has two 2TB Micron SATA SSDs. The attached FPGA board is a CAPI2 enabled AlphaData ADM-PCIE-9V3 [197]. This board utilizes a Xilinx Virtex UltraScale Plus (XCVU3P) FPGA, which run at 250MHz and have 788K Flip-Flops (FF), 394K LookUp Tables (LUTs), 2280 Digital Signal Processing blocks (DSPs), 25.3Mb Block RAMs (BRAM) and 90Mb Ultra RAMs (URAM).

For our second and third contribution of this thesis in Chapter 5 we evaluate our proposal on a POWER9 system with one CPU socket which has 16 cores with 4 threads per core running at 2.3GHz, The system has 512GB of RAM (16 32GB DDR4 DIMMs running at 2666MHz)

and 2 FPGA boards. Two ADM-PCIE-9H7 FPGA boards [198] are attached to the POWER9 with OpenCAPI. The FPGA devices are Xilinx Virtex UltraScale Plus (XCVU37P), which run at 200MHz and have 2607K FFs, 1304K LUTs, 9024 DSPs, 70.9Mb BRAMs and 270Mb URAMs each.

3.1.2 ASIC Platform

For synthesis we use Genus tool from Cadence with the GlobalFoundries 22nm Fully-Depleted Silicon-On-Insulator (GF22FDX) technology. We use Synopsys Standard Cells libraries in GF22FDX technology for 8-Track platform. Placement and routing is performed using Innovus tool from Cadence using 8 metal layers. We compare post-synthesis and post-PnR netlists with the RTL design using the conformal Logic Equivalence Checker (LEC) tool from Cadence.

Both RTL and gate level simulations (GLS) of our ASIC in the whole SoC are performed using Xcelium from Cadence. Since the chip is still in the manufacturing process, for verification and performance evaluation we simulate the whole SoC using FPGA prototyping on an Alveo U280 FPGA board [199] connected to a server with AMD Ryzen 9 5900X CPUs (12 cores/24 threads) and 4×32GiB 3200MHz DDR4s. Alveo is build on the AMD (Xilinx) 16nm UltraScale+ architecture and offers 8GB of HBM2 460 GB/s bandwidth and includes PCI Express 4.0 support. The FPGA device (core + accelerator) runs on 50MHz and have 2607K FFs, 1304K LUTs, 9024 DSPs, 70.9Mb BRAMs and 270Mb URAMs.

3.2 Baselines and Input Sets

3.2.1 K-mer Counting in SMUFIN

The original software implementation of SMUFIN (first contribution) which is developed at BSC is written in C++ and its initial version was executed on 16 nodes of Marenostrum supercomputer (application explanation in Section). The code is private and stored on BSC gitlab platform. The baseline of this contribution is the software implementation of the multi-threaded SMUFIN run on all threads of the POWER9 CPU cores explained in Section 3.1.1. The inputs of the SMUFIN are DNA samples of a customized human genome based on the Hg19 (also known as GRCh37) reference. Random somatic variants with random insertions, deletions and inversions are applied to the inputs. In silico sequencing is simulated using ART Illumina21. The total input size is 312GB and consists of 256 gzip compressed FASTQ files.

3.2.2 WFA for Pairwise Read Alignment

The original software implementation of WFA code (second, third and fourth contributions) is written in C and is publicly available on github [200] (application explanation in Section). The code has been optimized to run on multiple threads using OpenMP programming. Different DP-matrix calculation algorithms are also implemented in the C code for comparison purposes. The baseline of second and third contributions is the software implementation of the multi-threaded WFA run on all threads of the POWER9 CPU cores explained in Section 3.1.1. While for the fourth contribution the baseline is the single core software implementation of the WFA on the RISC-V of the SoC explained in Section 3.1. The WFA codes includes a data (read) generator which allows user to generate an input file with the required number of read pairs, maximum read length and maximum error rate with random mismatches, insertions and deletions. We also use this data generator to produce our synthetic inputs. The specifications of the input sets of three last contributions are shown in Table 3.1.

The parameters of the synthetic input sets of Table 3.1 are representative of the current sequencing platforms [201]. Using the same methodology as in related studies [19, 161, 172, 173], we randomly generate input sets with different maximum read lengths and error rates and with random mismatches, insertions and deletions.

For the real input sets we select two publicly available datasets representative of state-of-the-art sequencing technologies. The PacBio HiFi real dataset corresponds to a PacBio High Fidelity (HiFi) sample from HG002. This dataset was obtained from PrecisionFDA Truth Challenge V2 and can be found at [202]. The PacBio CCS real dataset corresponds to a PacBio Circular Consensus Sequencing (CCS) sample from HG002. This dataset was obtained from NIST's Genome in a Bottle (GIAB) project and can be found at [203].

Table 3.1: Input sets of second, third and fourth contributions.

<table>
<tr><th>Contributions</th><th>Read Length</th><th>Error Rate (%)</th><th>Number of Reads</th><th>Input Type</th></tr>
<tr><td rowspan="7">2
WFA-FPGA
(Short Reads)</td><td rowspan="2">100</td><td>5</td><td rowspan="7">10M</td><td rowspan="7">Synthetic</td></tr>
<tr><td>8</td></tr>
<tr><td rowspan="3">150</td><td>3</td></tr>
<tr><td>5</td></tr>
<tr><td>8</td></tr>
<tr><td rowspan="2">300</td><td>3</td></tr>
<tr><td>5</td></tr>
<tr><td rowspan="17">3
WFA-FPGA
(Long Reads)</td><td rowspan="3">1K</td><td>5</td><td rowspan="3">1M</td><td rowspan="15">Synthetic</td></tr>
<tr><td>10</td></tr>
<tr><td>20</td></tr>
<tr><td rowspan="3">5K</td><td>5</td><td rowspan="3">500K</td></tr>
<tr><td>10</td></tr>
<tr><td>20</td></tr>
<tr><td rowspan="3">10K</td><td>5</td><td rowspan="6">100K</td></tr>
<tr><td>10</td></tr>
<tr><td>20</td></tr>
<tr><td rowspan="3">25K</td><td>5</td></tr>
<tr><td>10</td></tr>
<tr><td>20</td></tr>
<tr><td rowspan="3">50K</td><td>5</td><td rowspan="3">20K</td></tr>
<tr><td>10</td></tr>
<tr><td>20</td></tr>
<tr><td colspan="2">PacBio HiFi: avg len = 12.8K</td><td rowspan="2">1M</td><td rowspan="2">Real</td></tr>
<tr><td colspan="2">PacBio CCS: avg len = 9.6K</td></tr>
<tr><td rowspan="6">4
WFA-ASIC</td><td rowspan="2">100</td><td>5</td><td rowspan="2">1K</td><td rowspan="6">Synthetic</td></tr>
<tr><td>10</td></tr>
<tr><td rowspan="2">1K</td><td>5</td><td rowspan="2">100</td></tr>
<tr><td>10</td></tr>
<tr><td rowspan="2">10K</td><td>5</td><td rowspan="2">20</td></tr>
<tr><td>10</td></tr>
</table>

Chapter 4
K-mer Counting FPGA Accelerator

4.1 Introduction

K-mer counting is a time-consuming step in various genomics applications that is complicated by the large memory requirements, especially when analyzing large datasets such as the entire human genome. Some methods have been proposed to speed up k-mer counting [50, 52–57, 118–120, 123, 124, 204], being the reduction of the memory requirements one of the main goals of many works [50, 52–57].

The computational capabilities of accelerators like FPGAs have persuaded engineers from different fields to propose hardware/software co-designs to accelerate their applications. However, the limited memory capacity of FPGAs is a crucial impediment to designing accelerators for applications that process massive amounts of data, as it is the case of k-mer counting. For this reason, in the last years chip manufacturers have put a lot of effort in developing communication protocols between processors and accelerators that provide accelerators with direct access to the processor memory. For instance, the IBM CAPI [192] protocol allows FPGAs to directly access the processor memory at a speed of up to 20GB/s. This feature makes CAPI-based systems ideal for accelerating workloads with large memory requirements such as genomics algorithms.

This chapter presents a hardware/software co-designed accelerator for k-mer counting using a CAPI-enabled FPGA. The main contributions of this chapter are: (i) an RTL design for FPGAs to accelerate the processes of extracting reads and generating k-mers; (ii) modifications to the software algorithm to remove dependencies between parallel threads, reduce overheads and use less memory; and (iii) three data compaction mechanisms to minimize the memory and disk requirements. We integrate the co-designed accelerator in an adapted version of SMUFIN [18], a state-of-the-art reference-free algorithm that identifies somatic mutations, and we evaluate it on a CAPI-enabled high-performance computing node with a dual-socket

POWER9 processor and an FPGA. Results show that the co-design outperforms the CPU-only design by 2.14× while consuming 2.93× less energy and 1.57× less memory.

4.2 Background

4.2.1 DNA Reads, K-mers and K-mer Counting

Read is a segment of DNA that is obtained through the process of sequencing. NGS technologies provide reads of lengths between 50 and 300 bases. A DNA *base* is represented by one of the letters A, C, G or T, so a compact 2-bit numerical representation is often used. A *k-mer* is a *k* length sub-string of a read, so a read of length *n* has *n-k+1* possible k-mers. *K-mer counting* is the process of counting the occurrences of all possible k-mers within a dataset.

4.2.2 SMUFIN Overview

SMUFIN [18] is a state-of-the-art application that detects DNA mutations by comparing normal and tumoral DNA samples from the same patient without requiring a reference genome. To do so, SMUFIN counts the frequency of the k-mers in the two sets of reads (normal and tumoral) separately and compares them to find imbalances that determine candidate DNA positions with mutations.

Figure 4.1 shows a diagram of the SMUFIN application, which consists of three main phases of *count, filter* and *group*. The count phase can also include two optional auxiliary steps of *prune* and *unify*. Unlike other applications, SMUFIN does not reconstruct genome sequences, yet like many *de-novo* assembly algorithms, its first phase is k-mer counting. The *k-mer counting* phase first generates all k-mers of reads from normal and tumoral datasets. Then, generates a histogram based on the k-mers of the whole dataset. The range of the length of generated k-mers in this algorithm is selectable between 24 and 32. This range has been determined by domain experts to be both unique enough to ensure precise genome alignment and general enough to facilitate accurate mutation detection. Once created, the histogram remains static and is not subject to any further updates. However, it is accessed by the filtering phase of the algorithm numerous times, with a frequency in the hundreds of billions. The *filtering* phase constructs the interesting reads and k-mers database. Biologists establish a set of criteria that reads and k-mers must fulfill to be regarded as interesting for further mutation analysis and reconstruction. The set of criteria encompasses various characteristics of individual reads (i.e., contains a particular sequence of bases, quality markers of the bases in the reads)

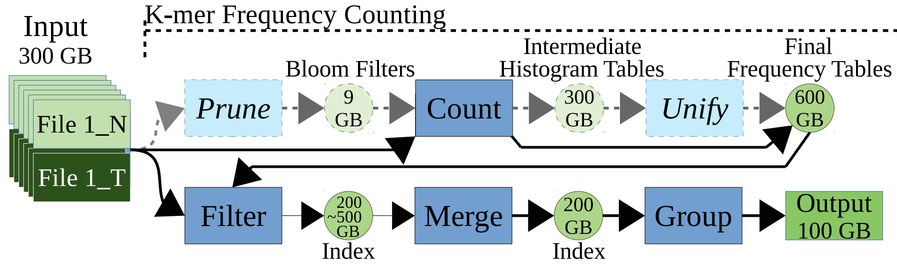


Figure 4.1: SMUFIN mandatory and optional (light color) phases.

and k-mers (i.e., frequency counts of the k-mers). The *grouping* phase groups reads with the same interesting k-mers together to reconstruct the mutations. After forming groups of reads, they are aligned with each other based on the locations of the matching k-mers. If the positions of matching k-mers are inconsistent between normal and tumoral reads, it may indicate the presence of a mutation [77].

SMUFIN processes the whole human genome, which is stored on multiple compressed FASTQ files that require 200 to 400GB of disk storage. The original SMUFIN algorithm [18] was designed to run on multiple nodes of a supercomputer with enough memory to store the whole data required for human genome analysis. Its successor, SMUFIN2 [205], rethinks the initial design to be more scalable and flexible. To do so, SMUFIN2 uses two levels of partitioning to distribute data among processes and threads. The first level splits data for processing into one or more partitions, and each one of these partitions can then be distributed either concurrently in multiple nodes or sequentially in a single node. In this chapter we use an adapted version of SMUFIN2 where the first level partitions are based on batches of input files instead of a logical data distribution. For the second level partitions we use the same mechanism as SMUFIN2, which is based on the first 5-mers of a k-mer. In the rest of this chapter we use the term SMUFIN to refer to our adapted version that partitions the data in batches of files, and partitioning refers to the second level of partitioning. The adapted version of SMUFIN2 is explained in the next subsections and it is also used as the CPU baseline in the evaluation presented in Section 5.5.

4.2.3 SMUFIN K-mer Counting Structure

K-mer counting is one of the most time-consuming phases of SMUFIN, up to 35% of the whole processing time [120]. Since the input of the count step is the whole human genome, it requires a huge amount of memory. For this reason, this algorithm is a very good candidate for being accelerated using CAPI-supported FPGAs, as they can access host memory directly without

the need of making multiple copies of the genome data. To reduce the memory requirements of this step, in the previous accelerations of SMUFIN [120], two optional steps of prune and unify are used.

The prune step creates Bloom filters from all input files to identify and discard unique k-mers later in the count step. Bloom filters reduce the memory footprint of the count step by reducing the total amount of k-mers to be analyzed and the size of the intermediate histogram tables to be stored on disk. However, generating the Bloom filters is time-consuming as three hash functions are applied to each k-mer [53, 123].

The count step, due to the large memory requirements for the whole input, processes input files in batches. It reads a batch of input files, generates k-mers, and counts their frequency in the normal and in the tumoral inputs. Then it stores the k-mers and their two corresponding frequency counters on disk as intermediate histogram tables. Afterwards, it reads and analyzes the next batch of input files, repeating this process until all input files are analyzed. The resulting intermediate histogram tables are represented as hash tables with the format *pair<(uint64) k-mer, (uint16) counter[2]>*.

Figure 4.2 shows the structure of the count step. This step has configurable numbers of producer and consumer threads, two of each type in the example. SMUFIN uses a k-mer partitioning mechanism that spreads k-mers among partitions based on their first five bases. The number of partitions can be configured between 1 and 128. Based on the statistical distribution of the k-mers, a lookup table is used to assign partitions to k-mers with the aim of distributing k-mers evenly among partitions. The number of producer threads is equal or less than the number of files in each batch, and the number of consumer threads is equal to the number of partitions.

Producer threads read data from one input file, extract reads, generate k-mers, convert k-mers to their numeric representation, check the lookup table to determine the partition where each k-mer belongs, and insert the k-mers into the producer-consumer queues of each partition. Consumer threads are in charge of counting the number of k-mers of each partition. To do so, they read k-mers from the producer-consumer queues, discard unique k-mers using the Bloom filters generated in the prune step, count the frequency of each k-mer, and populate the intermediate histogram tables that are stored on disk. The name of these tables on disk consists of two numbers, its partition and its index. For example, if there are 100 partitions and the count step analyzes 16 batches of files, the partition numbers are 0 to 99 and the indexes are 0 to 15.

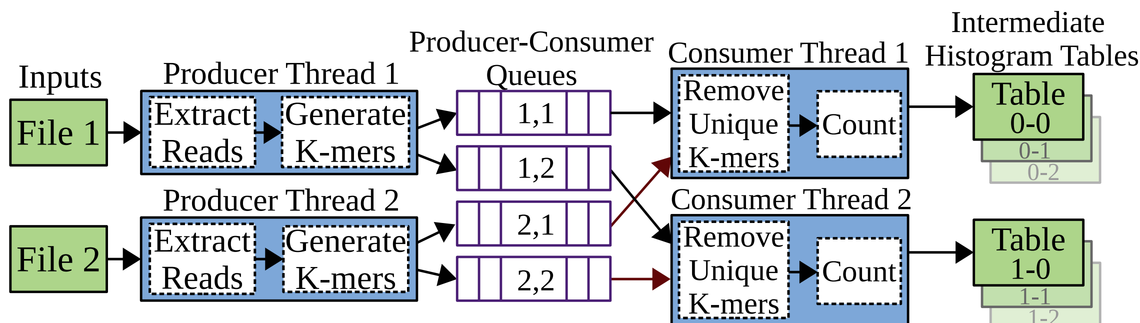


Figure 4.2: Procedure of the count step.

The unify step merges all the intermediate histogram tables of each partition into one final frequency table for that partition. The intermediate histogram tables are sequentially loaded from disk and, for each partition, the final frequency table is updated. Final frequency tables are represented as hash tables with a different layout than the intermediate histogram tables, so the unify step converts them to the final layout that is suitable for next steps. Finally, the unify step re-checks all the final frequency tables to discard unique k-mers that can appear due to false positives in the Bloom filters.

4.3 Acceleration Method of K-mer Counting in SMUFIN

This section presents a hardware/software co-design for k-mer counting. The proposed design consists of restructuring the count step to enable the usage of the FPGA and improving the unify step, which for our co-design is a mandatory step. The accelerated count phase is compatible with the rest of SMUFIN phases, which are left unaltered, and could also be integrated in other genomic applications.

4.3.1 Prune Step

The proposed accelerator does not use Bloom filters, so the prune step is not needed. As explained in Section 4.2.3, the Bloom filters created in the prune step reduce the memory and disk requirements by reducing the amount of k-mers to be analyzed. This allows the count step to use larger batches of input files and to generate smaller intermediate histogram tables, which decreases the memory usage and execution time of the unify step. The proposed design skips the prune step at the cost of using more disk space, and the count and unify steps are modified to use less memory and perform faster even without Bloom filters and for smaller batches of input files. Removing the unique k-mers is postponed to the unify step.

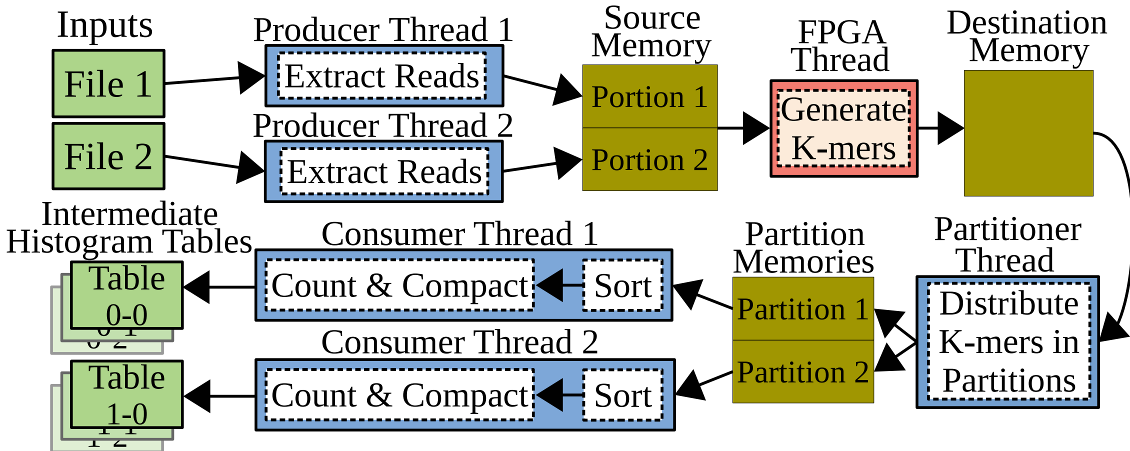


Figure 4.3: Proposed procedure of the count step.

4.3.2 Count Step

Figure 4.3 illustrates the proposed design for the count step. The main changes with respect to the original design consist of offloading the generation of k-mers to the FPGA, adjusting the behavior of the producer and consumer threads, and using a sort mechanism instead of hash tables in the consumer threads.

The producer-consumer queues of the original count step create inter-thread dependencies. The proposed design uses *partition memories* to pass data from producers to consumers instead of queues. The producer and FPGA threads first generate all the k-mers for a batch of input files, then one partitioner thread stores them in the partition memories, and finally the consumer threads start working. Each partition belongs to only one consumer, minimizing thread dependencies.

Producer Threads

Producer threads read the input files, extract the reads, and write them into the source memory. The reads in the source memory are marked as normal or tumoral so that the FPGA can determine their type. To avoid adding storage overheads, we use the first byte of each read to encode its type. If the read belongs to a normal file the first base is written as A, C, G, T or N (for unknown), otherwise the first read is incremented by 1, so it becomes B, D, H, U or O.

To maximize parallelism and avoid synchronization overheads, a configurable number of producer threads read small parts of different files and fill different portions of the source memory. Note that, when a producer thread finishes reading a file, the extracted reads may not be enough to fill the entire portion of that thread's source memory. When this happens, the

producer thread fills the remaining space with ’S’ reads indicating to the FPGA that these reads have to be skipped.

FPGA Design

The FPGA design is controlled by an FPGA thread. The FPGA thread continuously checks the status of producer threads and, when they all have filled their portion of source memory, triggers the FPGA design. The FPGA design reads data from the source memory, generates the k-mers, and writes them into the destination memory. Figure 4.4 shows a block diagram of the FPGA design, which consists of two main modules: the *Read_Extr* module extracts reads from the input data and encodes them with a 2-bit representation, and the *K-mer_Extr* module generates k-mers.

The Read_Extr module is responsible for extracting reads from input words. CAPI2 defines that the data width of the FPGA is 64 bytes. Since the read length is 80 bases (bytes) for our input dataset, the reads are split between input words. The *state machine* determines, in each state, what portion of the incoming input data belongs to one read and what portion to the next read. The incomplete reads are stored in *Temp Reg* and, once completed, they are put in *Read Reg*.

The Read_Extr module adds one bit to each read to define its type. The type is determined from the first base of each read, as explained previously. If the first base is A, C, G, T or N, the type bit is set to 0 for normal. Otherwise, it is set to 1 for tumoral and the first base is converted back to A, C, G, T or N. The read and its type are stored in the *Read_FIFO* in 161-bit words (80 bases/read $\times$ 2 bits/base + 1 type_bit).

The Read_Extr module also identifies k-mers with unknown bases, which are represented as ’N’ in the input data. These k-mers are marked as invalid and are omitted later by the partitioner thread. To do so, a valid bit is assigned to each base of a read and is set to one if the base is not ’N’. The valid bits of the reads are stored in a separate *Valid_FIFO* with a width of 80 bits, so each read in the Read_FIFO has a corresponding entry in the Valid_FIFO. In addition to unknown bases, the Read_Extr module handles reads with all their bases set to ’S’, which is the format that producer threads use to specify invalid entries in the source memory. These reads are skipped by not writing them in the Read_FIFO nor in the Valid_FIFO.

The next FPGA module is the K-mer_Extr, which is in charge of generating the k-mers. This module reads data from the Read_FIFO and separates the first bit, which contains the type of the read (tumoral or normal), from other bits that contain the read itself. In this FPGA design the k-mer length is set to 30. As the data width of the FPGA is 64 bytes, eight k-mers

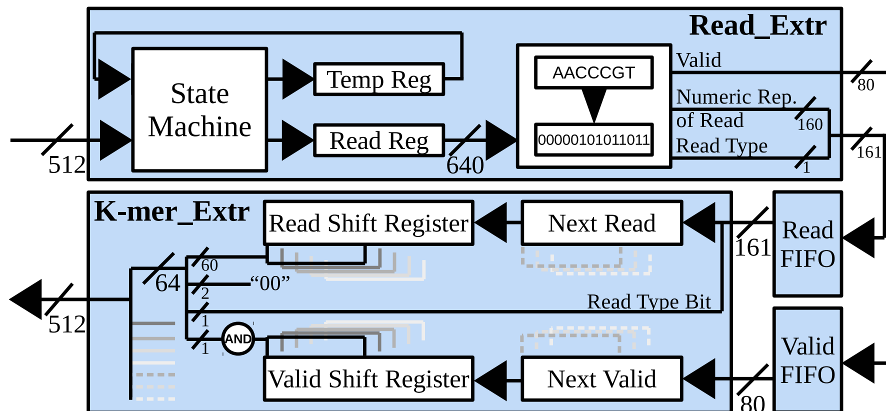


Figure 4.4: Block diagram of FPGA k-mer generator modules. The connections are labeled with their width, in bits.

(each eight bytes) of the read in the *Read Shift Register* are picked during each clock cycle, concatenated with the type bit of their read and their corresponding valid bits, and then written to the destination memory. The valid bit is obtained by an AND operation on the valid bits in the *Valid Shift Register*, which correspond to all the bases of the k-mer. During the same clock cycle, the Read and the Valid Shift Registers are shifted 16 and 8 bits, respectively. If in a given clock cycle the Read Shift Register contains less than eight k-mers, k-mers of the *Next Read* are generated to ensure a total of eight k-mers are processed in each clock cycle, and the Next Read and the Valid Read registers are shifted accordingly.

The FPGA design has two pipeline stages, one for each module. Although Read_Extr can analyze 64 bytes of data at each clock cycle, the K-mer_Extr module is the bottleneck because the output data of this module is $5\times$ larger than its input data. With data width equal to 64 bytes at the output side, only 1/6 of the output data of the K-mer_Extr module is sent to the destination memory at each clock cycle. Therefore, the Read and Valid FIFOs can become full and impose a wait time on the Read_Extr module. With a clock frequency of 250MHz the peak bandwidth is 16GB/s, or 2G*k-mer*/s nominally, but in practice 1.7G*k-mer*/s is obtained.

Partitioner Thread

When the FPGA action finishes, a partitioner thread in the CPU checks the destination memory, terminates invalid k-mers, determines the partition of each k-mer, and copies the k-mers to their corresponding partition memory. Meanwhile, the producer threads start loading the rest of the reads in the source memory.

Consumer Threads

The consumer threads count the frequency of the generated k-mers in partition memories. Each consumer is responsible for one partition memory. Each consumer thread first sorts and then counts the k-mers of its partition. After sorting, the counting is easily accomplished by comparing each k-mer with the previous one and, if they match, incrementing the frequency counter of the k-mer. After calculating the frequency of all the k-mers, consumer thread stores its intermediate histogram table on disk.

The benefits of using sort instead of hash tables, in addition to improved performance, are the lower requirements for memory and disk. The Google sparse hash tables used in the CPU-only version need four extra bytes per item [120]. So in that design, to avoid writing extra bits on disk, a loop iterates on all the elements of the hash tables and their values are extracted and copied in another part of memory that is then saved on disk. This adds extra time and memory usage that is eliminated in the proposed design. As there are no hash tables in our design, k-mers and their corresponding counters are directly put in a byte aligned array of memory.

The performance of writing the intermediate histogram tables to disk heavily depends on the speed of the I/O subsystem. Hence, compacting the data can lead to better performance and less storage requirements.

Data Compaction

As explained in Section 4.2.3, the intermediate histogram tables keep two frequency counters per k-mer in a format of *pair<(uint64) k-mer, (uint16) counter[2]>*. The consumer threads use three techniques to compress the information of these tables.

The first technique consists of grouping the batches of files by their type, so that a batch only contains normal or tumoral files. With this technique all the k-mers of the batch of files have the same type so, instead of two frequency counters per k-mer (one tumoral and one normal), only one counter per k-mer is required. First we feed the system with normal files and then with tumor files. As the number of normal and tumoral files is the same, the type of each k-mer is easily distinguishable from the index of its table name in later steps. If the index of a table is in the first half of all indexes, that table contains normal k-mers, otherwise it contains tumoral k-mers.

The second technique is based on the observation that, in practice, the frequency of 99% of the k-mers is below 255. To exploit this, we use a variable length counter controlled by a bit

		Index				
		0	1	2		10
Partition	0	616	397	0	...	-
	1	1023	109	543	...	-
	2	579	789	766	...	-
	⋮					
	98	969	295	310	...	580
	99	296	318	205	...	-

Figure 4.5: Example of partitioning lookup table.

cntr_ex_en (counter extension enable) per k-mer that indicates the size in bytes of the counter field (0 for one byte and 1 for two bytes). This compaction method is applicable when the k-mer field contains at least one unused bit. For example, with a k-mer field of 64 bits, this method is suitable for k-mer lengths of 31 and lower, where at most 62 bits of the k-mer field are used (31 bases $\times$ 2 bits/base) and at least two bits are free and can be used to encode the *cntr_ex_en* bit.

The third technique reduces the space required to store one k-mer. This is achieved by replacing the first 5-mer of each k-mer with its index in the partitioning lookup table (see Figure 4.5). This technique is only applicable when the number of partitions is more than 35. In this case each partition will contain less than 32 5-mers, so the index number of each 5-mer in the lookup table will be in the range of 0 to 31. Therefore, a 10-bit 5-mer is replaced by its 5-bit index. In our FPGA design the k-mer length is fixed to 30 and each k-mer occupies eight bytes (61 bits = 30 bases $\times$ 2 bits/base + *Cntr_ex_en*). With this data compaction method, each k-mer fits in 56 bits (61 bits_before_compaction - 10 bits/5-mer + 5 bits/index). Note that this method is only useful when the number of saved bits reduces the k-mer field by one byte. For example, with a k-mer length of 31, this method is not useful because 58 bits are needed after compaction, which still requires 8 bytes.

These three compaction methods allow to save 30% of the space needed for storing the k-mers and their frequency counters. In the proposed design, hash table containers are no longer used. Instead, intermediate histogram tables are written sequentially in a byte aligned array of memory. In this array, each k-mer occupies 7 bytes and its corresponding frequency counter occupies one or two bytes, depending on its value. Figure 4.6 illustrates an example of the format of storing a k-mer and its frequency after data compaction.

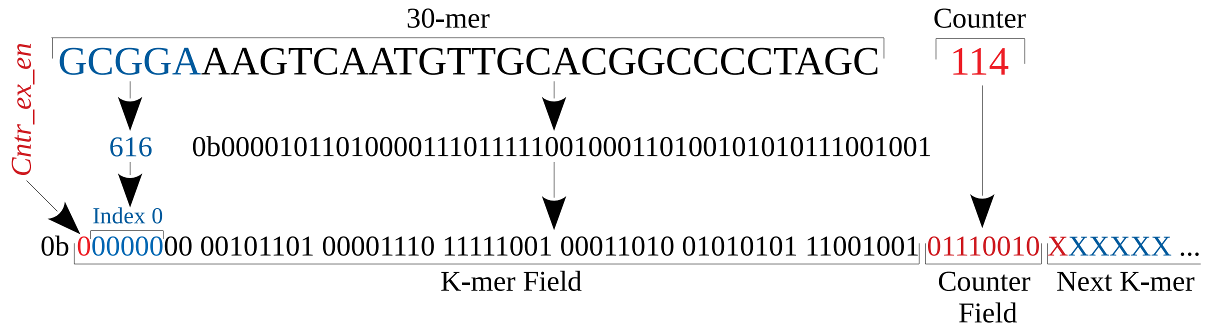


Figure 4.6: An example of k-mer and its frequency layout after data compaction.

4.3.3 Unify Step

The unify step has three responsibilities: merging all the intermediate histogram tables of each partition into a final frequency table of that partition, discarding unique k-mers, and converting the final frequency tables layout. The proposed unify step loads only the first k-mer of all intermediate histogram tables in memory, and determines the smallest k-mer of the loaded ones for each partition. If two or more k-mers are equal to the smallest one, their values are added together and stored in the final frequency table while, it is discarded if it is unique. Then the next k-mers of the tables are loaded and this procedure continues for all the k-mers of all the intermediate histogram tables.

The new unify design has four major changes compared to the CPU version: (i) it merges all intermediate histogram tables at once; (ii) it only loads the first k-mers of each table in memory; (iii) it removes unique k-mers before inserting them into the final frequency tables; and (iv) it stores final frequency tables as sorted vectors. The first modification makes the design faster while the others make it more memory efficient.

4.4 Evaluation and Results

4.4.1 Experimental Setup

As explained in Chapter 3 the experiments of this contribution are done on a POWER9 system with 2×20 cores (160 threads) running at 2.3GHz, with an attached CAPI2-enabled AlphaData ADM-PCIE-9V3 FPGA board. The FPGA code is written in VHDL and compiled using Vivado v2018.1. In this design the FPGA runs at 250MHz. To integrate our accelerator with CAPI2 interface, we use the platform provided by IBM in github [194].

Table 4.1: FPGA resources utilization (%) for the CAPI-related IP cores and for the proposed k-mer counting accelerator.

	LUT	FF	DSP	BRAM	URAM
CAPI	10.04	9.86	1	32.99	-
K-mer counting	11.25	0.63	-	2.57	-
Total	21.29	10.49	0.04	35.56	0

The input DNA samples consists of 256 gzip compressed FASTQ files, 128 for normal samples and 128 for tumoral. The read length is 80bp so each input file contains approximately 16M reads. To increase the speed of reading and writing files, half of input and output files are saved on one disk and the other half on the other disk.

In the evaluation we use the adapted version of SMUFIN as CPU baseline, as explained in Section 4.2.2. SMUFIN accepts k-mer lengths between 24 and 32. We use a k-mer length of 30 in both the CPU baseline and the FPGA co-design. To verify the correctness of our accelerator, we compare the outputs obtained in the executions with the accelerator with the ones obtained in the executions with the software baseline.

4.4.2 Results

Table 4.1 shows the resource utilization of the FPGA. The k-mer counting accelerator occupies only a small fraction of the FPGA resources, while the CAPI modules present a higher resource utilization, specially in FF and BRAM. Even consuming few resources, the proposed accelerator reaches the maximum bandwidth of CAPI2, so a more complex design would not provide any benefit. For future generations of CAPI that provide more bandwidth, like OpenCAPI, there is room to scale up the proposed design and utilize more FPGA resources.

The execution time of both designs is summarized in Figure 4.7. In this exploration we try all the possible combinations of batch sizes (16, 32, 64, 128), producer threads (16, 32) and consumer threads (100, 120). Other values for these parameters reduce the performance for both designs. The CPU+FPGA design results for a batch size of 128 are not shown because the system runs out of memory in this configuration. In this section the term "memory" refers to the main memory (DRAM) of the system. The CPU-only design uses producer-consumer queues of 32K elements as it gives the best performance. The best configuration for the CPU+FPGA design is 120 consumer threads, 16 producer threads and batches of 16 files, while for the CPU-only design it is 120 consumer threads, 32 producer threads and batches of 32 files.

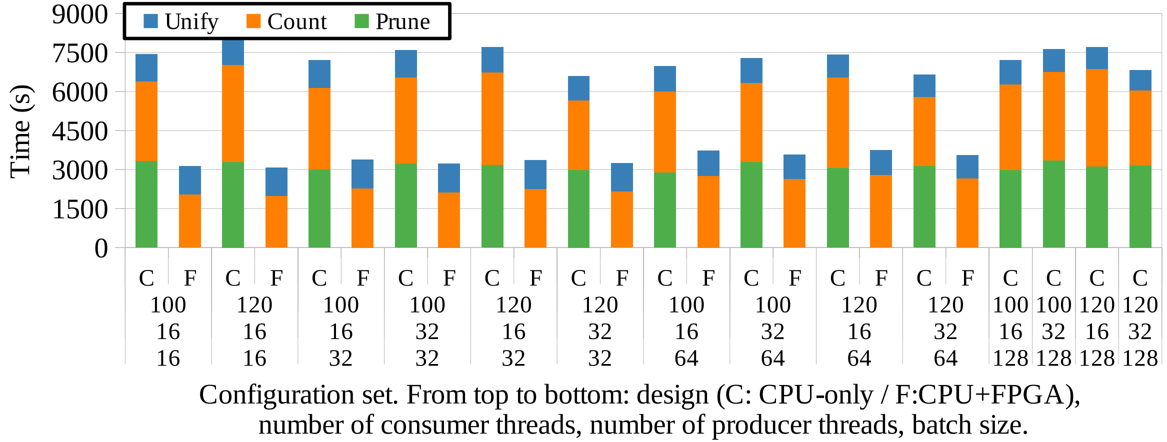


Figure 4.7: Execution time for different designs and configurations.

Comparing the best execution times of both designs, the CPU+FPGA design presents a speedup of 2.14× for the count phase.

Figure 4.7 also shows the distribution of the execution time among the different steps. The CPU+FPGA design does not use Bloom filters, so the time-consuming prune step is not needed. Even without Bloom filters, the count step is 1.34× faster by offloading work to the FPGA. Although the unify step of the CPU+FPGA design merges larger tables, its execution time is similar to the unify step of the CPU-only design with the prune step enabled.

Results show that the count step of the CPU+FPGA design presents less variability in the execution times when changing the number of producer and consumer threads. For a batch size of 32 files, the performance difference between the fastest and the slowest configuration in the CPU+FPGA design is 5.94%, while for the CPU-only design it is 24.59%. The higher variability of the execution times of the CPU-only design is due to the synchronization between producer and consumer threads in the producer-consumer queues. If the number of threads is not tuned properly, producer and consumer threads experience idle time when the queues are full or empty, negatively affecting performance. In contrast, the CPU+FPGA design decouples the operation of the two types of threads, resulting in a more stable performance.

In addition to the proposed FPGA co-design, we evaluate different design alternatives to try to further increase performance. One approach is to offload the partitioner thread to the FPGA. In this approach, the FPGA detects the partition of each k-mer and directly writes it in its corresponding partition memory. Since k-mers could belong to any partition, the memory write addresses are not sequential and the FPGA-memory bandwidth is not efficiently used. However, we try to use the bandwidth more efficiently by buffering 64 k-mer of each partition inside the FPGA FIFOs and then writing them into the memory in one burst transaction. We

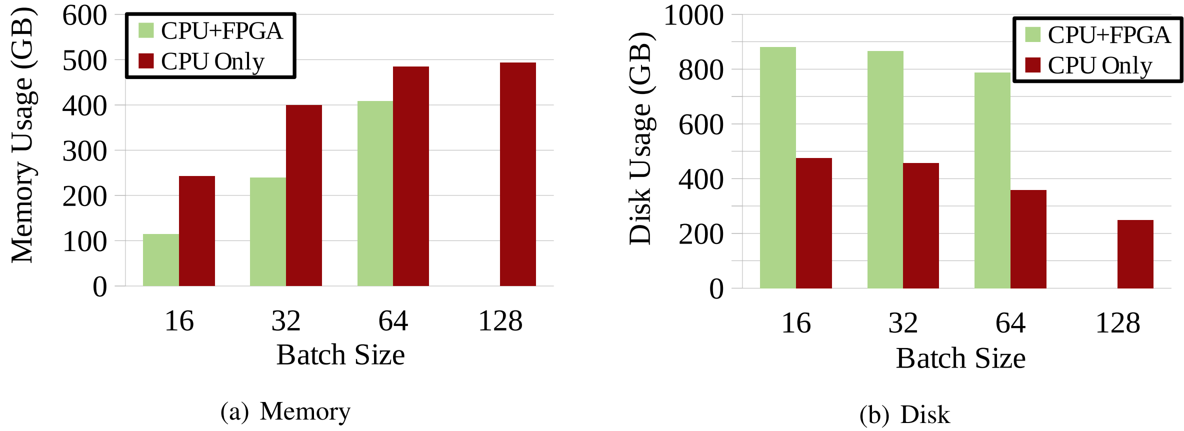
Table 4.2: Execution time and disk space requirements of the count and unify steps with the prune step enabled and disabled.

Prune Status	Time (s)				Disk (GB)
	Prune	Count	Unify	Total	
Enabled	3335	3049	1050	7434	475
Disabled	-	3994	16323	20317	1101

observe this approach also provides negligible speedup compare to partitioning in the CPU. Moreover, partitioning in the FPGA limits system configurability as it requires to reprogram the FPGA when the number of partitions changes. We also try to offload the work of the consumer threads to the FPGA, but results show that performing this work using 160 parallel threads is faster than doing it in a single FPGA. Finally, to observe the effect of the FPGA acceleration on the execution time of the co-design, we replace the FPGA design by an analogous CPU code and we observe that the k-mer generation and the whole count step are slowed down by 150× and 10×, respectively.

As explained in Section 4.2.3, the prune step that generates the Bloom filters for the count step is optional. To quantify the impact of the prune step, Table 4.2 compares the execution time of the whole k-mer counting phase with and without the prune step for the CPU-only design. This experiment uses 16 producer threads, 100 consumer threads, and batches of 16 files. The system runs out of memory for batches with more than 16 files when the prune step is disabled. Results show that enabling the prune step improves performance by 2.73× because, although the execution of this step takes 3335 seconds, it provides important speedups in the count and unify steps. With the prune step enabled, the count step avoids the computation of unique k-mers, which reduces the execution time of this step by 1.31× and makes the intermediate histogram tables smaller. The unify step takes 15.54× more time when the prune step is disabled because it needs to merge larger tables, and the execution time of this step grows exponentially with the size of tables to be merged. The unify step can merge half of the intermediate histogram tables at once when the prune step is enabled, while without the prune step only 1/12 of these tables fits in memory at once. When the prune step is enabled the required disk space is 475GB while, with the prune step disabled, it reaches 1101GB. This shows that 57.86% of the required disk space is occupied by unique k-mers which do not provide useful information.

The number of files in each input batch affects disk and memory usage. Figure 4.8 illustrates the memory and disk storage requirements of the count step for different batch



(a) Memory

(b) Disk

Figure 4.8: Memory and disk usage of the count step.

sizes. The memory usage of the prune and unify steps are not dependent on the number of files in each batch and are almost constant. The prune step uses 90GB of memory, while the unify step requires 470GB of memory with the CPU-only design and 300GB of memory with the CPU+FPGA design, i.e., 1.57× less memory than the CPU-only design. As seen in Figure 4.8(a), the memory usage of the count step scales up with the batch size and the CPU+FPGA design needs less memory for any batch size below 128. However, its count step does not fit in the system memory for batches of 128 files. Regarding the best performing configurations (from Figure 4.7), the CPU+FPGA design needs 114GB of memory while the CPU-only design requires 400GB. Regarding the disk utilization, Figure 4.8(b) shows that, as the batch size increases, the disk usage decreases in both designs. This happens because, with larger batches, more files are merged together at once and more data is packed in the intermediate histogram tables, which reduces the required disk space. In conclusion, the CPU+FPGA design is faster and its count step requires 3.51× less memory. These memory and performance improvements are achieved at the cost of using 1.93× more disk space, 881GB in the CPU+FPGA design versus 456GB in the CPU-only design.

The power consumption of the count step is illustrated in Figure 4.9. We measure the power consumption of the whole node with in-band readings from Linux to the On Chip Controller (OCC) [206]. It can be observed that the power consumption of the producers is lowered in the CPU+FPGA design by more than 200W. Peaks in the CPU+FPGA design belong to the sort part of consumer threads. There are 16 of them as the total input is analyzed in 16 batches. The power consumption of the unify step of the CPU+FPGA design, which is not shown in Figure 4.9, is also 170W below that of the CPU-only design. All together, for the whole count

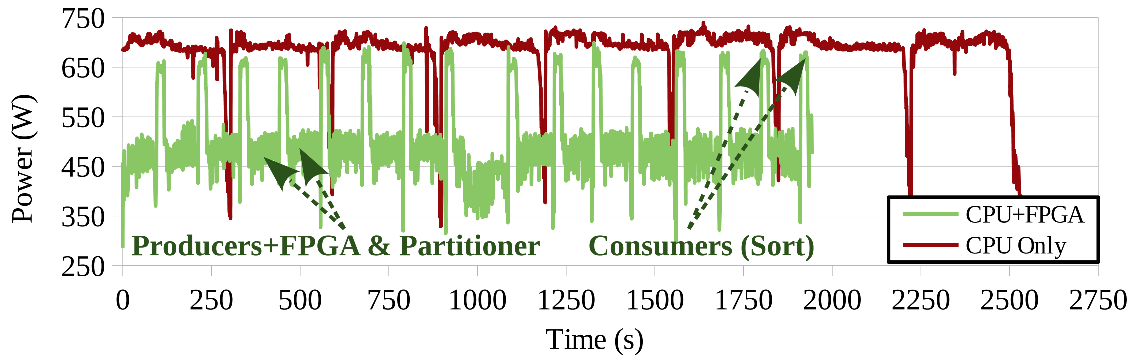


Figure 4.9: Power consumption of the count step over time.

phase, the CPU+FPGA design consumes 0.42kWh of energy while the CPU-only one uses 1.23kWh, so the total energy-to-solution is improved by a factor of 2.93×.

4.5 Conclusions

K-mer counting is one of the most time-consuming phases of many genomic applications. Although FPGAs are very well suited to accelerate the k-mer counting algorithm, their reduced memory capacity is a big limiting factor to handle the vast amount of data that is processed with this algorithm. To overcome this limitation, the IBM CAPI interface allows FPGAs to directly access the processor memory.

This chapter presents a hardware/software co-designed accelerator for k-mer counting on CAPI-enabled FPGAs. The proposed approach consists of an FPGA design to accelerate the generation of k-mers and a combination of optimizations on the software side to eliminate thread dependencies, replace hash-tables for sorted vectors, and re-define the memory layout using three data compaction mechanisms. The co-designed accelerator is able to efficiently count k-mers and unify the results without the need of Bloom filters, so the time-consuming phase to generate them can also be avoided. All together, the proposed co-design greatly accelerates the k-mer counting algorithm while reducing the memory requirements, the thread synchronization overheads, and the sensitivity to the algorithm parameters. Results show that the presented co-design achieves a speedup of 2.14× over the CPU-only design while reducing the energy-to-solution and the memory requirements by 2.93× and 1.57×, respectively. The proposed co-design can be easily integrated in the k-mer counting phase of any genomic application and can be scaled up in future CAPI-based systems that provide higher bandwidth between the FPGA and the processor memory.

Chapter 5

WFA FPGA Accelerator

5.1 Introduction

The current most widely used sequencing technologies, NGS, sequence DNA in millions of small fragments, aka reads, of length 50 to 300 base pairs. While, third generation sequencing machines, which are expected to be widely used in the future, generate longer reads of thousands of base pairs.

Read mapping is the first step in most DNA sequence analysis pipelines, which determines the location of each read in the reference genome. One of the main steps of read mapping, is pairwise read alignment which aligns pairs of sequences, input reads verses candidate locations of the reference genome. Modern read mappers, for their pairwise read alignment step, use variants of SW algorithm which uses dynamic programming and require quadratic $O(n^2)$ execution time and memory, where n is the sequence length. Hence by increasing the sequence length, the computational requirements of SW become the bottleneck.

Recently, the WFA algorithm has been proposed [19] which runs in $O(n \cdot s)$ time, proportional to the sequence length n and the error score s between sequences. To do so, the WFA uses a novel approach which only computes a reduced number of the DP-matrix cells to find the optimal alignment. With this approach, the WFA algorithm performs exact pairwise sequence alignment between the query and every potential candidate of the database, so its results are identical to the gapped SWG. Thus, the SWG algorithm of any full mapper could be replaced by the WFA algorithm to improve the mapper performance. Since the error score is typically much smaller than the sequence length, the WFA algorithm is significantly faster than other algorithms when aligning short reads. In addition, the WFA algorithm also scales much better with increasing sequence lengths, achieving $10-100\times$ speedups over other methods with long reads such as those produced by third generation sequencing systems.

This chapter presents the first FPGA-based accelerator for the WFA algorithm. In a hardware/software co-designed scheme, the FPGA accelerator computes the alignment of pairs of sequences and sends the output to the CPU. Then, multiple CPU threads process the output of the FPGA and produce the final results in parallel.

The FPGA design of the accelerator is composed of multiple Aligner cores that collaboratively compute the sequence alignments. Unlike the CPU implementation of the WFA, which supports any read length and computes any error score, the proposed design of the Aligners allows a configurable maximum read length and error score between the reads. These two design parameters determine the compute and memory resources required by each Aligner and, thus, the number of parallel Aligners that can be placed in the FPGA. By configuring the maximum read length and error score, the Aligners can be adapted to the characteristics of the reads generated by different sequencing machines and technologies. The source code of the WFA accelerator is open source and publicly available [207].

The main contributions of this chapter are:

- We propose the first accelerator for the WFA algorithm. The initial design of the proposed accelerator targets input sets composed of short reads and it performs the alignment of sequences in a hardware/software co-designed scheme. With this scheme, the compute intensive parts of the algorithm are done in the FPGA and the CPU gathers the output of the FPGA and computes the final results.

- We present key modifications to the initial design of the WFA accelerator to extend its functionality to long reads. To this end, we propose techniques to intelligently use the FPGA RAMs to store the sequences and the data structures of the WFA algorithm, and we also propose a re-organization of the tasks performed by the FPGA and by the CPU in the hardware/software co-design.

- We do a thorough evaluation of the WFA accelerator on a high performance system with a POWER9 CPU and two FPGAs. The evaluation uses synthetic input sets for short and long reads as well as real PacBio input sets, and it compares the performance and the power efficiency of the WFA accelerator against the CPU implementation of the WFA and of the SWG.

- For short reads, we demonstrate that the WFA accelerator achieves speedups of 4.5× to 8.8× with one FPGA compared to the WFA CPU-only implementation, and the speedups increase to 8.2× to 13.5× with two FPGAs. In addition, the energy-to-solution is reduced by 6.1× to 9.7× with one FPGA, and by 11.4× to 14.6× with two FPGAs.

- For long reads, we demonstrate that the WFA accelerator achieves speedups of 2.6× to 5.5× with one FPGA and of 2.7× to 9.9× with two FPGAs compared to the WFA CPU-only implementation. The energy-to-solution is also reduced by 3.6× to 7.5× with one FPGA, and by 3.7× to 10.9× with two FPGAs.

5.2 Background

The WFA [19] is an exact gap-affine-based pairwise alignment algorithm with identical results to the SWG algorithm. However, the WFA computes only a minimal number of cells of the DP-matrix to find the optimal alignment. This is done by proposing a different way of encoding the DP-matrix, as shown in Equation 5.1.

$$\begin{aligned}
\widetilde{I}_{s,k} &= \max \left\{ \begin{array}{ll} \widetilde{M}_{s-g_o-g_e,k-1} & \text{(Open Insertion)} \\ \widetilde{I}_{s-e,k-1} & \text{(Extend Insertion)} \end{array} \right\} + 1 \\
\widetilde{D}_{s,k} &= \max \left\{ \begin{array}{ll} \widetilde{M}_{s-g_o-g_e,k+1} & \text{(Open Deletion)} \\ \widetilde{D}_{s-g_e,k+1} & \text{(Extend Deletion)} \end{array} \right\} \\
\widetilde{M}_{s,k} &= \max \left\{ \begin{array}{ll} \widetilde{M}_{s-x,k}+1 & \text{(Substitution)} \\ \widetilde{I}_{s,k} & \text{(Insertion)} \\ \widetilde{D}_{s,k} & \text{(Deletion)} \end{array} \right\}
\end{aligned} \tag{5.1}$$

with initial condition $\widetilde{M}_{0,0} = 0$.

In Equation 5.1, $\{x, g_o, g_e\}$ are penalty scores (x: mismatch, g_o: gap-opening, g_e:gap-extension), a match penalty score is 0, s is the alignment error score or simply score, and k is the diagonal offset, which is 0 for main diagonal of the DP-matrix. It increases for diagonals on the right side of the main diagonal, and decrease (negative values) for diagonals on the left side of the main diagonal, depending the distance or offset of the diagonal with regard to the main diagonal. The WFA algorithm computes three wavefront vectors $\widetilde{M}_{s,k}$, $\widetilde{D}_{s,k}$ and $\widetilde{I}_{s,k}$ for each score, being k the index of their elements. The $\widetilde{M}$, $\widetilde{D}$ and $\widetilde{I}$ wavefront vectors, respectively, track alignments that end with a match/mismatch, a deletion or an insertion. The WFA encodes the diagonal cells, progressively as the score increases, from the left-most column to the farthermost cell that has score s. Hence, the wavefront vectors length increases as the score increases, and consequently it runs in $O(n \cdot s)$ time.

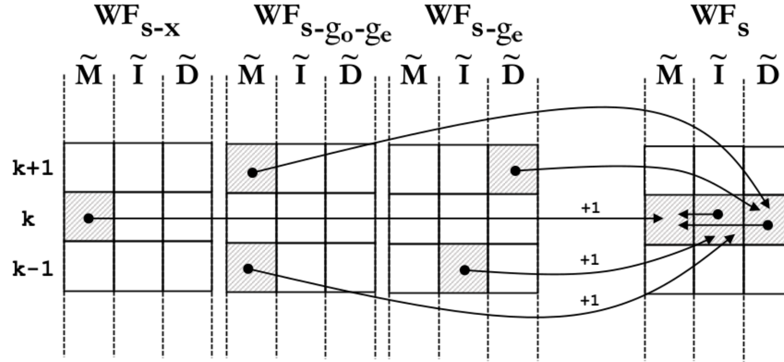


Figure 5.1: Dependencies between previous wavefronts to compute one element of the new wavefront [19].

Regarding Equation 5.1 wavefronts of a new score, WF_s, only depend on the previously calculated wavefronts of scores $s-x$, $s-g_o-g_e$ and $s-g_e$ (WF_{s-x}, $WF_{s-g_o-g_e}$ and WF_{s-g_e}). Figure 5.1 shows the dependencies between previously calculated wavefronts to compute one element of the new wavefront.

Figure 5.2 illustrates an example of aligning two sequences with the classical SWG using a DP-matrix (left) and its equivalent alignment with the WFA using wavefront vectors (right). The SWG computes all the cells in the matrix of Figure 5.2 (left), while the WFA only computes the colored cells. The WFA starts from score 0 and calculates only the cells which could have a score 0. Then it increases the score and calculates all possible cells with the new score. This process repeats until the end of alignment is reached, i.e., the calculation reaches cell (n,m), where n and m are sequences lengths. The positions of the cells with a specific score are kept in the wavefront vector of that score. For example as shown in Figure 5.2, for score 8, the vector $\widetilde{M}_8$ holds the offsets 2, 5 and 1 for the diagonals k=1, k=0 and k=-1, respectively. This represents that, in the diagonal 1 (k=1) of the matrix, the cell with offset 2 has a score of 8. Similarly, the cells with score 8 in diagonals 0 and -1 are at offsets 5 and 1, respectively. Note that, in the diagonal 0, the offsets 3, 4 and 5 have cells with score 8, and the WFA only stores the biggest offset (furthest-reaching point) of a score in a specific diagonal.

The WFA algorithm has two main operators to perform the alignment: $extend()$ and $compute()$. First, $extend()$ compares the sequences for each diagonal cell from starting positions i and j in the M DP-matrix until a mismatch is found and outputs the number of contiguous matching characters which are stored in $\widetilde{M}_s$. After extending all the cells of the $\widetilde{M}$ wavefront vector, the $compute()$ operator computes the offsets of the next $\widetilde{M}$, $\widetilde{D}$ and $\widetilde{I}$ wavefront vectors based on Equation 5.1.

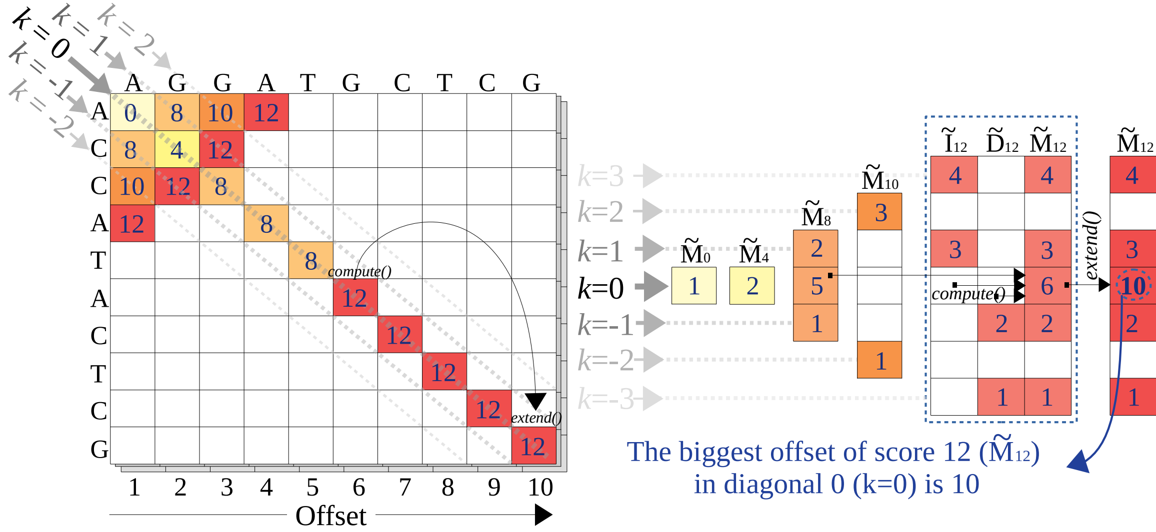


Figure 5.2: Alignment of two sequences using penalties $(x, g_o, g_e) = (4, 6, 2)$. Left) SWG DP-matrix highlighting the cells that are computed by the WFA. Right) Calculation of the necessary wavefront vectors by the WFA.

The WFA iteratively performs $extend()$ and $compute()$ until a wavefront, with score s, reaches the end of both sequences. So, the final alignment score is s. After that, to obtain the differences between the sequences, the $backtrace()$ operator is performed. This operator traces all the cells back from cell (m,n) to cell $(0,0)$, or in other words, from the cell that gave the optimal alignment score to the initial wavefront $\widetilde{M}_{0,0} = 0$. This is done by looking at the values that Equation 5.1 has generated for each cell towards the final alignment score.

The whole WFA algorithm, $extend()$, $compute()$, and $backtrace()$ algorithms are depicted, respectively, in Algorithms 1, 2, 3, and 4. As mentioned earlier, by increasing score the new wavefront grows by one element from both sides which is spanning over one more diagonal on each end. In Algorithms 2 and 3, hi and lo respectively indicate the highest and lowest elements or diagonals (k) that each type of wavefronts $(\widetilde{M}, \widetilde{D}, \widetilde{I})$ include for the score s.

In the reminder of this thesis we use the terms *error rate*, *error score* and *penalty score* (or simply *penalty*). The *error rate* refers to the accuracy of the sequencing machine. For example, a sequencing machine which has an error rate of 2% assures a DNA sequencing with 98% of accuracy, that is, two bases out of 100 may be sequenced incorrectly. The *error score* is the score that the alignment algorithm calculates when aligning a pair of sequences. The error score represents the degree of difference between two reads, and it indicates the cost of changing one read to make it identical to the other one. The error score is computed based on the *penalty score* (or simply *penalty*) used in the alignment algorithm, which is a numerical

Algorithm 1: Gap-affine WFA algorithm

Input: S_a, S_b strings, $p = \{x, g_o, g_e\}$ gap-affine penalties
Output: Gap-affine alignment $\mathscr{A}$ between S_a and S_b under p penalties

```
Function WF_ALIGN(S_a, S_b, p) begin
    // Diagonal and offset to (n,m)
    A_k ← (m − n)
    A_offset ← m
    // Initial conditions
    M~_{0,0} ← 0
    // Incremental computation of wavefronts
    s ← 0
    while true do
        // Exact extend s-wavefront
        WF_EXTEND(M~_s, S_a, S_b)
        // Check exit condition
        if (M~_{s,A_k} ≥ A_offset) then break
        // Compute wavefront for the next score
        s ← s + 1
        WF_COMPUTE(M~, I~, D~, S_a, S_b, s)
    // Backtrace alignment
    A ←WF_BACKTRACE(M~, I~, D~, S_a_Len, S_b_Len, s) return A
```

Algorithm 2: Wavefront extend

Input: $\widetilde{M}_s$ wavefront, S_a, S_b strings

```
Function WF_EXTEND(M~, S_a, S_b) begin
    for k ← M~^lo to M~^hi do
        i ← M~_{s,k} − k
        j ← M~_{s,k}
        while S_a[i] == S_b[j] do
            M~_{s,k} ← M~_{s,k} + 1
            i ← i + 1
            j ← j + 1
```

Algorithm 3: Compute next wavefront

Input: $\widetilde{M}, \widetilde{I}, \widetilde{D}$ wavefronts, s score

Function WF_COMPUTE($\widetilde{M}, \widetilde{I}, \widetilde{D}, s$) **begin**

$hi \leftarrow \max\{\widetilde{M}^{hi}_{s-x}, \widetilde{M}^{hi}_{s-g_o-g_e}, \widetilde{I}^{hi}_{s-g_e}, \widetilde{D}^{hi}_{s-g_e}\} + 1$

$lo \leftarrow \min\{\widetilde{M}^{lo}_{s-x}, \widetilde{M}^{lo}_{s-g_o-g_e}, \widetilde{I}^{lo}_{s-g_e}, \widetilde{D}^{lo}_{s-g_e}\} - 1$

for $k \leftarrow lo$ **to** hi **do**

$\widetilde{I}_{s,k} \leftarrow \max\{\widetilde{M}_{s-g_o-g_e,k-1}, \widetilde{I}_{s-g_e,k-1}\} + 1$

$\widetilde{D}_{s,k} \leftarrow \max\{\widetilde{M}_{s-g_o-g_e,k+1}, \widetilde{D}_{s-g_e,k+1}\}$

$\widetilde{M}_{s,k} \leftarrow \max\{\widetilde{M}_{s-x,k} + 1, \widetilde{I}_{s,k}, \widetilde{D}_{s,k}\}$

Algorithm 4: Compute backtrace

Input: $\widetilde{M},\widetilde{I},\widetilde{D}$ wavefronts, S_a_Len, S_b_Len strings lengths, s score

Function WF_BACKTRACE($\widetilde{M},\widetilde{I},\widetilde{D},S_a_Len,S_b_Len,s$) **begin**

```
    k ← S_a_Len − S_b_Len
    m ← S_a_Len + S_b_Len
    offset ← M~_{k,s}
    BT_TYPE ← Mismatch
    i ← offset − k
    j ← offset
    while (i > 0 and j > 0 and s > 0) do
        // find the origin and its corresponding difference
        (max, Difference) ←
          max_difference_type(M~_{k+1,s−g_o−g_e}, M~_{k,s−x} + 1, M~_{k−1,s−g_o−g_e} + 1, I~_{k−1,s−g_e} + 1, D~_{k+1,s−g_e})
        if (BT_TYPE == Mismatch) then
            for n ← 0 to offset−max do
                A[m−−] ← "M"
            offset ← max
        switch Difference do
            case Deletion_extension: do
                s ← s − g_e , k ← k + 1, A[m−−] ← "D", BT_TYPE ← Deletion
            case Deletion_opening: do
                s ← s − g_o − g_e, k ← k + 1, A[m−−] ← "D", BT_TYPE ← Mismatch
            case Insertion_extension: do
                s ← s − g_e , k ← k − 1, A[m−−] ← "I", offset ← offset − 1, BT_TYPE ← Insertion
            case Deletion_opening: do
                s ← s − g_o − g_e, k ← k − 1, A[m−−] ← "I", offset ← offset − 1,
                  BT_TYPE ← Mismatch
            case Mismatch: do
                s ← s − x, A[m−−] ← "X", offset ← offset − 1, BT_TYPE ← Mismatch
        i ← offset − k
        j ← offset
    if s == 0 then
        for n ← 0 to offset−max do
            A[m−−] ← "M"
    else
        while i > 0 do
            A[m−−] ← "D", i ← i − 1
        while j > 0 do
            A[m−−] ← "I", j ← j − 1
```

S1 [100] = "AACCTG………...AAACTTTG"
S2 [100] = "AACGTG………...AAACTT_G"

Defined *penalties*: x = 4; g_o = 6; g_e=2

Error Rate = 2% (2 errors in 100 bases)
Error Score = 1x4 + 1x(6+2) = 12

Figure 5.3: Example of error rate, error score, and penalties.

value that represents the cost of each difference (mismatch, gap-opening and gap-extension) between two reads. The example in Figure 5.3 clarifies these terms.

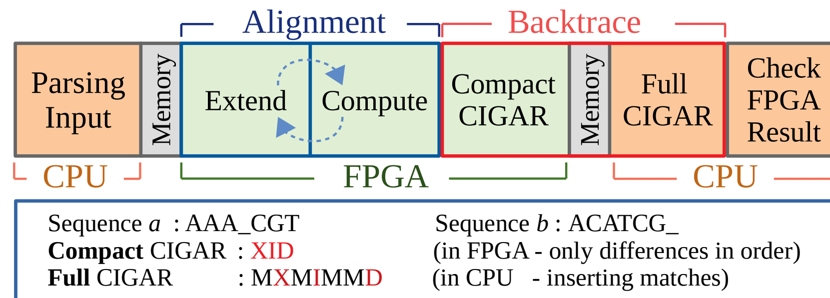


Figure 5.4: Steps in the WFA co-designed accelerator of short reads and example of the compact and full CIGAR. The compact CIGAR is computed in the FPGA and only returns differences between sequences. Then, the CPU recovers the full CIGAR inserting matches by comparing both sequences in the CPU.

5.3 WFA Accelerator for Short Reads

This section presents the proposed WFA accelerator for short reads. Figure 5.4 shows the co-design steps. First, the CPU parses the input data and stores them in the memory. Then, the FPGA reads the sequences, computes the alignments by iteratively performing the extend and compute operations, and writes the results to the memory in compact CIGAR form. After that, multiple CPU threads read and check the FPGA alignment results and finish the backtrace step by unpacking the compact CIGARs to full CIGARs.

In our experiments with the reference WFA CPU-only implementation of the WFA algorithm [200], the extend, compute and backtrace steps are responsible for around 50%, 45% and 5% of the total execution time, respectively. Hence, offloading the extend and compute steps to the FPGA is crucial to accelerate the algorithm. In addition, offloading the backtrace step to the FPGA is also beneficial because, although this step has a small weight on the total execution time, it requires reading the whole data of all the wavefronts. Thus, to minimize bandwidth-bound data transfers between the memory and the FPGA, the presented accelerator innovatively divides the backtrace operation into two parts, one in the FPGA that computes the CIGARs in a compacted form of only eight bytes, and one in the CPU that unpacks the compact CIGARs and generates the full CIGARs. An example of a compact and a full CIGAR is shown in Figure 5.4.

The FPGA design is composed of three main modules, as shown in Figure 5.5. The *Aligner* module implements the main computational steps of the WFA algorithm. A configurable number of Aligners can be instantiated in the design so they process alignments in parallel. The

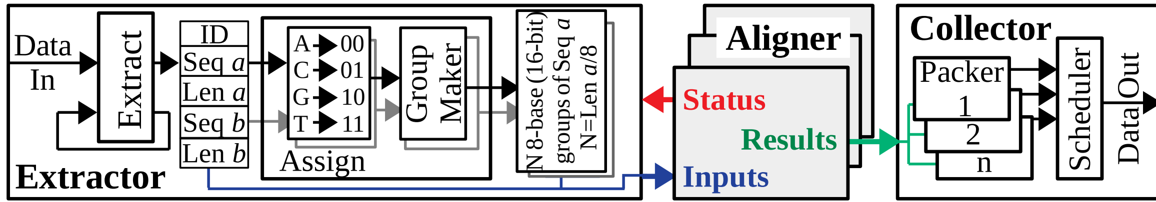


Figure 5.5: Structure and different modules of the FPGA design of the WFA accelerator for short reads.

Extractor module distributes sequences among the Aligners and the *Collector* module gathers results from them.

5.3.1 Extractor Module

The Extractor module distributes sequences among Aligners. This module has two states, extract and assign. The first state reads data from memory and extracts the DNA sequences, their alignment IDs, and their lengths. The length of the input sequences is fixed in the Extractor module at design time. The second state compresses the sequences by mapping each base to two bits and sequentially packs them in groups of eight bases. Blocks of eight bases (16 bits) of each sequence are stored in an array of 16-bit registers. When an Aligner becomes idle, the Extractor module assigns to it one pair of sequences along with their lengths and the alignment ID, in order to process the alignment.

5.3.2 Collector Module

The Collector module collects the results of the Aligners and writes them to the memory. The result of each Aligner is 16 bytes, while the FPGA data width is 128 bytes in our setup. Thus, the Collector module first packs eight results of each Aligner in one 128-byte word and then writes it to the memory. A *Scheduler* handles the order in which the results of the different Aligners are written to the memory.

5.3.3 Aligner Module

The Aligner module computes the sequences alignment. To do so, each Aligner contains a configurable number of *Extend* and *Compute* sub-modules, a *Backtrace* sub-module, and a *Controller* sub-module that controls the operations and the data flow. The Extend and Compute sub-modules operate sequentially multiple times, as the output of one is the input of the other

one. At the end, the Backtrace sub-module computes the compact CIGAR. Although these steps are executed sequentially, they are pipelined and internally parallelized.

Aligner Parallel Structure

The parallel operation of the Aligners is achieved by dividing a *wavefront matrix* in independent parts. A wavefront matrix, as shown in Figure 5.6 (a), is a structure that unifies all the wavefront vectors of a given type ($\widetilde{I}$, $\widetilde{D}$ and $\widetilde{M}$). The values stored in the cells of the matrix are called *offsets*. The X axis of the wavefront matrix represents the scores, and each column of the matrix stores the wavefront vector for the corresponding score (i.e, the column 0 of the wavefront matrix $\widetilde{M}$ stores the wavefront vector $\widetilde{M}_0$). The Y axis is defined at compile time with a parameter called k as in Equation 5.1, which limits the maximum supported error score between sequences and sets the range of the Y axis from $-k$ to k. As explained in Section 5.2, this parameter defines the maximum number of diagonals that a wavefront vector can include, or in other words, the maximum number of elements of a wavefront vector. Since the length of the wavefront vectors increases with the score, some cells of the wavefront matrix are invalid. In the example of Figure 5.6 (a), valid cells are marked with an *X*. In addition, the whole cells of some columns, depending on the penalties, are not valid. Therefore, each column has a *Null* tag that indicates if all the cells of that column are invalid. In the design, invalids cells hold a negative value.

The proposed accelerator exploits parallelism by computing multiple offsets of a column of a wavefront matrix at the same time. For a given score, the corresponding columns offsets in the three wavefront matrices can be calculated in parallel, as they only depend on the previously calculated columns of the wavefront matrices (see Equation 5.1 and Figure 5.1). We define *window* as the set of columns of a wavefront matrix that are needed to compute a given column. The rightmost column of a window, called *frame column*, is the one being processed, and the other columns of the window are needed as inputs to compute the frame column. The width of the windows depends on the penalty scores. For typical penalties $(x, g_o, g_e) = (4, 6, 2)$, the computation of the frame column requires 2, 2 and 8 previous columns of the $\widetilde{I}$, $\widetilde{D}$, and $\widetilde{M}$ wavefront matrices, respectively. When all the offsets of a frame column are calculated, all the columns of the window are shifted to the left and the leftmost one is discarded. This allows to reduce the FPGA resource utilization, as we only need to keep a limited number of columns of the wavefront matrices. The windows of the wavefront matrices are implemented as 2D-arrays of registers to provide concurrent and fast access to the cells. The width of the registers depends on the sequence length.

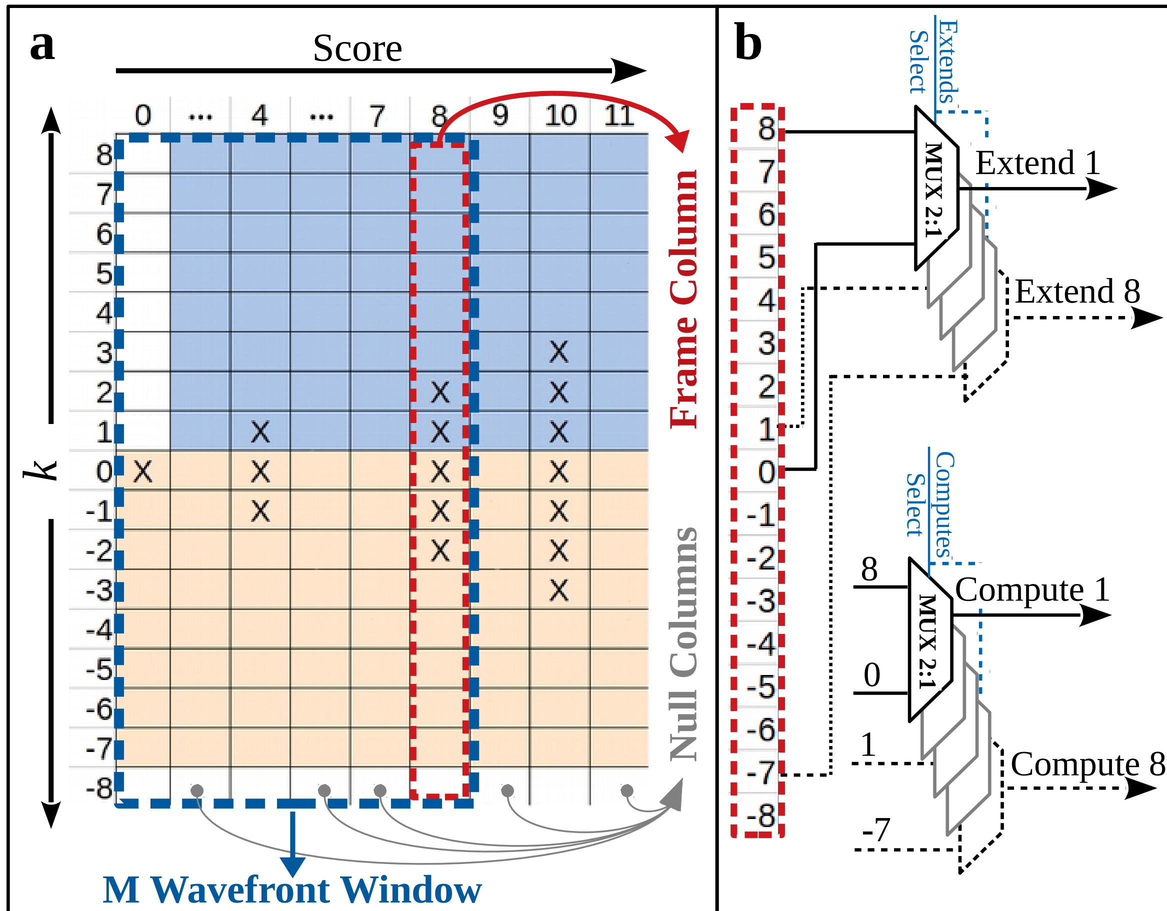


Figure 5.6: a) An example WFA wavefront matrix with $k = 8$. Valid cells for penalties $(x, g_o, g_e) = (4, 6, 2)$ are marked with an X. Same colored cells of each column are the parallel inputs of the Extend and Compute modules at each clock cycle. The appropriate Extend and Compute inputs are selected using multiplexers shown in (b).

To compute a frame column, each of its cells is fed to a configurable number of pairs of Extend and Compute sub-modules, called *parallel sections*. In the example of Figure 5.6 there are eight parallel sections which compute eight cells of the frame column in parallel. Note that the number of parallel sections can be lower than the number of cells in a frame column, so the computation of the frame column can take several cycles. To efficiently use the FPGA resources, we restrict the possible inputs of the multiplexers that pass the offsets of the frame column to the Extend and Compute sub-modules. In the example of Figure 5.6 (b), the offsets of cells 8 and 0 are inputs of the first Extend and Compute sub-modules, cells 7 and -1 are inputs of the second sub-modules, and so on. In Figure 5.6 (a) the colors represent the cycle in which each cell is processed. In the example, the frame column of score 8 has 5 valid cells, and the computation of the cells in rows $k = 1$ and $k = 2$ is performed in the first cycle, while

the computation of the cells in rows $k = -2$ to $k = 0$ are performed in the second cycle. We pipeline the processing of frame columns that require more than one cycle by first performing the compute for eight cells and, while these eight cells are extended, the next eight cells are computed.

The next subsections explain in detail the architecture of the Aligner module and its sub-modules, shown in Figure 5.7.

Extend Sub-module

The Extend sub-module receives the offset of a cell, its k position and a start signal. From these inputs, the Extend sub-module calculates the initial positions in sequence a and sequence b based on Equation 5.2, compares the bases of both sequences starting from the initial positions until a mismatch is found, and returns the number of matching bases.

$$\begin{cases} \text{Starting position of sequence } a = offset_in - k \\ \text{Starting position of sequence } b = offset_in \end{cases} \tag{5.2}$$

To increase the speed of the design and minimize resources, the sequences are compared in blocks of eight bases. To do so, the Extractor module packs blocks of eight bases in an array of registers. However, each base of sequence a can be compared with any base of sequence b, and their positions may not be at the boundaries of the blocks of eight bases. For this reason, two blocks of eight bases of each sequence are passed to the Extend sub-module, which uses two multiplexers for each sequence to select the eight bases that need to be compared. The selected 16 bases of each sequence are then concatenated and passed to a shift register that aligns them to the comparator input. The design is pipelined in such a way that the comparator compares eight bases of the sequences at each clock cycle.

The extend operation continues until a mismatch is found. Then the Extend sub-module returns the number of matches and the new offset for the cell. The Extend sub-module may receive a negative (invalid) input. For example in Figure 5.6, the inputs of Extend sub-modules 1 to 6, when calculating offsets of score 8 in first cycle, are invalid. In such cases, the starting position of at least sequence b will be negative (Equation 5.2) and hence, the number of matches will be zero. Therefore, the output offset will be equal to the input offset. The new offsets are stored in the rightmost column of the $\widetilde{M}$ wavefront window. After extending a column, if the alignment has not reached the end of the sequences and k has not reached the maximum value,

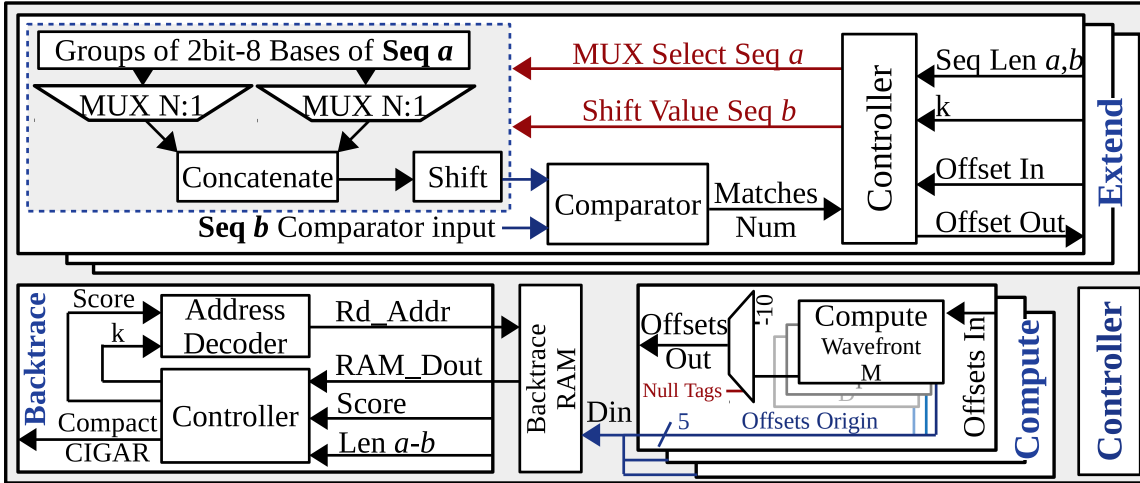


Figure 5.7: Architecture of the Aligner module and its sub-modules in the FPGA design of the WFA accelerator for short reads.

the Controller moves the window to the right, increases the score, and extends k by one from both ends.

Compute Sub-module

After the extend step, the compute step determines the new offset of a cell of each wavefront frame column by comparing some of the previously calculated offsets of previous columns, as described in Equation 5.1.

The Compute sub-module is also in charge of managing the Null tags of the columns of the wavefront matrices. The Null tag of the frame column is determined by the Null tags of the input columns that are used to compute the frame column. If the Null tag of any of the input columns is not set, the Null tag of the frame column is set to zero and the offsets are computed normally. Otherwise, the Null tag of the frame column is set to one and a negative value is returned. Then the upcoming extend operation is skipped because the extend of an invalid offset always returns the same offset.

This sub-module also tracks the origin of each computed cell in a *backtrace RAM*, as the backtrace step requires this information. Clearly the origins of the invalid columns are not useful. Hence, to minimize the depth of the backtrace RAM, the Compute sub-module only writes the origins of cells of columns with a Null flag of zero. As shown in Equation 5.1 and Figure 5.1, the origin of a cell in the $\widetilde{I}$, $\widetilde{D}$, and $\widetilde{M}$ wavefronts matrices can come from 2, 2 and 5 positions, respectively, so we need 1, 1 and 3 bits to store them. At the end of the compute step, the origins of the computed cells are concatenated in five bits. Since in our design of short

reads eight Compute sub-modules process eight cells in parallel, the width of the backtrace RAM is 40 bits, and a depth of 250 words is needed to support k values of up to ± 32. The write address of the backtrace RAM is controlled by the Controller sub-module of the Aligner.

Backtrace Sub-module

At the end of alignment, the backtrace determines the mismatches, insertions and deletions that have to be applied to one sequence to make it identical to the other sequence. For this step we propose a novel hardware/software co-designed technique that reduces the amount of memory required by the algorithm and avoids doing a traditional memory-bound backtrace.

On the FPGA side, the *Controller* of the Backtrace sub-module receives the final score, the difference in the length of sequences, and the last written data in the backtrace RAM. With these values it calculates the new k and score and it passes them to the *Address Decoder* to find the backtrace RAM addresses of the previous location of the last cell. As the Backtrace sub-module reads the output data of the backtrace RAM, it decodes the backtrace data and updates the compact CIGAR register by the corresponding difference, i.e, mismatch, insertion-opening, deletion-opening or extension. Each of these differences are reflected by two bits in the compact CIGAR register. The process of reading backtrace data, updating CIGAR, calculating new k and score, and decoding next addresses, is iteratively repeated until the calculated score becomes zero and the backtrace is completed. Then, an 8-byte backtrace in compact CIGAR form is sent to the CPU along with the alignment ID.

The CPU then traverses the two sequences to unpack the backtrace in full CIGAR format. If an extension appears after an insertion-opening, the CPU interprets it as an insertion-extension. The same is true for the deletion-extension. CPU threads can recover different backtraces in parallel, as they are completely independent processes.

5.4 WFA Accelerator for Long Reads

This section presents the extensions to the WFA accelerator to make it suitable for long reads. Long reads are those with a length of more than 1K bases. Unlike short reads, long reads cannot be stored in FPGA registers due to the huge LUT utilization and the complications in the routing. Moreover, as the read length increases, the error rate of sequencing technologies also increases. Thus, in the accelerator for long reads, the dimensions of the wavefront matrix and the parameters k and score (Figure 5.6) are significantly larger than in the one for short reads. In addition, as the wavefront matrix expands, the number of parallel Extend and Compute

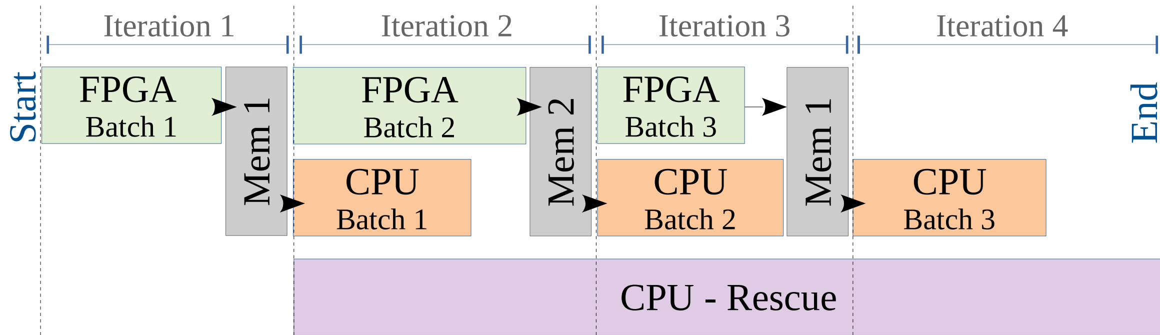


Figure 5.8: Co-design steps in the WFA accelerator for long reads. The FPGA aligns reads in batches while the CPU checks the results and performs the backtrace. The CPU Rescue computes the alignments that the FPGA has failed to compute.

sub-modules, i.e., parallel sections, should increase to improve performance. Lastly, the higher error rates require bigger RAMs to store the backtrace data.

The WFA accelerator for short reads presented in the previous sections of this chapter does not scale to long reads. We have tried to configure the WFA accelerator for short reads with Aligners using eight parallel sections for an input set with read lengths of 1K and error rates of 5%. In this step we have observed that, in our setup, the resulting FPGA design can only fit two Aligners inside the FPGA. As a result, it offers less performance than the multi-threaded execution of the WFA implementation for CPUs. We have also tried to scale the WFA accelerator for short reads to a read length of 10K and an error rate of 10%. However, in this step the resulting FPGA design cannot even fit a single Aligner in the FPGA. To overcome these scalability problems, we propose the following modifications and extensions to the design of the accelerator so that it is applicable to long reads:

1 Place the input sequences in RAMs instead of registers.

2 Place the wavefront matrices in RAMs instead of registers.

3 Move the backtrace computation of the compact CIGAR to the CPU.

4 Re-structure the entire hardware/software co-design to adapt it to the new changes.

5.4.1 Hardware/Software Co-design Structure

Figure 5.8 shows the hardware/software co-design structure of the WFA accelerator for long reads. In such co-design, the FPGA does the alignment in batches and sends the backtrace data to the CPU. The CPU checks the results received from the FPGA, separates the backtrace data

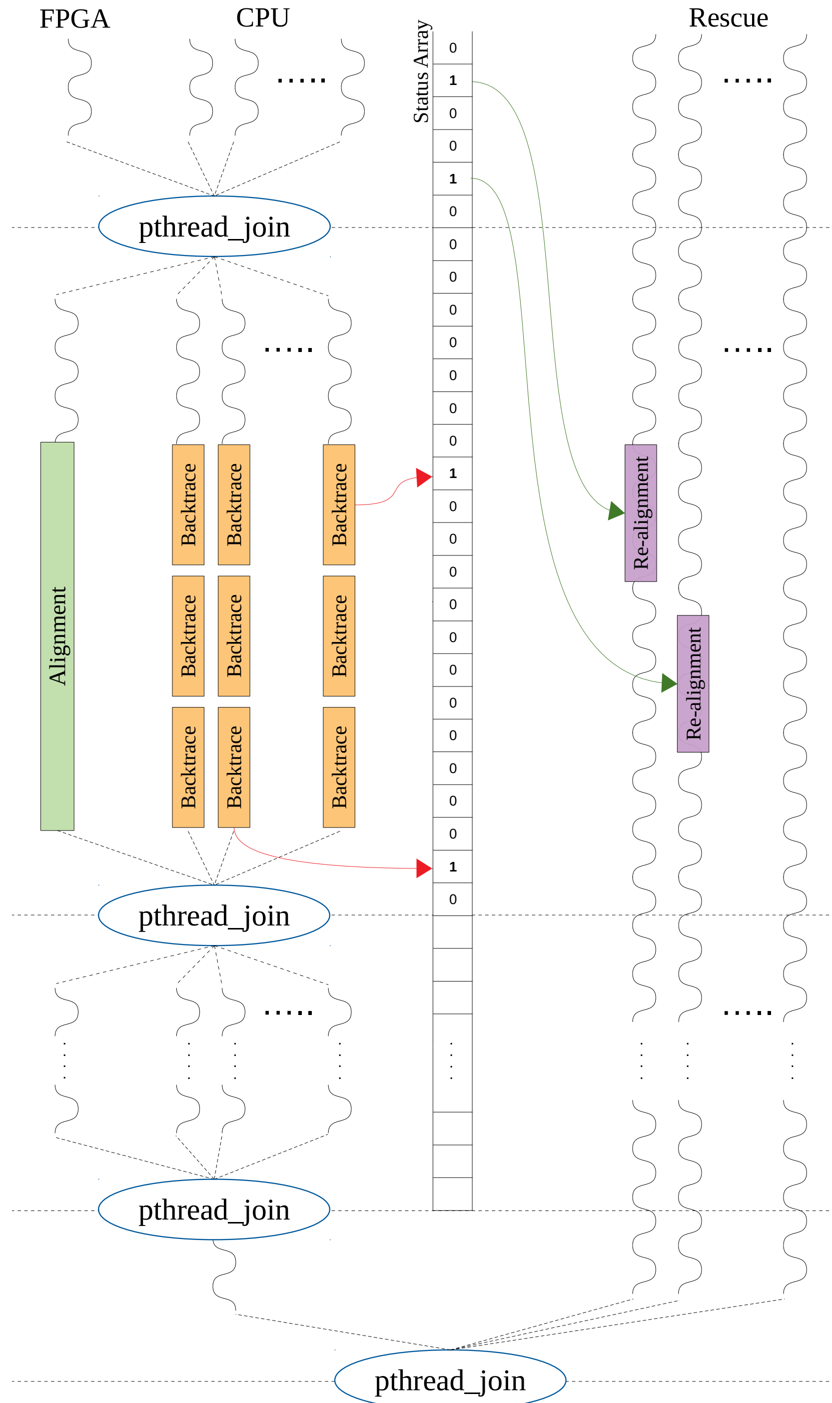


Figure 5.9: Synchronization of different threads.

of each sequence, performs the backtrace to obtain the compact CIGARs (this is done in the FPGA in the WFA accelerator for short reads), and then it recovers the full CIGARs from the compact CIGARs (Figure 5.4). In addition, the CPU also performs the *Rescue*, which aligns reads that the FPGA is not able to align. Such reads are those with lengths or alignment scores higher than what is supported by the FPGA design.

The hardware/software co-designed scheme processes the alignments in batches and uses double buffering to overlaps the work done in the CPU and the FPGA. The batch processing also allows to control the memory requirements, which are much larger in the WFA accelerator for long reads compared to the one for short reads. The batch size is configured by the user. Based on the characteristics of the input, the user can estimate the maximum memory required to store the backtrace data. The amount of memory required is approximately $k^2 \times 5$ bits for each pair of reads. For example, given an input set with read lengths of 10K bases and error rates of 10% that uses k=4K, around 10MB of memory is needed to store the backtrace data of the alignment of a pair of reads. Thus, the user can select the most appropriate batch size based on the estimated size of the backtrace data, the number of reads, and the memory size of the system. In addition, selecting an appropriate batch size is also important for achieving a balanced CPU-FPGA workload.

Each task involving the FPGA, CPU, and Rescue in Figure 5.8 is executed using separate threads. The synchronization among them is illustrated in Figure 5.9. Initially, a single FPGA thread is active, responsible for initiating the alignment of the first batch of inputs. Following the first iteration, the Rescue threads are launched, and they remain active throughout the whole execution. From the second to the penultimate iterations, both the single FPGA thread and the CPU threads run concurrently within a for loop. The CPU threads are spawned at each iteration using an OpenMP for clause that traverses the backtrace data generated in the previous iteration. At the end of each iteration of the loop, thread_join is used to ensure synchronization between the FPGA thread and the CPU threads. In the final iteration, outside the loop, only the CPU threads are activated to perform the backtrace of the last batch of reads.

We use two global variables to synchronize the CPU threads and the Rescue threads. One variable is an array that holds the alignment status of all pairs of sequences in the input set, and the other variable is an integer indicating the last processed ID by the CPU threads. Each ID corresponds to a pair of sequences in the input set. When a CPU thread detects an unaligned ID, it sets the corresponding index in the status array to one. The last processed ID integer is updated at the end of each iteration. Rescue threads are assigned a portion of the status array using an OpenMP for clause and, when they encounter a one in an element of the array, they

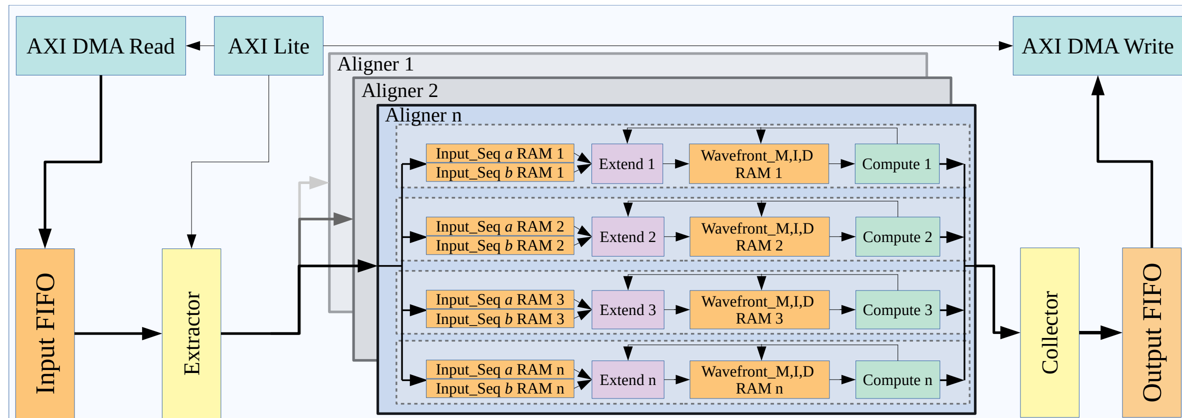


Figure 5.10: Structure and different modules of the FPGA design of the WFA accelerator for long reads.

perform the rescue alignment for that ID. The Rescue threads can advance in the array only up to the index indicated by the last processed ID specified in the synchronization variable. The entire application is considered finished when the Rescue threads have traversed all the indexes of the status array and realigned all the unaligned IDs.

At the beginning of the alignment, all the input reads are read from disk by the CPU. Then, as shown in Figure 5.8, the FPGA reads the first batch of reads, processes the alignments, and stores the results in a memory partition of the CPU memory (Mem 1). Then, multiple CPU threads read the results from Mem 1, check the results, and perform the backtrace. Note that the results can contain alignments that the FPGA has failed to compute because the read lengths or the error of the alignment are higher than what is supported by the Aligners. The result of each alignment includes a Success flag that is set to zero if the FPGA cannot compute it. When the CPU checks the results, the alignments that have the Success flag unset are sent to the Rescue. The Rescue aligns the reads using the CPU implementation of the WFA, and it is executed by multiple CPU threads in parallel with the FPGA and the main CPU block.

Figure 5.10 shows the structure of the FPGA design of the WFA accelerator for long reads. The layout is very similar to the one for short reads. The Extractor module extracts the sequences and distributes them among different Aligners. The results of the Aligners are collected by the Collector module, which sends the results to the CPU. The following subsections explain the modifications and extensions introduced in the design of each module, compared to the design for short reads.

5.4.2 Extractor Module

The Extractor module of the WFA accelerator for long reads is more generic than that of the short reads. The length of the reads is fixed by design in the accelerator for short reads. In contrast, the Extractor of the accelerator for long reads defines a configurable maximum read length, and it can process reads with any length as long as they do not surpass the maximum length. To do so, the CPU defines a *MAX_READ_LEN* for the input set and sends it to the FPGA via the AXI-Lite bus in Figure 5.10. The MAX_READ_LEN must be divisible by the data width of the FPGA (128 bytes in our setup). For example, if the longest sequence in the input set has a length of 9010 bases, the MAX_READ_LEN is set to 9088 bases and the extra 78 bases are filled by dummy bases in the CPU. Dummy base padding is applied to all the sequences of the input set, and the Extractor module ignores the dummy bases when it reads them.

The Extractor module monitors the activity of the Aligner modules and, when one of them becomes idle, it starts extracting data and passing it to the idle Aligner. For a pair of sequences, the Extractor module spends three initial clock cycles in reading the ID of the alignment and the length of the two sequences to be aligned. Then, at each clock cycle, it reads 16 bytes (i.e., 16 bases) from each sequence until it reaches the end of the sequences. When reading bases, the Extractor module maps them to two bits, so the blocks of 16 bytes are converted to 32 bits. In this phase, the Extractor module is also in charge of detecting two types of unsupported reads: those with a length longer than MAX_READ_LEN, and those including ’N’ (unknown) bases. If an unsupported read is found, the Extractor module signals the corresponding Aligner in order for the latter to ignore the inputs and not process the alignment. Then it sets the Success flag of the alignment to zero.

When the Extractor reads valid bases, these are stored in the input RAMs of the Aligner. The two sequences (*a* and *b*) are stored in separate RAMs (see Figure 5.10), since parallel accesses to the two sequences are required during the processing of the alignment. In addition, the sequences are replicated multiple times, one per each set of Extend and Compute sub-modules. Each sequence is stored in its input RAMs using the following format: alignment ID (4 bytes), sequence length (4 bytes), sequence bases (MAX_READ_LEN bytes).

5.4.3 Aligner Module

The Aligner module is responsible for computing the sequence alignment. Each Aligner contains a configurable number of Extend and Compute sub-modules. Figure 5.10 shows an

example with four Extend and Compute sub-modules working in parallel in different parallel sections. Each parallel section processes one cell of the wavefront matrix and has its own independent resources, i.e., the Extend and Compute sub-modules, the input RAMs, and the wavefront RAMs. As the Extend and Compute sub-modules work in parallel, they need parallel access to the input sequences and to the wavefront matrices. Hence, in each Aligner the sequences are replicated in different number of RAMs, one per each parallel section. In contrast, the wavefront matrices are distributed among the wavefront RAMs. As in the design for short reads, the Extend and Compute sub-modules operate sequentially until the end of the alignment, as the output of one is the input of the other one. However, the Extend and Compute sub-modules are internally parallelized and their operation is pipelined.

Aligner Parallel Structure

In the WFA accelerator for long reads, the wavefront windows (Section 5.3.3 and Figure 5.6) are stored in wavefront RAMs instead of 2D-arrays of registers. In addition, unlike in the accelerator for short reads, in the accelerator for long reads the Null columns are not stored in the wavefront windows, so the $\widetilde{M}$, $\widetilde{I}$ and $\widetilde{D}$ wavefronts windows only contain, respectively, four, one and one columns of previous data and one frame column (for typical penalties $(x, g_o, g_e) = (4, 6, 2)$). Figure 5.11 shows an example of how the $\widetilde{M}$ wavefront window is implemented as a matrix (left) and how it is mapped to a series of wavefront RAMs in the accelerator for long reads (right). Each cell contains the coordinates of the matrix to ease the understanding of how each position of the matrix is mapped to the wavefront RAMs. In the design for long reads, after calculating each frame column, we only shift the *tags* of the columns to the right, and the right-most one to column 0. In the example of Figure 5.11 (left), after calculating the cells of the frame column which is tagged as ***Score***, the columns from left to right will be tagged as ***Score*** (frame column), ***S***-8, ***S***-6, ***S***-4 and ***S***-2.

Cells with the same color in Figure 5.11 are processed at the same time, so parallel writes to the cells with the same colors of the frame column are required. In addition, processing the cells of the frame column requires reading data of the cells from previous columns (according to Equation 5.1), so parallel reads of those cells are also required. Thus, naively mapping the wavefront window to a single RAM would lead to serialized accesses to the different elements of the window required to compute a given cell of the frame column. For this reason, the proposed design distributes the wavefront window among multiple RAMs to enable parallel accesses to the elements required in the computation. As shown in Figure 5.11, the data in

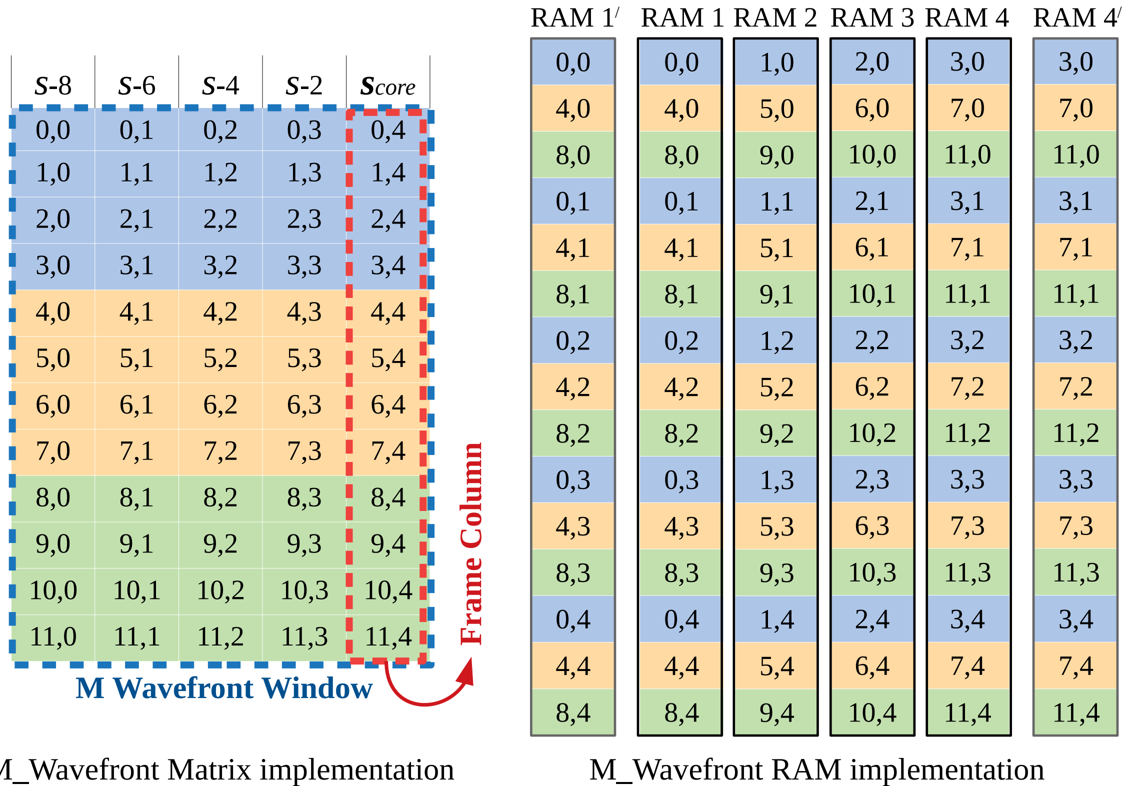


Figure 5.11: Implementation of the $\widetilde{M}$ wavefront matrix using RAMs in the WFA accelerator for long reads.

the RAMs is structured in a way that the cells of each column that have the same color are accessible in parallel. Each RAM has one read and one write port.

The number of RAMs required to store the wavefront window depends on the number of parallel sections, as it determines the number of cells with the same color that are processed in parallel. Following the example with four parallel sections, four RAMs are required for each wavefront window. In addition, in the $\widetilde{M}$ wavefront window we duplicate the first and the last RAMs (labeled RAM $1'$ and RAM $4'$ in Figure 5.11), as discussed in the following paragraphs. These duplicated RAMs are not required in the $\widetilde{I}$ and $\widetilde{D}$ wavefronts windows. The width of the wavefront RAMs is equal to the number of bits required to store ±MAX_READ_LEN (i.e. 15 bits for a sequence length of 10K), and their depth is equal to number of cells of the wavefront window divided by the number of parallel sections, as the wavefront window is distributed among multiple wavefront RAMs.

Figure 5.12 shows the cells of the $\widetilde{M}$ wavefront window that are read to compute the values of cells (4,4), (5,4), (6,4) and (7,4) of the frame column. The figure shows how these elements

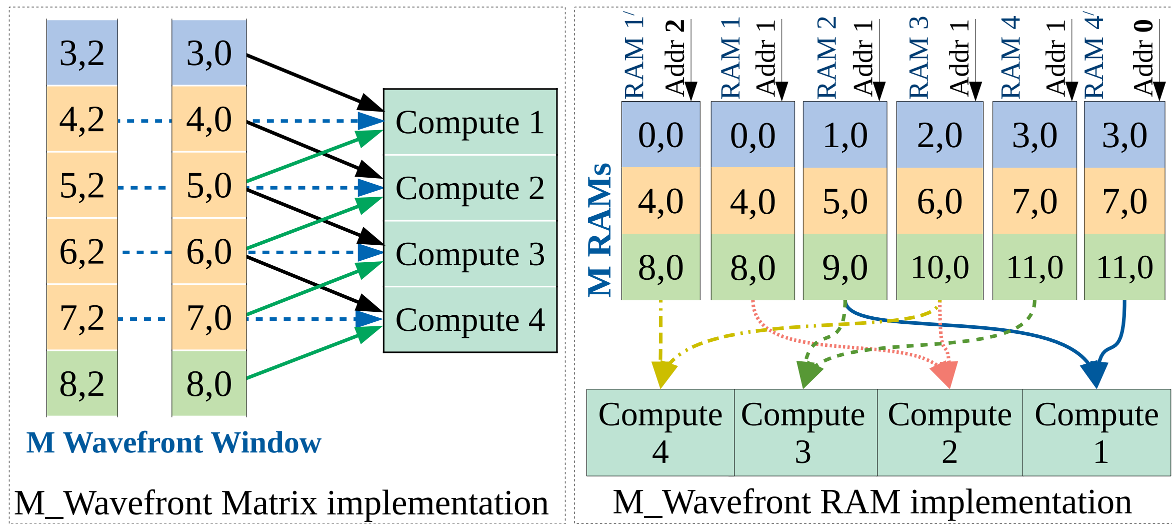


Figure 5.12: Data required by each Compute sub-module and how it is accessed as cells of a matrix (left) or words of a RAM (right). Two accesses in two consecutive clock cycles are required to read the data required by the Compute sub-modules. For simplicity, the right figure only shows connections and addresses for the first access.

are accessed when the window is implemented with registers in the design for short reads (left) and when the window is implemented using RAMs in the design for long reads (right).

The left part of Figure 5.12 shows that each Compute sub-module requires three accesses to the data of $\widetilde{M}$ wavefront window, one for reading data in column 2 and two for reading data in column 0. In addition, it can be observed that the second access to column 0 is only needed for the first and the last cells of each parallel section.

The right part of Figure 5.12 shows that replicating RAM 1 and RAM 4 allows reading all the data required from column 0 in a single clock cycle. For simplicity, the right part of Figure 5.12 only shows the RAM accesses to the data of column 0, as the access to column 2 is trivial. It can be observed that, apart from the elements (4,0), (5,0), (6,0) and (7,0) which are distributed among RAMs, two other elements from column 0 are required: element (3,0), which is mapped in RAM 4 and is required by Compute 1, and element (8,0), which is mapped to RAM 1 and is required by Compute 4. Thus, replicating RAMs 1 and 4 in RAMs $1'$ and $4'$ allows parallel access to all the required elements. In particular, element (4,0) is read from address 1 in RAM 1 while element (8,0) is read from address 2 in RAM $1'$. Similarly, element (7,0) is read from address 1 in RAM 4 while element (3,0) is read from address 0 in RAM $4'$. Therefore, two accesses to the wavefront RAMs (one for column 0 and one for column 2) are needed to provide the data of the $\widetilde{M}$ wavefronts to the Compute sub-modules.

The Compute sub-modules also require data from the $\widetilde{I}$ and the $\widetilde{D}$ wavefronts. To reduce the RAM utilization we merge the data of the $\widetilde{I}$ and the $\widetilde{D}$ wavefronts in a single RAM and access the data of these wavefronts in two accesses. For example, in one of our designs that supports read lengths of up to 10K bases and 10% error rate, the RAM width required to store the $\widetilde{I}$ and $\widetilde{D}$ wavefronts is 15 bits each, and the required RAM depth is 256. In our setup, the minimum size of the FPGA physical block RAMs is 18Kb and they can be configured as 4K×4, 2K×9, 1K×18, and 512×36 [208]. Thus, if we used the 1K×18 block RAM configuration for the $\widetilde{I}$ and $\widetilde{D}$ wavefronts RAMs, we would need two physical block RAMs, one for the $\widetilde{I}$ and one for the $\widetilde{D}$ wavefronts RAM. Instead, by concatenating the $\widetilde{I}$ and $\widetilde{D}$ wavefronts into a single RAM, we require a RAM width of 30 bits and a RAM depth of 256, so we can use a single physical block RAM with the 512×36 configuration to store both the $\widetilde{I}$ and the $\widetilde{D}$ wavefront RAMs. The downside of this optimization is that the accesses to the $\widetilde{I}$ and $\widetilde{D}$ wavefronts have to serialized. However, this does not cause any performance penalty, as the two accesses to the $\widetilde{I}$ and $\widetilde{D}$ wavefronts required to compute a cell are done in parallel to the two accesses to the $\widetilde{M}$ wavefront.

Figure 5.13 illustrates the parallel structure of an Aligner of the accelerator for long reads. The Extend and Compute sub-modules work in parallel in a pipelined fashion. For the sake of simplicity, each Extend and Compute sub-module is connected to the same index of the wavefront RAMs. All the Extend sub-modules have their own copy of the sequences a and b in separate input RAMs. Each Extend sub-module performs the extend operation and writes the result in the corresponding cell of its $\widetilde{M}$ wavefront RAM. Then, the Compute sub-modules read data of the $\widetilde{M}$, $\widetilde{I}$ and $\widetilde{D}$ wavefronts and compute the cells. The results of the $\widetilde{I}$ and $\widetilde{D}$ wavefronts are directly written in the $\widetilde{I}/\widetilde{D}$ wavefront RAMs, and the results of the $\widetilde{M}$ wavefront are passed to the Extend sub-modules, which perform the extend operation and write the results in the $\widetilde{M}$ wavefront RAMs. This process is repeated until the end of alignment is reached.

Extend Sub-module

The Extend sub-module compares two sequences from different starting positions and outputs the number of matching characters. The number of Extend sub-modules is as many as parallel sections. Each Extend has a comparator and the size of its inputs is equal to the width of the input RAMs (32 bits in the accelerator for long reads, which means comparing 16 bases at the same time). Unlike in the accelerator for short reads, where each Extend has fast and direct access to the sequences, in the accelerator for long reads each Extend sends read requests to the input RAMs and receives 16 bases at each clock cycle. Similar to the accelerator for short

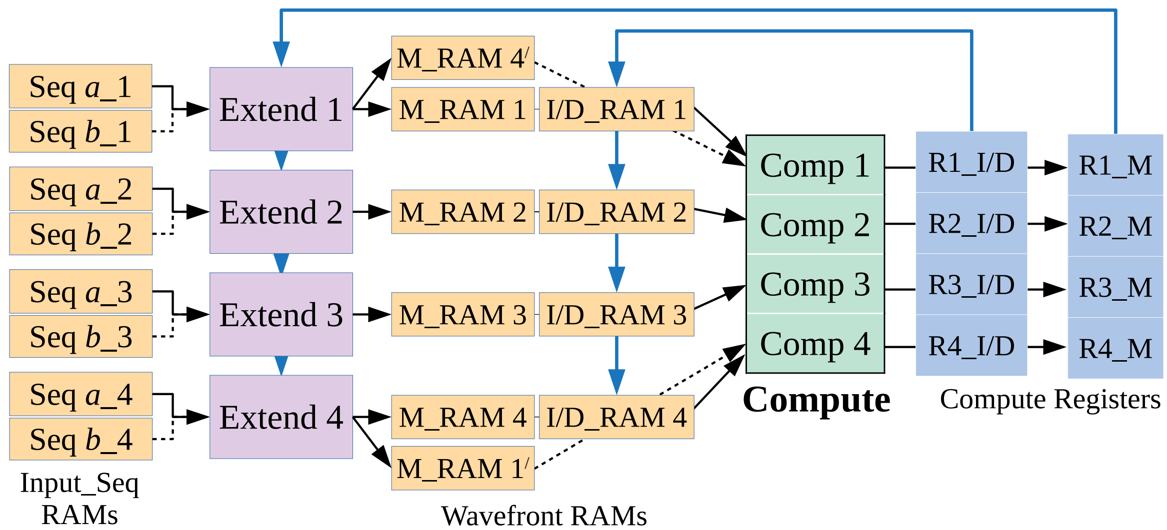


Figure 5.13: Parallel structure of an Aligner using input RAMs and wavefront RAMs in the WFA accelerator for long reads.

reads, the starting positions of the sequences to be compared may not be at the boundaries of the received bases. Hence, two blocks of 16 bases of each sequence are needed to do the shift and align the sequences to the comparator inputs before starting the comparison.

The new version of Extend sub-module for long reads is shown in Figure 5.14. The Extend sub-module, at each clock cycle, sends read requests to the each of the Input_Seq RAMs (RAMs a and b) starting from the address which holds the starting index, and increasing the address by one. The received blocks of sequences a and b are stored in 32-bit registers of REG_1 of Seq_a and REG_1 of Seq_b, respectively. At each clock cycle the value of these registers are shifted in two other registers, REG_2 of sequence a and REG_2 of sequence b, and their values are overwritten by the new values from Input_Seq RAMs. When both registers of the sequences have valid bases, the value of both registers of each sequence are concatenated and shifted to the starting base index of each sequence. The Extend sub-module is pipelined in a way that it compares 16 bases of the two sequences at each clock cycle and, in parallel, it reads the next 16 bases of each sequence that are going to be compared in the next clock cycle.

The Extend sub-modules also require information about the lengths of the sequences. Thus, the lengths are written on the first address of the input RAMs by the Extractor module. The lengths of the sequences are read at the beginning of the alignment by the Aligner and are provided to each Extend. The output result of each Extend is written in the corresponding $\widetilde{M}$ wavefront RAM.

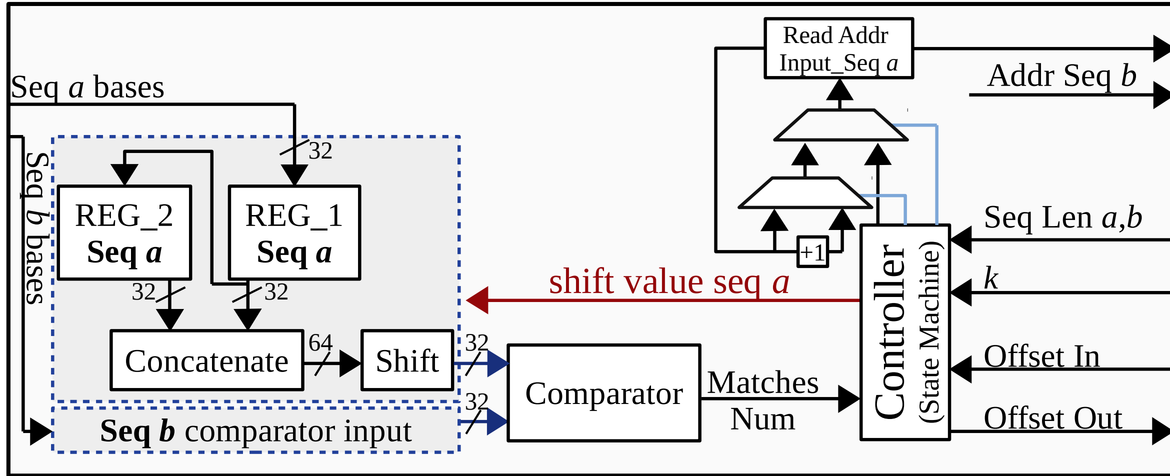


Figure 5.14: Extend sub-module in the FPGA design of the WFA accelerator for long reads.

Compute Sub-module

The Compute sub-module computes the cells of the frame column based on Equation 5.1. In contrast to the accelerator for short reads, the input cells of the accelerator for long reads are read from the wavefront RAMs, so the compute operation takes more time compared to the accelerator of short reads.

When the compute operation finishes, the new values of the $\widetilde{M}$ wavefront are written in the corresponding registers to be extended by the Extend sub-modules, while the new values of the $\widetilde{I}$ and the $\widetilde{D}$ wavefronts are directly written in their corresponding RAMs (Figure 5.13).

Backtrace Data

The backtrace data is generated by the Compute sub-modules, as explained in Section 5.3.3. The design of long reads process 64 cells of the three wavefront vectors of $\widetilde{I}$, $\widetilde{D}$, and $\widetilde{M}$ wavefronts in parallel. Hence, the Compute sub-module concatenates the backtrace data of 64 cells in a backtrace block of 320 bits (5×64). In contrast to the accelerator for short reads, the accelerator for long reads sends the backtrace data to the Collector module instead of to the backtrace RAM. To avoid over-writing backtrace data, the Compute sub-modules check if the previously sent backtrace data has been consumed by the Collector. If the previously sent backtrace data has not been consumed by the Collector, the Aligner is paused until it is consumed. At the end of the alignment, the score and the Success flag are appended to the backtrace data.

5.4.4 Collector Module

The Collector module collects the backtrace data and the final alignment result from different Aligners, packs them in blocks of 128 bytes (data width of our FPGA setup) and sends them to the CPU. Note that the size of the backtrace data depends on the number of parallel sections. For example, when using 64 parallel sections, the size of the backtrace data is 40 bytes (64 parallel sections $\times$ 5-bits of backtrace data / 8). So, for this setup, the Collector module contains three 40-byte registers for each Aligner to collect its backtrace data.

The Collector module accepts backtrace data from an Aligner as long as its corresponding registers are not filled. When the three registers of an Aligner are filled, the Collector module packs the contents of the registers in a block of 128 bytes (3$\times$40 bytes of backtrace data and eight bytes of information) and sends it to a FIFO queue from which the data is sent to the CPU. At this moment, the Collector sets the wait signal for that Aligner to avoid receiving more backtrace data from that Aligner. When the queued backtrace data is sent to the CPU, the Collector unsets the wait signal of that Aligner.

The Collector sends alignment data (backtrace data and alignment result) to the CPU by the format illustrated in Figure 5.15 (A). As shown in this figure, the Collector module sends the data of different Aligners to the CPU without any order. So, in order to be able to identify the data of each alignment in the CPU, the Collector attaches eight bytes of information to each data block of 3$\times$40 bytes. The attached information includes: the ID of the alignment, a counter of the block, and other information including the Last flag, the Success flag, and the position of the alignment result in the Collector registers which are shown in Figure 5.15 (B). The ID determines the pair of sequences that the block of data belongs to. The counter indicates the order of the block of the backtrace data for each ID. The Last flag indicates that this block of the backtrace data is the last one of the alignment and contains the alignment result. Note that the alignment result of an alignment is stored in a backtrace data register of the Collector and sent to the CPU as the last 40-byte block of the last backtrace data of that alignment. Obviously, the format of the alignment result block is different from that of the backtrace data block, as shown in Figures 5.15 (C) and 5.15 (D), respectively. Since we do not know how many backtrace data each alignment may have, the last backtrace data block which contains the alignment result may be stored in any of the three backtrace registers of the Collector, i.e. $n-2$, $n-1$ or n (Figure 5.15 (A)). Hence, the information field contains the position field which specifies the register where the alignment result is written. In addition to alignment score, the alignment result block (Figures 5.15 (C)) also includes the k_{max} and *Length_difference* required

for performing the backtrace. At the end of all the alignments of a batch of reads, the Collector sends three debug data of 128 bytes which are served for debugging purposes.

As shown in Figure 5.15 (D), the first 192 bits (3×64) of each 320-bit (40-byte) backtrace block include the origins of 64 cells of the $\widetilde{M}$ wavefront, The next 64 bits include the origins of 64 cells of the $\widetilde{I}$ wavefront, and the last 64 bits include the origins of 64 cells of the $\widetilde{D}$ wavefront. In a backtrace data block, the origins of cells 0 to 63 are positioned from right to left. For example, in a backtrace block, the origin of cells 0 and 1 of the $\widetilde{M}$ wavefront are located at bits 0 to 2, and 3 to 5, respectively. The origin of cells 0 and 1 of $\widetilde{I}$ wavefront are located at bits 192 and 193, respectively. Similarly, the origin of cells 0 and 1 of $\widetilde{D}$ wavefront are located at bits 256 and 257, respectively.

When the alignment of a batch of reads is finished, the Collector writes into the MMIO registers (explained in Section 3.1.1) the amount of 128-byte results data it has sent to the memory. As the size of backtrace data could vary for different inputs, by reading this register, we find out what range of memory we should read.

5.4.5 Backtrace in CPU

After the FPGA has aligned a batch of reads, the CPU reads the results and does the backtrace. This process is done in different stages. First, multiple CPU threads separate the backtrace data of the alignments in different memory partitions, one per alignment, and remove the information fields. This is done by looking at the alignment ID of each block of backtrace data. After separating backtrace data, each backtrace block of 40 bytes (320 bits) are written consecutively in their specific memory section. At the same stage of separating data, unsuccessful alignments are detected by checking the Success flag and are sent to the Rescue phase.

Once the backtrace data is separated, the backtraces of the alignments are computed. A single thread is responsible for computing the backtrace of an individual alignment, but multiple threads are used to compute multiple backtraces in parallel. The threads first compute the compact CIGAR and then recover the full CIGAR. The processes used to compute the two CIGAR forms are the same as the ones described in the accelerator for short reads, the only difference being that the compact CIGAR form is extracted in the CPU instead of in the FPGA.

Figure 5.16 illustrates the ASM chart and equations to compute the compact CIGAR. It also indicates how each backtrace data of each cell is decoded. The *BT* string holds the backtrace strings: ‘M’ for mismatch, ‘I’ for insertion-opening, ‘D’ for deletion-opening, and ‘E’ for gap(insertion/deletion)-extension. Please note that in this figure:

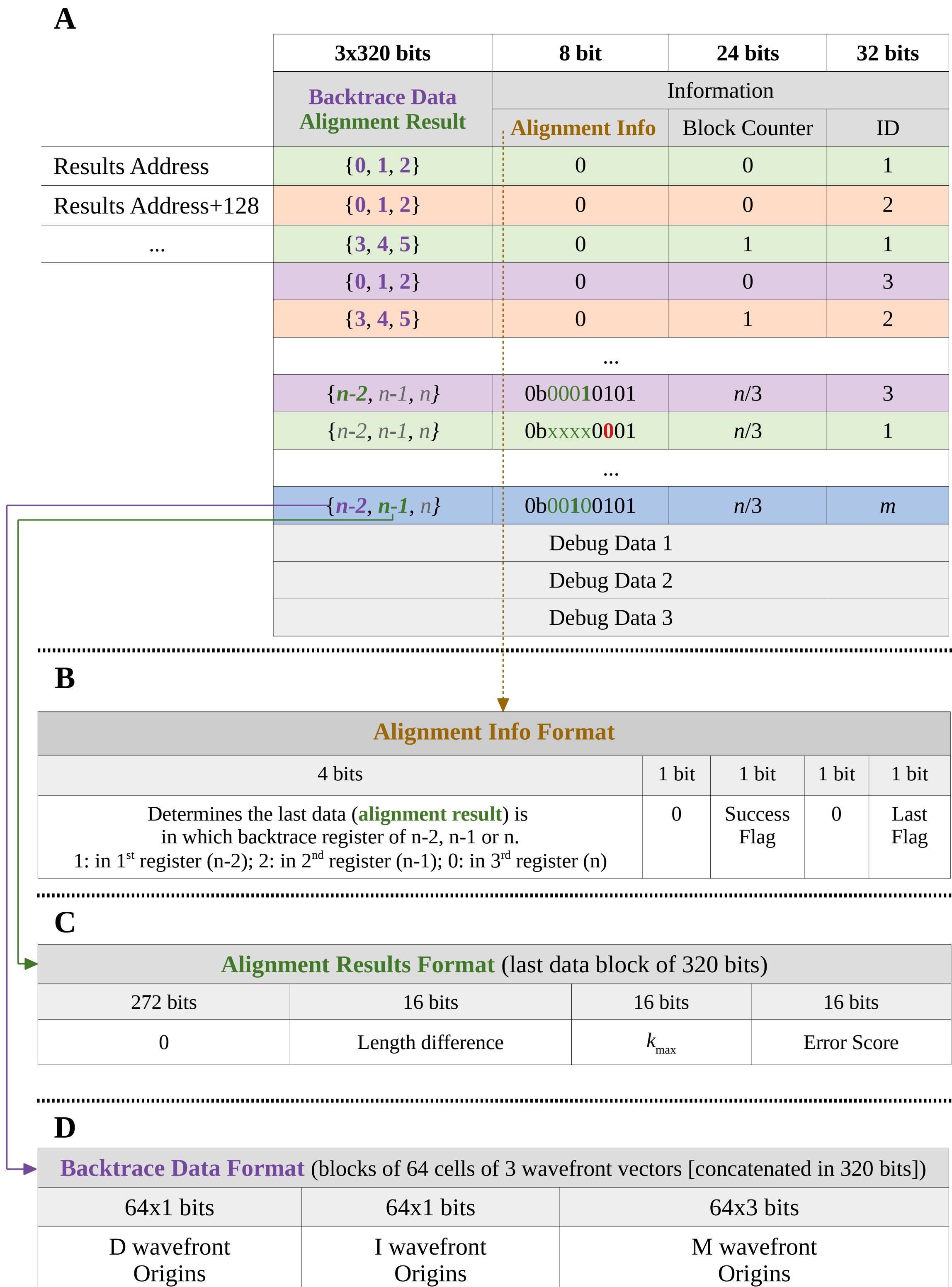


Figure 5.15: The format of writing the alignment and backtrace data in memory for the design of long reads.

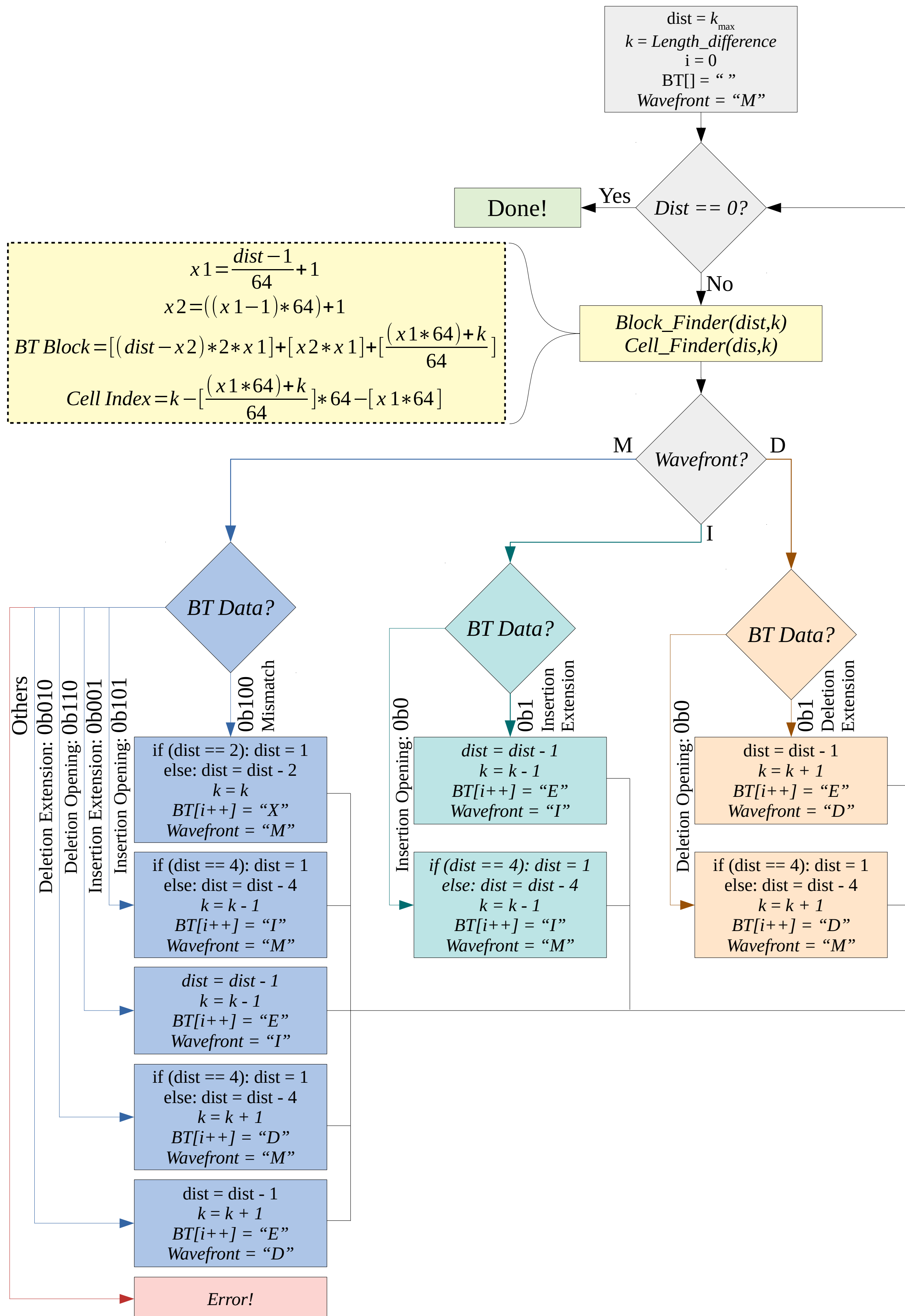


Figure 5.16: ASM chart and equations for calculating compact CIGAR.

- The equations use the k_{max} and *Length_difference* values which are provided in the alignment result field of each ID.
- The *Block_Finder* operation uses *BT Block* equation to give the number (address) of the 320 bits block of backtrace (each memory word contains 320 bits of backtrace data).
- The backtrace block address starts from 0.
- The *Cell_Finder* operation, which uses the *Cell Index* equation, gives the cell index in the backtrace block determined by the *Block_Finder*.

Recovering full CIGAR from short CIGAR is shown in Figure 5.4.

In parallel with the computation of the backtraces, multiple threads also perform the Rescue phase. In this phase, threads use the sequential CPU implementation of the WFA to perform the alignments of sequences that have been failed in the FPGA.

5.5 Evaluation and Results

5.5.1 Experimental Setup

As mentioned in Chapter 3, we evaluate our proposal on a POWER9-based system with 16 cores (64 threads) running at 2.3GHz and two OpenCAPI-enabled ADM-PCIE-9H7 FPGA boards. The FPGA code is written in VHDL and compiled using Vivado v2019.1. In this design the FPGAs run at 200MHz. To integrate our accelerators with the OpenCAPI interface, we use the platform provided by IBM in github [196]. The evaluation reports results for executions with one and two FPGAs.

The proposed WFA accelerator is open source and publicly available [207]. The code allows to configure the sequence length and the maximum k of the FPGA design to meet the input set characteristics. In our evaluation analysis, we selected as many as 25 FPGA designs, in order to cover a wide representative range of sequence lengths and k values, found in state-of-art sequencing technologies.

Table 5.1 summarizes our 25 designs, 7 for short reads (designs 1 to 7), 15 for long reads (designs 11 to 25) and 3 for medium length reads (designs 8 to 10). Designs 8 to 10 are similar to the designs 11 to 13, the main difference is that they do the backtrace in the FPGA and in the CPU, respectively. Designs 8 to 10 are mixed designs that use RAMs to store the required data as the designs for long reads, but they do the backtrace in the FPGA as the designs for short reads.

Table 5.1: FPGA designs, resource utilization, number of Aligners in each FPGA and synthetic inputs used in the evaluation.

NO	FPGA Design				Resource Utilization (%)				Num	Input
	Len	*k*	PS[1]	BT[2]	LUT	FF	BRAM	URAM	Aligners[3]	Len-ERR[4]
1	100	16	8	FPGA	88	26	9	0	100	100-5%
2	100	32	8	FPGA	91	33	8	0	80	100-8%
3	150	16	8	FPGA	84	26	8	0	80	150-3%
4	150	32	8	FPGA	86	32	7	0	64	150-5%
5	150	64	8	FPGA	86	37	6	0	45	150-8%
6	300	32	8	FPGA	87	30	7	0	55	300-3%
7	300	64	8	FPGA	90	37	6	0	40	300-5%
8	1K	224	32	FPGA	92	23	95	77	44	1K-5%
9	1K	448	32	FPGA	93	24	92	83	42	1K-10%
10	1K	896	32	FPGA	89	21	93	84	37	1K-20%
11	1K	256	32	CPU	97	21	88	83	44	1K-5%
12	1K	512	32	CPU	97	21	84	83	42	1K-10%
13	1K	1K	32	CPU	98	20	79	83	40	1K-20%
14	5K	1K	64	CPU	95	21	94	83	21	5K-5%
15	5K	2K	64	CPU	89	19	96	83	19	5K-10%
16	5K	4K	64	CPU	95	19	70	83	18	5K-20%
17	10K	2K	64	CPU	95	19	85	83	17	10K-5%
18	10K	4K	64	CPU	91	18	92	83	16	10K-10%
19	10K	8K	64	CPU	92	16	96	90	14	10K-20%
20	25K	5K	64	CPU	78	14	92	90	12	25K-5%
21	25K	10K	64	CPU	85	13	96	90	11	25K-10%
22	25K	20K	64	CPU	93	11	93	90	9	25K-20%
23	50K	10K	64	CPU	74	12	92	90	9	50K-5%
24	50K	20K	64	CPU	72	10	95	93	8	50K-10%
25	50K	40K	64	CPU	73	8	94	96	6	50K-20%

[1] PS: Parallel Sections, determines how many cells of a wavefront matrix are computed in parallel.
[2] BT: BackTrace, determines if backtrace is performed in the CPU or in the FPGA.
[3] Num Aligners: Number of Aligner cores which fit in the design of one FPGA.
[4] Len-ER: Length - Error Rate(%), determines the length and the error rate between pairs of sequences in the inputs set applied to the FPGA design.

Table 5.1 describes, for each design, its maximum sequence length and *k*, its number of parallel sections, its type of backtrace implementation, its resource utilization, and the number of parallel Aligners that fit in each FPGA. The table also shows the characteristics of the synthetic input sets that are fed to each design. Detailed characteristics of the input sets are summarized in Table 3.1.

We feed each synthetic input set of Table 3.1 (contributions 2 and 3), which is also reflected in the last column of Table 5.1, to its corresponding FPGA design. In Section 5.5.4 we use real input sets of Table 3.1 (contributions 3).

Note that, given a maximum sequence length and k, an FPGA design can correctly process any input containing shorter sequences and smaller ks. Nevertheless, a tailored instantiation requires less space in the FPGA and maximizes the number of Aligners that can fit in the FPGA.

Equation 5.3 shows the relation of the k value and the score for the penalties used in this work, i.e. $(x, g_o, g_e) = (4, 6, 2)$.

$$Score = k \times 2 + 4 \tag{5.3}$$

Regarding the penalties and the equation above, The k value determines how many differences in a pair of sequences can be supported by the FPGA design. For example, design 18 uses k=4K, so it can support scores of up to 8K. A gap-opening has the maximum penalty score of 8 ($g_o + g_e$). Thus, if all the differences between the pair of sequences are gap-openings (which is the worst case) 1K differences between the sequences can be detected in an FPGA design with k=4K, regardless of the length of the sequences.

As explained in Chapter 3, the baseline used in the evaluation is the reference CPU implementation of the WFA algorithm proposed by Marco-Sola et al. [19], which is open source and publicly available [200]. In the evaluation we refer to this CPU-only baseline implementation as WFA-CPU. In our experiments we perform an exploration of the batch size (number of reads in each batch) on the FPGA designs for long reads, and also an exploration of the number of CPU threads (from 1 to 64) on the WFA-CPU implementation and on all the FPGA designs. All the results report the execution time of the best performing number of threads and batch sizes unless stated otherwise. The time and energy measurements include the data transfers between the CPU and the FPGAs, while parsing the input files is excluded in the measurements of all evaluation setups (WFA-FPGA or WFA-CPU).

The WFA-CPU software implementation is also used to verify our accelerators. To do so, in all the experiments we check that the output of the WFA using the accelerators is exactly the same as the output of the WFA-CPU.

To measure the power consumption of the entire node (CPU and FPGAs), we use in-band readings from Linux to the OCC [206].

Compared to a reference multithreaded CPU implementation of the traditional SWG [68] algorithm, the WFA-CPU achieves speedups of 8.4× to 53.3× for the 7 input sets applied to

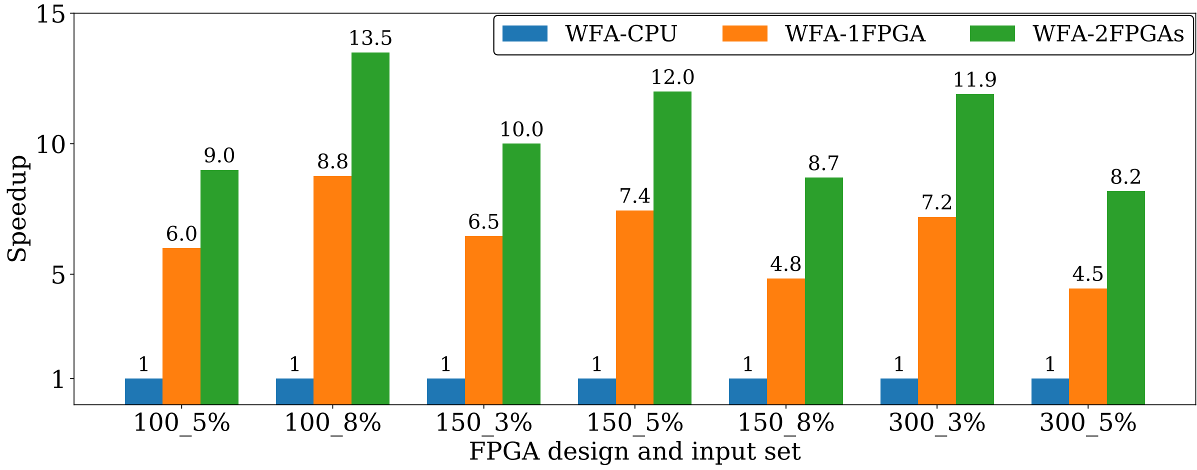


Figure 5.17: Speedup of the FPGA designs of the WFA accelerator for short reads with respect to WFA-CPU.

designs 1 to 7 of short reads, and 3× to 132× for the 15 input sets applied to designs 8 to 25 of medium and long reads. The proposed WFA accelerator outperforms SWG by 42.6× to 383.8× with one FPGA and by 76.6× to 634.3× with two FPGAs for input sets applied to the short reads designs; 18× to 193× with one FPGA and 36× to 378× with two FPGAs for input sets applied to the medium reads designs; and 13× to 400× with one FPGA and 26× to 794× with two FPGAs for input sets applied to the long reads designs. In addition, compared to a Banded Smith-Waterman heuristic method [200] that does not perform the backtrace, the proposed WFA accelerator achieves speedups of 37.4× to 93.5× with one FPGA and of 55.9× to 154.8× with two FPGAs for input sets applied to the short reads designs; 12× to 41× with one FPGA and 24× to 80× with two FPGAs for input sets applied to the medium reads designs; and 9× to 77× with one FPGA and 17× to 152× with two FPGAs for input sets applied to the long reads designs.

5.5.2 Results of Short Reads for Synthetic Input Sets

Figure 5.17 shows the speedup achieved by applying seven different synthetic input sets to seven designs (designs 1 to 7) of the proposed WFA accelerator for short reads compared to the WFA-CPU. The different FPGA designs achieve speedups of 4.5× to 8.8× with one FPGA, and of 8.2× to 13.5× when the two FPGAs in the system are used. The lowest speedups belong to the designs with biggest ks. This happens because the size of the Aligner module increases as k grows, so fewer Aligners fit in the FPGA and fewer sequences are aligned in parallel. Using two FPGAs increases the speedup by a factor of 1.5× to 1.8× compared to one FPGA.

Table 5.2: Duration (in clock cycles) of alignment, backtrace and extracting reads of one sequence pair and maximum efficient Aligners in each FPGA.

FPGA Design	Alignment (clk cycles)	Backtrace (clk cycles)	Total (clk cycles)	Extracting Input (clk cycles)	Max Efficient Num of Aligners
1	185	15	200	3	67
2	265	23	288	3	196
3	185	15	200	4	50
4	265	23	288	4	147
5	1900	39	1939	4	485
6	265	23	288	6	98
7	1900	39	1939	6	324

Using two FPGAs doubles the speed of the FPGA part of the co-design, but the time of the CPU part remains unchanged. As a result, the total execution time is not halved.

Table 5.2 shows, for each FPGA design, the maximum possible number of Aligners before saturating the OpenCAPI bandwidth. The table indicates, for each design, how many FPGA clock cycles are needed to do the alignment, the backtrace, and to extract one pair of sequences. From these numbers, the maximum possible Aligners in each system is calculated and shown in the last column. In this test we feed each design with inputs with k values equal to the maximum supported k in each design. If these designs are fed with inputs with smaller ks, the bandwidth is saturated with even fewer Aligners. As shown in Table 5.1, design 3 has 80 Aligners per FPGA, although Table 5.2 shows that this design saturates the bandwidth with 50 Aligners. So, in this design, adding more than 50 Aligners per FPGA does not provide any benefit because the Extractor module cannot feed them with data on time due to bandwidth limitations.

Figure 5.18 shows the scalability of multi-threaded runs of the FPGA accelerator with one FPGA and of the WFA-CPU. All the speedups are computed over the single threaded execution of the WFA-CPU. The POWER9 CPU has 64 threads, so the scalability of the WFA-CPU and the FPGA designs saturates at that point. In single threaded runs, the FPGA designs achieve speedups over WFA-CPU of 19× to 32×. The scalability of the WFA-CPU is linear up to 16 threads. However, after that point, the parallelization efficiency drops because the threads share the resources of the CPU cores. The scalability of the co-designs is less effective because, when increasing the number of threads, the time spent in the CPU part decreases and, hence, the constant thread-independent FPGA time dominates the total execution time.

Next we study the impact of encountering input pairs of sequences with ks larger than the maximum k supported in the FPGA designs. We feed the FPGA design 4 with an input in

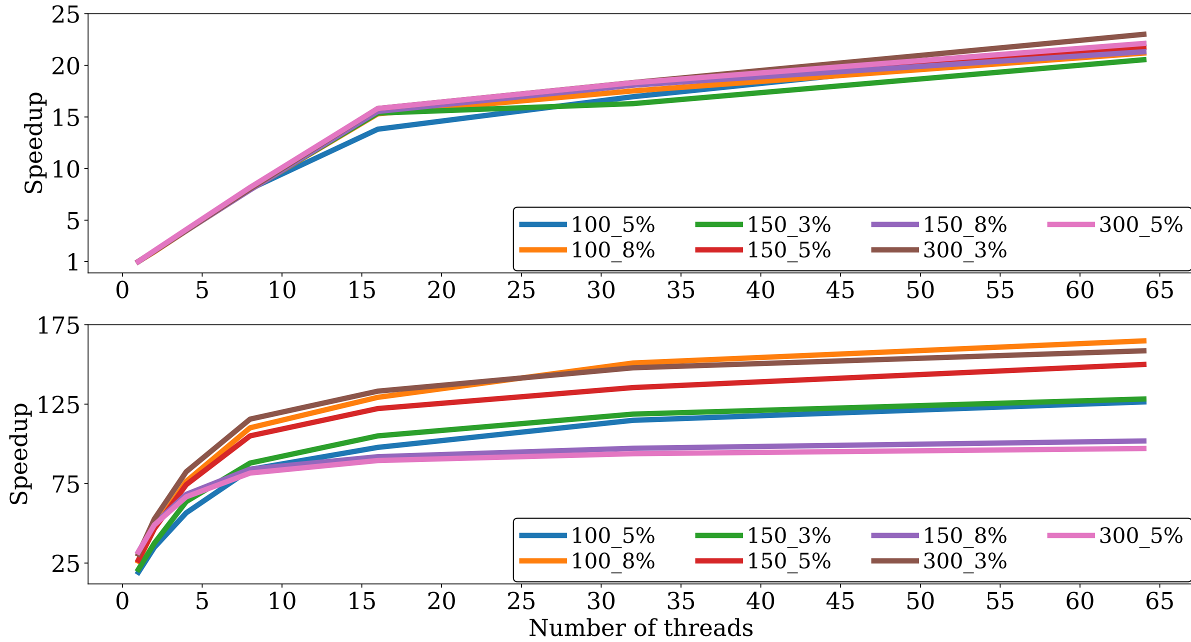


Figure 5.18: Speedup of the FPGA designs for short reads with WFA-CPU (top) and one FPGA (bottom) for multi-threaded runs over single-threaded WFA-CPU.

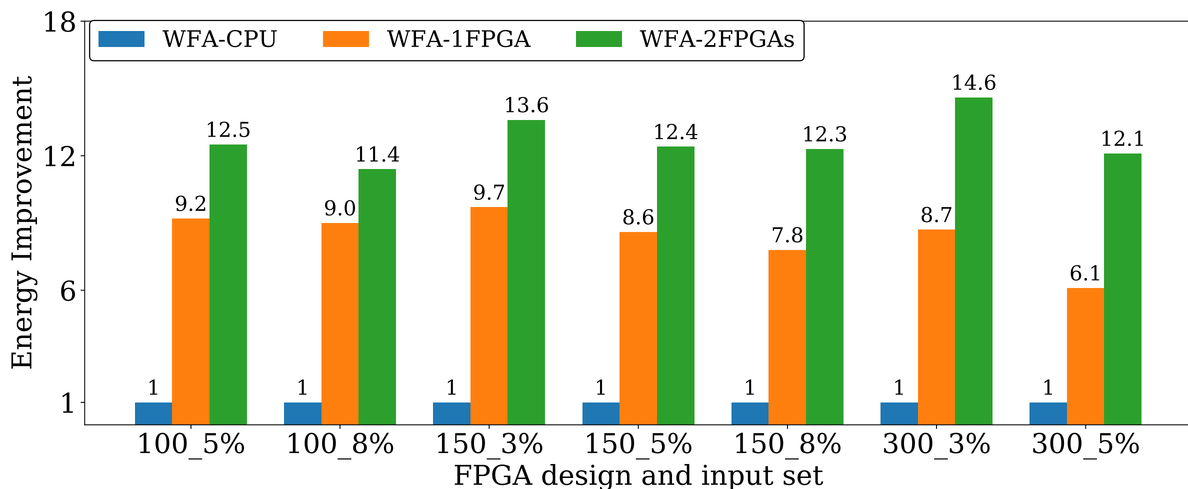


Figure 5.19: Energy improvement of the FPGA designs for short reads with respect to WFA-CPU.

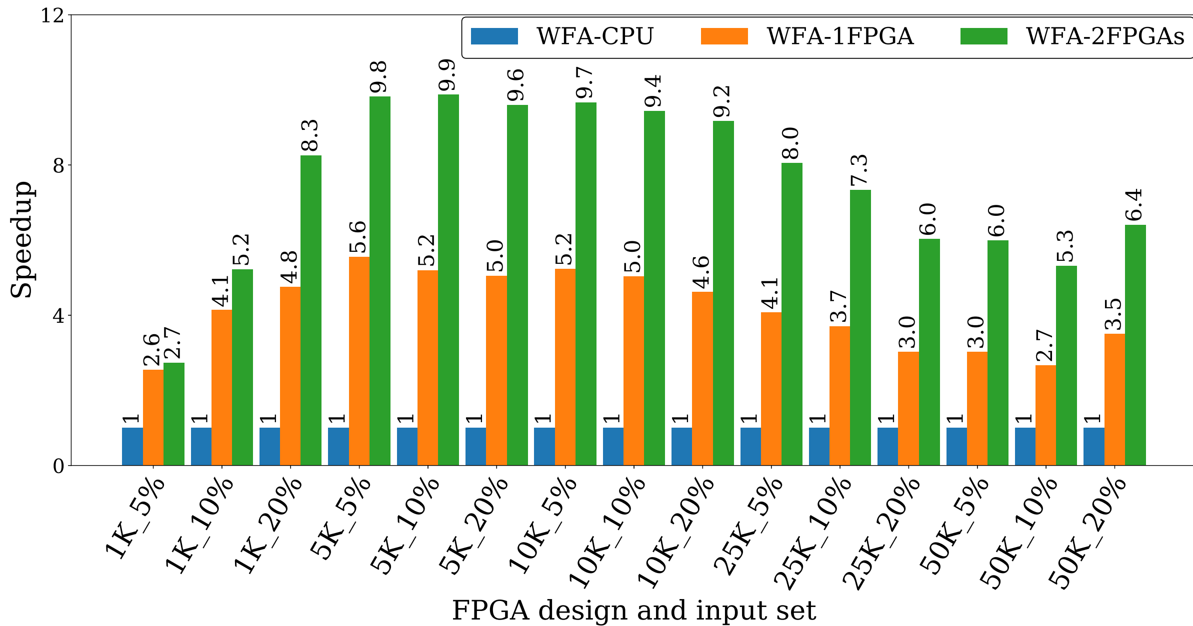


Figure 5.20: Speedup of the FPGA designs for long reads with respect to WFA-CPU.

which 10% of the total sequences have scores larger than what is supported by the design, so the alignment of that percentage has to be performed in the CPU. The best execution times of the FPGA design are 1318ms with one FPGA and 1030ms with two FPGAs, while the fastest WFA-CPU execution for this input is 4500ms, so the FPGA design achieves speedups of 3.4× and 4.4× with one and two FPGAs, respectively.

Finally, Figure 5.19 shows the improvement in energy consumption of the different FPGA designs of the WFA accelerator for short reads compared to the WFA-CPU. To report fair and meaningful power consumption measurements, in this experiment we repeat the alignment of the same input set several times so the fastest execution takes at least 30 seconds. For each input set, the number of repetitions and the total amount of work to be performed is the same in the WFA-CPU and the FPGA design. Results show that the different FPGA designs consume significantly less energy than the WFA-CPU, with energy improvements of 6.1× to 9.7× with one FPGA and of 11.4× to 14.6× with two FPGAs.

5.5.3 Results of Long Reads for Synthetic Input Sets

Figure 5.20 shows the speedup achieved by applying 15 different synthetic input sets to the 15 designs (11 to 25) of the proposed WFA accelerator for long reads, compared to applying those input sets to the WFA-CPU. The different FPGA designs achieve speedups of 2.6× to 5.6×

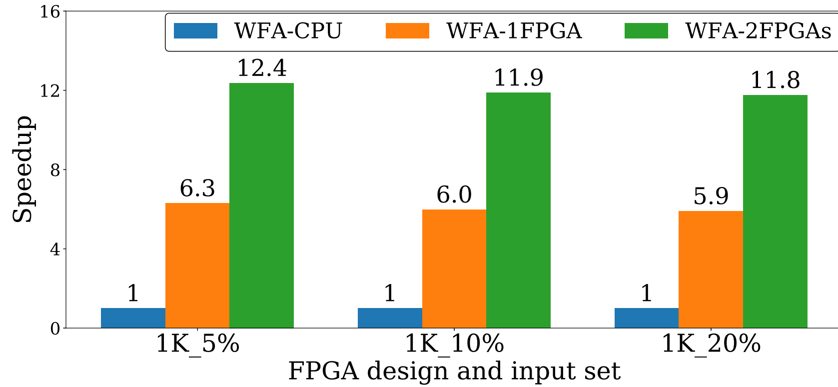


Figure 5.21: Speedup of the FPGA designs for medium reads with respect to WFA-CPU.

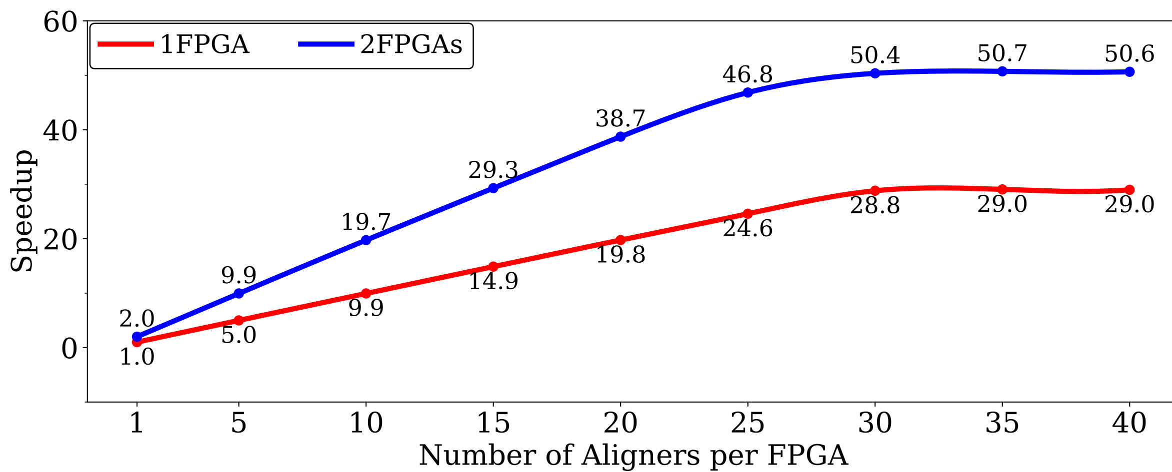


Figure 5.22: Speedup of the FPGA design 13 for long reads with different number of Aligners and with one and two FPGAs with respect to the same design with one Aligner and one FPGA.

with one FPGA, and of 2.7× to 9.9× with two FPGAs. The best speedups are achieved for input sets with lengths of 5K bases and 10K bases, which are applied to designs 14 to 19.

We observe that the speedups decrease at both ends of the figure. At the right end of the figure this happens because, as shown in Table 5.1, the FPGA resources used by the Aligners increase with the supported read length and error rate, so less Aligners can fit inside the FPGA. Note that this trend changes for the last input set (50K_20%) because the WFA-CPU can only be executed using eight threads. This is due to memory limitation issue by using 64 threads. At the left end of the figure, for read lenghts of 1K bases, the speedups of the FPGA designs are also lower even though a large number of Aligners fit in the FPGA. The reason is that, for these inputs, the FPGA alignment time is relatively fast but the time spent sending backtrace data to the CPU and performing the backtrace in the CPU dominate the total execution time.

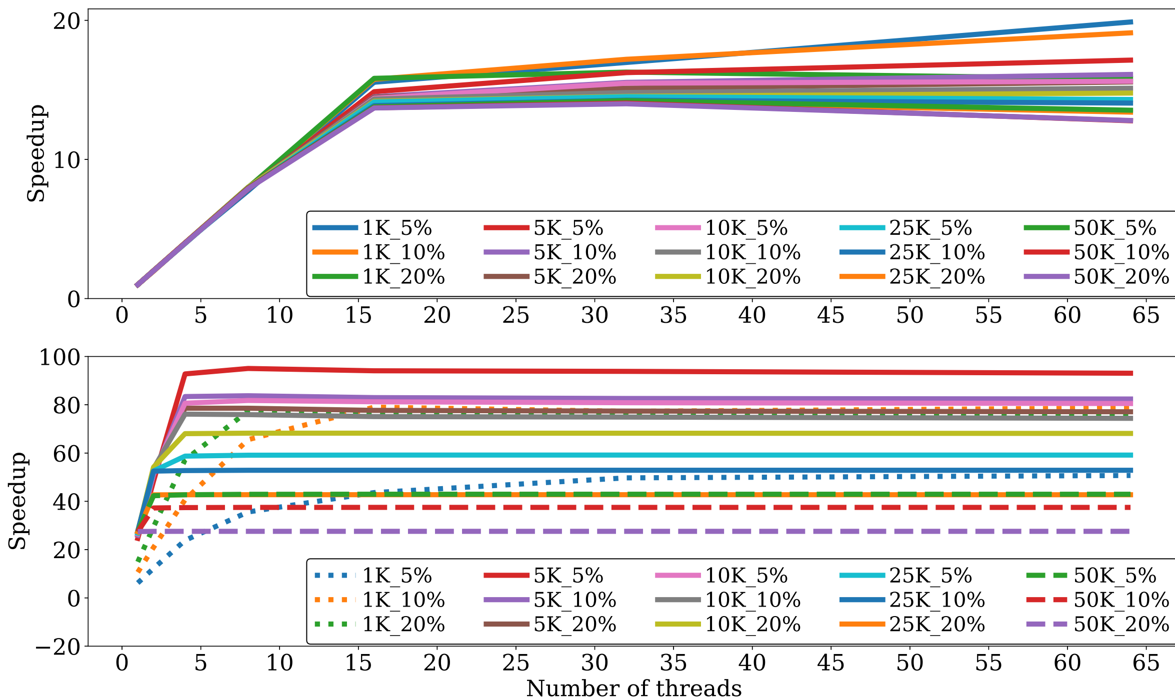


Figure 5.23: Speedup of the FPGA designs for long reads with WFA-CPU (top) and one FPGA (bottom) for multi-threaded runs over single-threaded WFA-CPU.

Designs 8 to 10 in Table 5.1 attempt to increase the speedups achieved for input sets with read lengths of 1K bases. To this end, these designs decrease the amount of work done in the CPU and avoid saturating the bandwidth by doing the backtrace in the FPGA. With this approach, the amount of data sent from the FPGA to the CPU is drastically reduced, and the CPU only needs to perform the computation of the full CIGAR from compact CIGAR as in the designs for short reads. Thus, these designs combine features of the designs for long reads, in which data is stored in RAMs, and the designs for short reads, in which the backtrace in compact CIGAR form is computed in the FPGA. This creates mixed designs which we call medium reads designs. Figure 5.21 shows the speedups achieved by the medium reads designs (8 to 10 in Table 5.1) over WFA-CPU when they are fed input sets with read lenghts of 1K bases and error rates of 5%, 10% and 20%. The speedups are significantly larger than the ones achieved by the designs for long reads, and they scale perfectly to two FPGAs.

Figure 5.22 shows the speedups of the design 13 in Table 5.1 with different number of Aligners with regard to using only one FPGA and one Aligner. The figure shows that the design scales perfectly when increasing the number of Aligners and FPGAs as long as the bandwidth is not saturated. For this specific design and input set (1K_20%), the bandwith is saturated

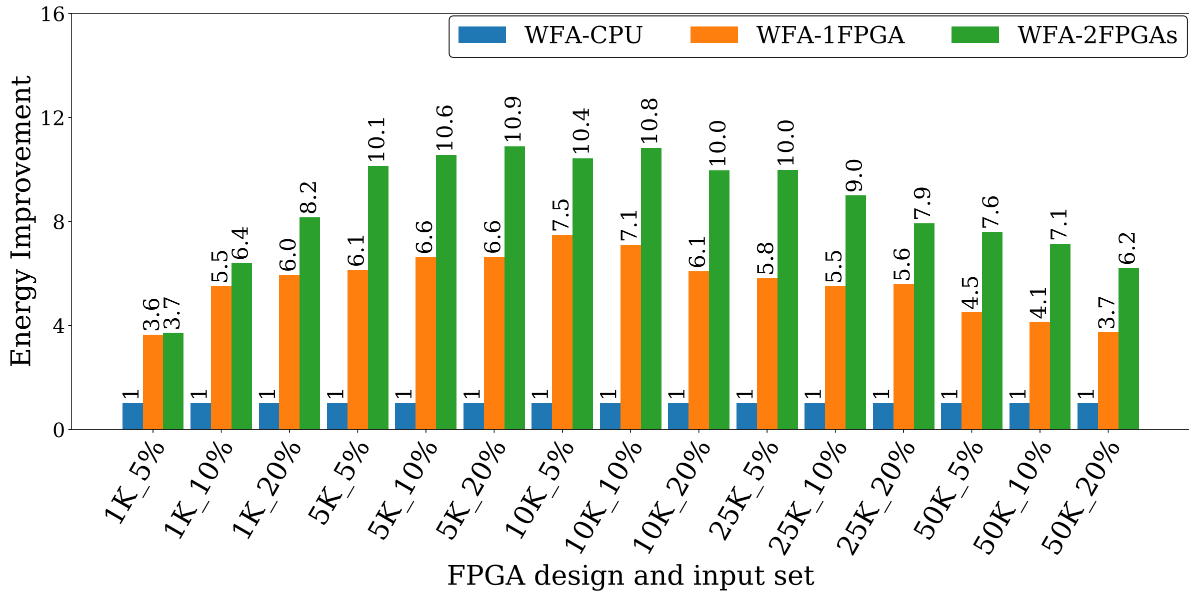


Figure 5.24: Energy improvement of the FPGA designs for long reads with respect to WFA-CPU.

at nearly 30 Aligners per FPGA. This figure is a representative for the scalability of all the designs for long reads, which present the same trend.

Figure 5.23 shows the scalability of multi-threaded runs of the FPGA designs of the WFA accelerator for long reads (designs 11 to 25 in Table 5.1) with one FPGA and of the WFA-CPU. All the speedups are computed over the single threaded execution of the WFA-CPU. In single threaded runs, the FPGA designs achieve speedups over WFA-CPU of 10× to 27× (Figure 5.23 (bottom)). The WFA accelerator for long reads requires multiple threads: one main thread, one thread to control the operation of the FPGA, and multiple threads to perform the backtrace and the Rescue. However, the Rescue is not active for synthetics inputs. In Figure 5.23 the number of threads for the co-design is the number of threads which are doing the backtrace. In the smaller designs with read lengths of 1K bases the executions have longer CPU times compared to the FPGA times, specially due to the computation of the backtraces. For this reason, in these designs, increasing the number of threads up to 16 gradually increases the speedup. In contrast, in the designs with larger read lengths, the execution times are dominated by the FPGA, which are significantly larger than the CPU times. Thus, increasing the number of threads does not provide performance benefits, and the speedups saturate using less than four threads. Figure 5.23 (top) shows that the scalability of the WFA-CPU increases linearly until 16 threads, which is the number of cores of the POWER9 CPU. After 16 threads the parallel efficiency drops because the threads have to share the resources of the CPU cores.

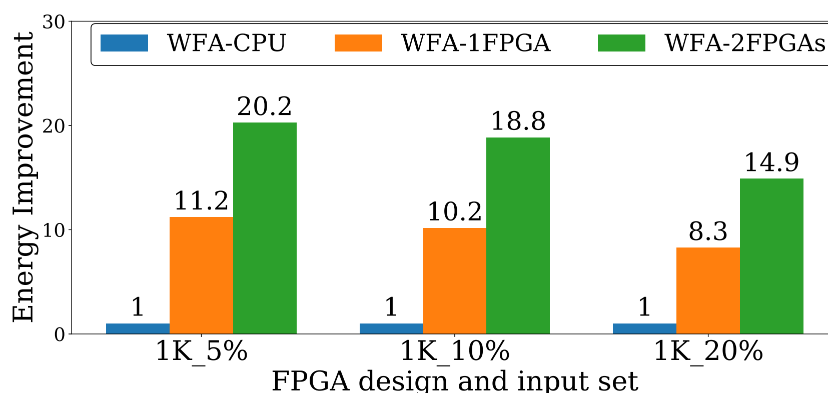


Figure 5.25: Energy improvement of the FPGA designs for medium reads with respect to WFA-CPU.

Finally, Figure 5.24 shows the improvement in energy consumption of the different FPGA designs of the WFA accelerator for long reads compared to the WFA-CPU. Similarly, Figure 5.25 shows the energy improvements of the FPGA designs for medium reads over WFA-CPU. Results show that the different FPGA designs for long and medium reads consume significantly less energy than the WFA-CPU, with energy improvements of 3.6× to 7.5× with one FPGA and of 3.7× to 10.9× with two FPGAs for long reads. When input sets with read lengths of 1K bases are applied to the designs for medium reads (8 to 10 in Table 5.1), the energy improvements are significantly higher than the ones achieved by the designs for long reads with read lengths of 1K bases (designs 11 to 13 in Table 5.1). In this case, energy improvements are 8.3× to 11.2× with one FPGA and 14.9× to 20.2× with two FPGAs.

5.5.4 Results of Long Reads for Real Input Sets

This section evaluates our FPGA designs of the WFA accelerator for long reads using two real input sets. Compared to the previous subsections using synthetic input sets, the real input sets include sequences with very diverse characteristics in terms of read lengths and, thus, errors in the alignments. For these reasons, some of the alignments cannot be computed by the FPGA, so the functionality of the Rescue in Figure 5.8 becomes more relevant.

For this evaluation we use the two publicly available input sets of Table 3.1. These input sets do not have a fixed length. Hence, we analyze the real input sets and summarize their characteristics in Table 5.3. It can be observed that PacBio HiFi has longer reads with less quality compared to PacBio CCS.

Table 5.3: Real input sets specifications.

Input set	Avg score	Max score	Avg len	Max len	Reads including unknown base 'N'
PacBio HiFi	370	11.8K	12.8K	24.6K	881
PacBio CCS	225	9.4K	9.6K	18.3K	175

Input set	Reads with len >16K	Alignments with score >			Failed alignments in FPGA design of Table 5.4		
		4K	8K	16K	1	2	3
PacBio HiFi	7.2K	4.9K	146	0	12.5K	8.2K	8.1K
PacBio CCS	1	131	1	0	284	175	175

Table 5.4: FPGA design configurations evaluated with real input sets.

NO	Len	*k*	Max Score	Parallel Sections	Backtrace	Num Aligners
1	16K	2K	4K	64	CPU	17
2	16K	4K	8K	64	CPU	16
3	16K	8K	16K	64	CPU	14

Based on the characteristics of the input sets, we extend the FPGA designs 17, 18 and 19 of Table 5.1 to support read length up to 16K bases, with almost the same resource utilization. The resulted designs with their specifications are shown in Table 5.4.

Table 5.3 also shows the number of sequences that contain 'N' bases, the number of sequences with lengths larger than 16K bases, and alignments with scores higher than 4K, 8K and 16K (maximum supporting scores of the designs in Table 5.4). In all these cases, the alignments cannot fully be performed by the FPGA designs, and they fall back to the CPU implementation in the Rescue. The last three columns of the bottom part of the table indicate the total number of alignments that cannot be done by the FPGA and have to be computed by the Rescue.

We feed the real input sets to the three designs of Table 5.4. For each design we perform an exploration of the number of sequences in every batch (batch size) and the distribution of the total 64 threads performing the backtrace and Rescue. The best performance for both input sets is achieved by the design 2 (read length of 16K and design parameter k=4K) with a batch size of 5K sequences. For PacBio HiFi, the best thread distribution with one FPGA is 16 threads doing backtrace and 32 threads doing Rescue, while with two FPGAs it is 8 threads doing backtrace and 16 threads doing Rescue. For PacBio CCS, the best thread distribution with one FPGA is 16 threads doing backtrace and 48 threads doing Rescue, while with two FPGAs it is 16 threads doing backtrace and 32 threads doing Rescue.

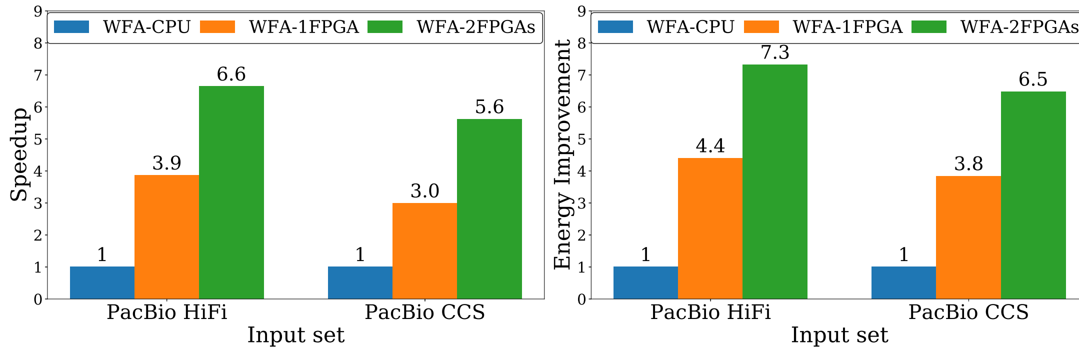


Figure 5.26: Speedup (left) and energy improvement (right) of the FPGA designs of long reads with respect to WFA-CPU when applying real input sets.

Figure 5.26 shows the speedup (left) and energy improvement (right) achieved by the FPGA design 2 with the aforementioned configurations over WFA-CPU when fed with real input sets. Results show that the FPGA design, for PacBio HiFi, exhibits speedups of 3.9× and 6.6× with one and two FPGAs, respectively. The speedups for PacBio CCS are 3.0× with one FPGA and 5.6× with two FPGAs. The energy improvements for PacBio HiFi with one and two FPGAs are 4.4× and 7.3×, respectively, while for PacBio CCS with one and two FPGAs are 3.8× and 6.5×, respectively.

5.5.5 Performance Comparison

In this section we compare our FPGA-based accelerators with other FPGA accelerators which perform exact gap-affine-based alignment. For this comparison we use Cell Updates Per Second (CUPS) which is a common metric used to measure the performance of SW algorithms independently of their target devices and implementation specifics. CUPS represent the number of cells from the DP-matrix computed per second.

Table 5.5 shows the GCUPS (Giga CUPS) achieved by our accelerators, for all the considered input sets. It is important to note that the WFA algorithm avoids the full computation of the DP-matrix, so GCUPS can be computed in two ways. On the one hand, the *SWG Equivalent* column of Table 5.5 shows the GCUPS achieved by our co-designs considering the equivalent number of DP-matrix cells that the SWG algorithm would need to compute the optimal alignment. On the other hand, the *Computed* column of Table 5.5 shows the GCUPS considering only the average number of wavefront cells computed by the WFA. The last column of Table 5.5 shows the percentage of the average number of wavefront vector cells that the WFA computes compared to the DP-matrix cells that the SWG computes. It can be seen that

Table 5.5: Computed and equivalent GCUPS achieved by our co-designs for different inputs.

Input Length - Error Rate	GCUPS				WFA/SWG Cell Update (%)
	SWG Equivalent		Computed		
	1 FPGA	2 FPGAs	1 FPGA	2 FPGAs	
100-5%	320.5	478.5	7.7	11.5	2.41
100-8%	238.1	367.6	14.3	22.1	6.01
150-3%	637.4	991.2	6.8	10.6	1.07
150-5%	409.1	661.8	10.9	17.7	2.67
150-8%	138.5	251.7	8.2	14.9	5.92
300-3%	1251.7	2073.7	10.5	17.4	0.84
300-5%	423.5	776.5	9.7	17.9	2.30
1K-5%	980.4	1923.1	22.2	43.6	2.27
1K-10%	322.6	641.0	29.1	57.9	9.03
1K-20%	103.4	206.2	37.3	74.4	36.06
5K-5%	802.3	1420.5	18.1	32.0	2.25
5K-10%	248.5	471.7	22.4	42.5	9.01
5K-20%	87.8	167.0	31.6	60.1	36.01
10K-5%	823.1	1519.8	18.5	34.2	2.25
10K-10%	252.7	473.9	22.7	42.7	9.00
10K-20%	81.9	162.5	29.5	58.5	36.01
25K-5%	679.5	1342.4	15.3	30.2	2.25
25K-10%	188.1	371.7	16.9	33.5	9.00
25K-20%	53.2	106.0	19.1	38.2	36.00
50K-5%	512.2	1016.3	11.5	22.9	2.25
50K-10%	134.8	268.2	12.1	24.1	9.00
50K-20%	34.5	63.3	12.4	22.8	36.00

reduction in the number of computed cells by the WFA is independent of the read length and only depends on the error rate. It should be taken into account that the computation of a WFA wavefront vector cell is more costly than the computation of a SWG DP-matrix cell. This is because the computation of the WFA wavefront vector cell requires two operations, compute and extend, while the computation of the DP-matrix cell in SWG only the requires the compute operation.

In Table 5.6 we compare our peak GCUPS, for different input lengths, with other accelerators. Our WFA accelerator achieves significantly higher performance than all the FPGA-SWG-based state-of-the-art optimal solutions.

Table 5.6: Peak GCUPS of different exact SWG FPGA accelerated methods.

Work	Year	Device	Freq. (MHz)	Read Len.	Algorithm	GCUPS
Ours	2023	2x Virtex U+ 37P	200	5K∼50K	WFA	1519.8
Ours	2023	1x Virtex U+ 37P	200	5K∼50K	WFA	823.1
Ours	2023	2x Virtex U+ 37P	200	1K	WFA	1923.1
Ours	2023	1x Virtex U+ 37P	200	1K	WFA	980.4
Ours	2021	2x Virtex U+ 37P	200	100∼300	WFA	2073.7
Ours	2021	1x Virtex U+ 37P	200	100∼300	WFA	1251.7
[179]	2019	Virtex U+ VU9P	200	50∼200	SWG	8.7[1]
[172]	2017	Virtex 7	200	128∼8.2K	SWG	105.9
[177]	2017	Arria 10	N/A	10K∼67M	SWG	268.8[2]
[180]	2015	2x Stratix V	N/A	144∼5.5K	SWG	442
[187]	2013	Stratix V A7	193	JF801956.1[3]	SWG	24.7
[181]	2011	Xilinx XC5VLX330T	130	N/A	SWG	129.0[4]
[186]	2009	Xilinx XC2V6000-4	47.6	362	SWG	8.0
[182]	2007	Altera EPS1S30	82	4K,40K,80K	SWG	6.6

[1] GCUPs for this accelerator are not reported explicitly, we calculate them from the data provided in the original paper [179].
[2] This accelerator does not perform backtrace.
[3] Mamavirus's complete genome.
[4] This accelerator reports the theoretical FPGA peak performance, without considering I/O limitations.

5.6 Conclusions

This chapter presents efficient FPGA accelerators of the WFA algorithm that accelerates the pairwise alignment of DNA sequences. We propose two different accelerators customized for short and for long DNA sequences. The WFA accelerator for short sequences stores the necessary data in the registers in order to have fast access to them. It calculates the backtrace in compact CIGAR form inside the FPGA, while the computation of the full CIGAR is done in the CPU. In contrast, the WFA accelerator for long reads stores the necessary data in RAMs instead of registers, since the size of the sequences and the data structures of the WFA algorithm are much larger. As the backtrace data is also immense, the FPGA sends the raw backtrace data to the CPU and both the compact CIGAR and the full CIGAR computations are performed in the CPU.

We implement and evaluate the proposed WFA accelerator in a POWER9-based system with two FPGAs connected with OpenCAPI, which provides coherent access to the host memory from the FPGAs and ideal data transfer speeds of up to 25GB/s. Results show that, for different combinations of sequence lengths and error rates, the WFA accelerator largely outperforms

the reference WFA CPU-only implementation. The WFA accelerator for short reads achieves speedups of 4.5× to 8.8× with one FPGA, and of 8.2× to 13.5× with two FPGAs, while reducing energy to solution by 6.1× to 9.7× with one FPGA and by 11.4× to 14.6× with two FPGAs. The WFA accelerator for long reads achieves speedups of 2.6× to 5.6× with one FPGA, and of 2.7× to 9.9× with two FPGAs, while reducing energy to solution by 3.6× to 7.5× with one FPGA and by 3.7× to 10.9× with two FPGAs.

Chapter 6

WFA ASIC Accelerator

6.1 Introduction

Most of the SW-based FPGA and ASIC accelerators targeting long reads tend to implement a heuristic method as there is not enough space on the FPGA or a big area is needed for the ASIC to store the data of the exact method. This problem also results in low working frequencies of the exact-based accelerators due to the routing complexity.

This chapter presents WFAsic, the first ASIC accelerator for exact pairwise alignment of long reads based on the WFA algorithm. Our WFAsic implementation includes one Aligner core and supports read lengths up to 10Kbp and error rates up to 10%. However, these parameters are configurable depending on the available resources and input characteristics. As WFA only computes a reduced number of the DP-matrix cells to find the optimal alignment, our WFAsic accelerator is able to perform exact pairwise read alignment even for long reads, and fits in an area of 1.6mm^2 after PnR in GlobalFoundries 22nm technology, and reaches a frequency of 1.1GHz.

This chapter also presents the integration of the WFAsic accelerator in a Linux-capable RISC-V SoC. The WFAsic accelerator is configured using a standard Linux driver and API. In addition, the WFAsic accelerator runs as an independent process in parallel to other CPU processes. Integrating the WFAsic accelerator with the CPU in the same SoC provides great benefits to genomics applications, as it eliminates the need for external accelerators and their costly communication. The integrated WFAsic accelerator provides performance improvements of up to 1076$\times$ compared to the CPU implementation of the WFA running on Sargantana, the in-order single-threaded RV64G RISC-V CPU of the chip, which also runs at 1.1GHz.

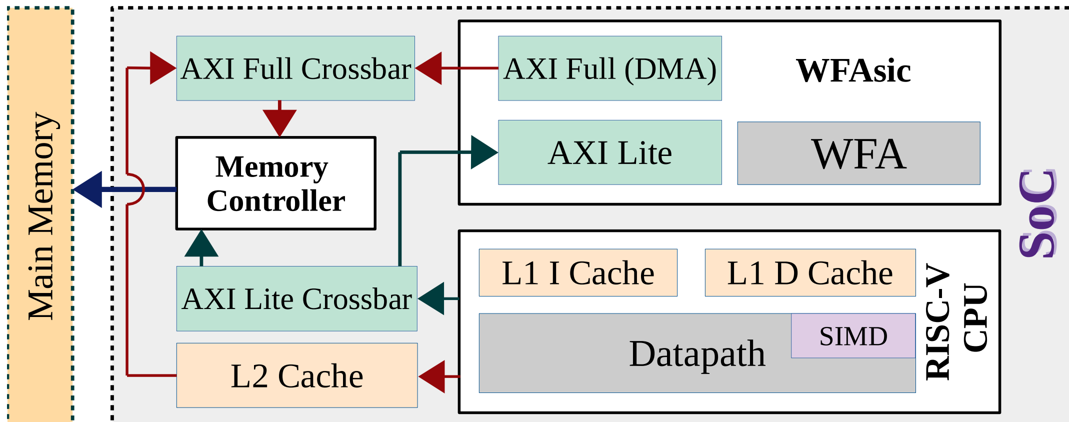


Figure 6.1: SoC architecture including the RISC-V CPU, the WFAsic accelerator and their connections.

6.2 System on Chip Architecture

The architecture of the SoC, including the WFAsic accelerator, the CPU and the intra-chip connections, is illustrated in Figure 6.2. The CPU communicates with the WFAsic accelerator through the AXI-Lite bus. The WFAsic accelerator includes a set of memory mapped registers, and the CPU writes into this registers the configuration of the accelerator. The configuration includes the backtrace functionality (enabled or disabled), the maximum sequence length of the input set, and the DMA configurations, which consist of the address and the size of the input set in the main memory, and the address where results should be written to the main memory. The WFAsic accelerator also has two registers, Start and Idle, that communicate with the CPU through the AXI-Lite bus. The CPU triggers the start of the accelerator by writing to the Start register, and it checks the completion of the computation in the accelerator by polling the Idle register. A dedicated interrupt could also be enabled to signal the job completion to the CPU.

The WFAsic accelerator has direct access to the off-chip main memory through the memory controller via the AXI-Full bus. In contrast, the CPU can access the main memory in two different ways: (1) via the AXI-Lite bus and the memory controller, and (2) via the AXI-Full bus, the L2 cache, and the memory controller.

The processor used in the SoC is Sargantana [20], a 64 bit single-threaded in-order Linux-capable RISC-V CPU that implements the RV64G ISA. For accelerating domain-specific applications, it uses a Single Instruction Multiple Data (SIMD) unit and supports the vector instructions defined in the vector extension RVV 0.7.1. The CPU has a 7-stage pipeline that implements register renaming, out-of-order write-back, and a non-blocking memory pipeline.

It has two first level caches: an instruction cache of 16KB, and a non-blocking data cache of 32KB. The system also has a 512KB L2 cache outside the CPU.

6.3 WFAsic Accelerator

The WFAsic accelerator is based on the FPGA accelerator of the WFA for long reads described in Section 5.4. In this section we adapt the design for an SoC implementation, targeting the alignment of sequences generated with third generation sequencing technologies up to 10K bases.

Depending on the available area and resources, as explained in Section 5.4, the WFAsic accelerator can include multiple Aligner modules to align sequences in parallel. The Aligner module of the WFAsic accelerator remains almost the same as in WFA FPGA accelerator for long reads. However, Extractor and Collector modules are adapted to the new platform. These are the input and output modules that adapt input/output data formats compatible with the Aligner/SoC data formats. In the SoC the data width of the AXI-Full is 16 bytes, which is much smaller than that of the FPGA designs.

In the design of WFAsic, in order to be able to evaluate the accelerator design without being limited by memory-accelerator bandwidth, we add an option to enable and disable the backtrace functionality. We avoid transferring huge amount of data from accelerator to the memory by disabling backtrace, and in this case only the alignment score is calculated. Therefore, if backtrace is enabled, the Aligner computes the alignment scores and generates the backtrace data, otherwise it only computes the alignment scores. Figure 6.2 shows the WFAsic design and its Aligner.

6.3.1 Memory implementations

To implement the Input and Output FIFOs, and Input_Seq and Wavefront RAMs used in the FPGA accelerator (see Figure 6.2 (bottom)) in the ASIC, we have to use GlobalFoundries memory macros. There is only one input and one output FIFO. Both are *show ahead* FIFOs, in which the next data is available on the FIFO output port and is cleared by triggering the read request signal of the FIFO. These two FIFOs are the biggest memories in our design, with a width of 16 bytes (AXI-Full data width) and a depth of 256 words. We have used high-performance dual port register files to implement these FIFOs. To do this, we create a wrapper for these memories, which handles the internal pointers and read/write procedures to mimic the functionality of a show ahead FIFO for other modules (DMA, Extractor, and

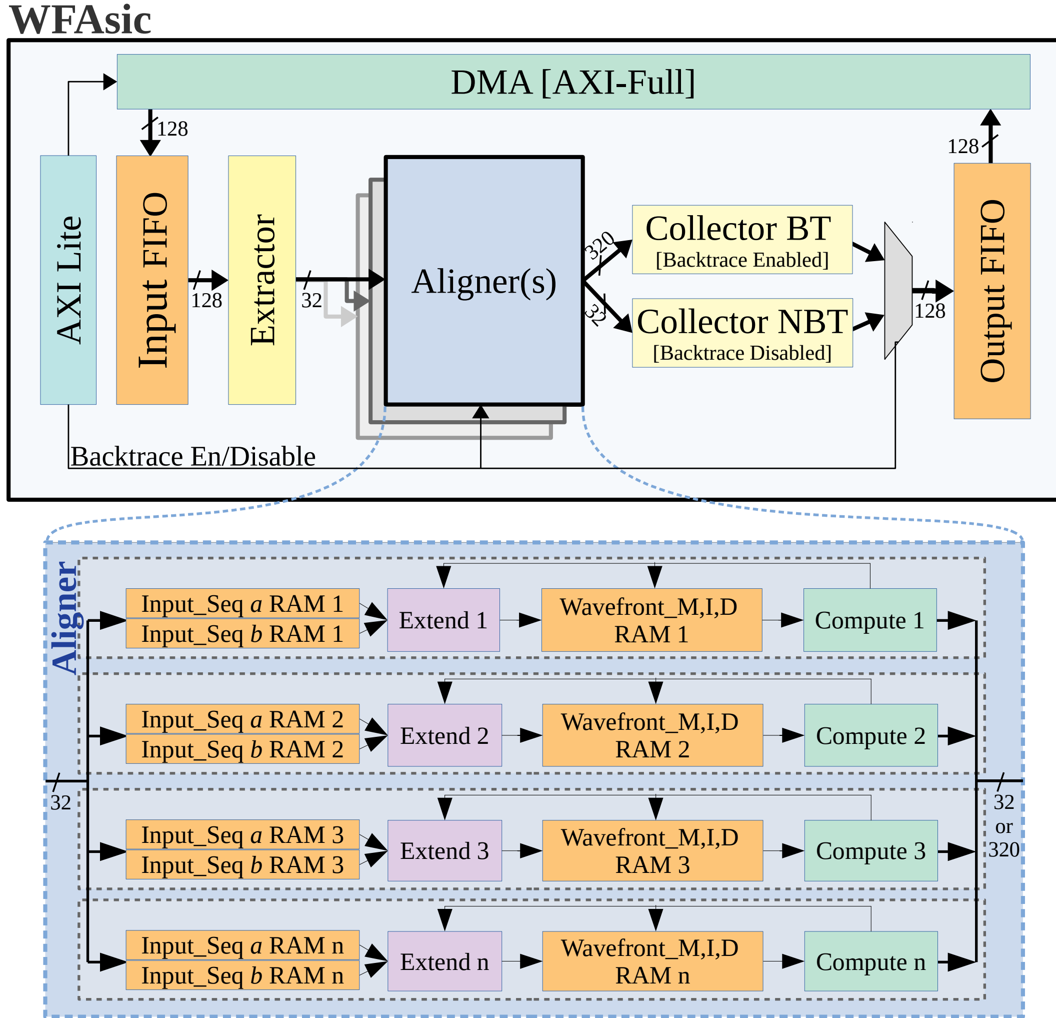


Figure 6.2: WFAsic structure and different modules.

Collector modules). Hence, the interactions of the modules with the Input/Output memories remain the same as in the FPGA accelerator. In addition, to increase the frequency, we have implemented single word show ahead FIFOs between DMA and Input FIFO, and Output FIFO and DMA.

Depending on the design configuration, each Aligner could have multiple sets of Extend and Compute sub-modules, called parallel sections. The number of parallel sections determines the number of input and wavefront RAMs of the Aligner. These RAMs in the design of the FPGA are implemented as dual port RAMs. one port for writing and another independent port for reading. The number of each type of these RAMs, i.e., Input_SEQ *a*, Input_SEQ *b*, Wavefront_M and Wavefront_ID, for a design of long reads is more than 16, and 64 in our case. Hence, there are a lot of dual port RAMs in our design that should be replaced by memory

macros. The large number of RAMs can reduce the frequency of the ASIC due to complicated routing and their maximum frequency limitations. To attain elevated frequencies for our ASIC accelerator, we select memory macros capable of achieving the utmost frequency, which in our case are high performance single port memory macros. In this case, again, to avoid changing the interaction protocols between RAMs and other modules, we design a wrapper that handles pointers and read/write procedures of a single port memory, but from the perspective of other modules, it looks like a dual port RAM.

6.3.2 Extractor Adaptation

The data width of the AXI-Full is 16 bytes, so the Extractor receives data (sequences to be aligned) in batches of 16 bases. Therefore, as soon as new data is received, it is stored in one row of the Input_Seq RAMs of the idle Aligner. At the beginning of a new sequence, the alignment ID and the sequences lengths are received, each in one batch of 16 bytes. However, they are stored in 4-byte rows of the Input_Seq RAMs. ID in address 0 of both types of Input_Seq RAMs, *a* and *b*, lengths in address 1 of their corresponding Input_Seq RAMs, and sequences bases from address 3 of their related Input_Seq RAMs.

6.3.3 Collector Adaptation

The Collector module must also provide data (alignment results) to the memory in batches of 16 bases. We design two *Collector* modules, each with different input widths to handle backtrace functionality. *Collector BT* is activated when backtrace is enabled and *Collector NBT* is activated when it is disabled.

If backtrace is enabled, the Aligner provides data in batches of 40 bytes. In the design for FPGA, the data width is 128 bytes, so the Collector merges three data of the Aligner, adds eight bytes of information, and sends data to the memory. However, in the WFAsic, the output data width is smaller than the Aligner data width. Hence, each output data of the Aligner should be divided by 16 bytes and sent in multiple memory transactions. Here two problems arise. First, the Aligner pause time increases compared to the FPGA accelerator. This is because the Aligner is not allowed to generate new data before the previous data is sent to the memory by the Collector. The memory bandwidth is decreased, and hence the Aligner pause time increases. The second problem is that the overhead increases. 40-byte data of the Aligner fits in three 16-byte transactions. Still, when we divide it to be sent in multiple transactions, each part of the split data requires attached information to be identifiable later in the CPU. Therefore, as

shown in Figure 6.3 (A), in each transaction, we combine 10 bytes of the Aligner backtrace data with six bytes of information in one block of 16 bytes and send each Aligner backtrace data to the memory in four transactions. In the Figure 6.3 (A) it is assumed that there is only one Aligner in the design, otherwise the data of different Aligners are sent randomly to the memory (as in Figure 5.15).

When backtrace is enabled, the last data that the Aligner provides to the Controller BT is the alignment result, which is identifiable by the Last flag. The Aligner sends the alignment result to the Controller BT in 40 bytes, but only seven bytes of it are useful and 10 bytes of it are sent to the memory. Unlike the backtrace data, the alignment result is sent to the memory in one transaction. Figure 6.3 (B) shows the format of the alignment result field. In addition to Success flag and error score, it includes the k_{max} and *Length_difference* required for performing the backtrace.

The accelerator computes the backtrace data of 64 cells of the three wavefront vectors of $\widetilde{M}$, $\widetilde{I}$, and $\widetilde{D}$ wavefronts and concatenates them in a backtrace block of 320 bits. However, it divides each block of backtrace data into four 80-bit sub-blocks of A, B, C, and D. Figure 6.3 (C) depicts the format of a 320-bit backtrace block. The origin of each cell of $\widetilde{M}$, $\widetilde{I}$, and $\widetilde{D}$ wavefront vectors are encoded in 3, 1, and 1 bits, respectively. The first 192 bits (3×64) of each backtrace block include the origins of 64 cells of the $\widetilde{M}$ wavefront. The next 64 bits include the origins of 64 cells of the $\widetilde{I}$ wavefront, and the last 64 bits include the origins of 64 cells of the $\widetilde{D}$ wavefront. In a backtrace data block, the origins of cells 0 to 63 are positioned from left to right. For example, in a backtrace block, the origin of cells 0 and 1 of $\widetilde{M}$ wavefront are located at bits 0 to 2, and 3 to 5, respectively, which are located in sub-block A. The origin of cells 0 and 1 of $\widetilde{I}$ wavefront are located at bits 192 and 193, respectively, which are located in sub-block C. Similarly, the origin of cells 0 and 1 of $\widetilde{D}$ wavefront are located at bits 256 and 257, respectively, which are located in sub-block D.

If backtrace is disabled, the Aligner only provides the alignment results to the Controller NBT in four bytes. These four bytes include the Success flag in one bit, the alignment error score in 15 bits, and the alignment ID in two bytes. Controller NBT merges the alignment results of four alignments and sends them to the memory in one transaction. The Controller NBT does not attach any extra information to this data as the Aligner adds all the necessary information in four bytes, as explained. This way, the design is less limited by the memory bandwidth, and Aligner(s) pause time decreases significantly compared to when backtrace is enabled.

When the accelerator job is finished, the accelerator writes into the MMIO registers (explained in Section 3.1.1) the amount of 16-byte results data it has sent to the memory. Regardless of the backtrace functionality, the accelerator sends three 16-byte debug data at the end of all alignments. The amount of results data, that the accelerator reports, also includes these three blocks of debug data.

6.3.4 Backtrace

If the backtrace is enabled, as in the design of the FPGA, the CPU computes it when the alignment in the accelerator is finished. The backtrace data generated by the Compute submodule is stored in memory. The backtrace operation starts from the backtrace data of the last cell, which determines the alignment score. The backtrace procedure is the same as in the FPGA design as shown in Figure 5.16. However, in the WFAsic, the layout of storing backtrace data in memory is different from the FPGA design. The backtrace data is now divided into groups of 10 bytes with information data of six bytes between them. So when separating data, the information data are removed and sub-blocks of each backtrace block are written consecutively in memory.

However, if there is only one Aligner in the ASIC accelerator, the data separation is unnecessary, as the alignments data (alignment result and backtrace data) are written consecutively in the memory (as shown in Figure 6.3 (A)). The only important matter is determining the data boundaries of each alignment ID, and handling the gaps between backtrace sub-blocks correctly. We implement a method that identifies these boundaries and performs the backtrace of each alignment ID sequentially, as the CPU is single-threaded. Hence, the CPU code includes both single-Aligner and multi-Aligner backtrace computation methods.

Unlike in the FPGA design format, the Success flag of each alignment, which determines a successful alignment in the accelerator for each ID, is written in alignment result block. The alignment result format is shown in Figure 6.3 (B). Figure 6.3 (C) indicates how each backtrace block is divided in to 4 sub-blocks. The definition of each field is exactly the same as explained in Section 5.4.4 for the WFA-FPGA accelerator.

A

	80 bits	24 bits	1 bit	23 bits
	Backtrace Data **Alignment Result**	Block Counter	Last Flag	ID
Results Address	BT Data = 0A	0	0	1
	BT Data = 0B	1	0	1
	BT Data = 0C	2	0	1
	BT Data = 0D	3	0	1
	...			
	BT Data = n_1A	$4n_1$	0	1
	BT Data = n_1B	$4n_1$+1	0	1
	BT Data = n_1C	$4n_1$+2	0	1
	BT Data = n_1D	$4n_1$+3	0	1
	Alignment Result	$4n_1$+4	**1**	1
	...			
	BT Data = 0A	0	0	*m*
	...			
	BT Data = n_mD	$4n_m$+3	0	*m*
Results Address+*n-16*	**Alignment Results**	$4n_m$+4	**1**	*m*
Results Address+*n*	Debug Data 1			
Results Address+*n*+16	Debug Data 2			
Results Address+*n*+32	Debug Data 3			

B

Alignment Result Format (80 bits)						
24 bits	16 bits	16 bits	16 bits	5 bits	1 bit	2 bits
0	Length difference	k_{max}	Error Score	0	Success Flag	1

C

Backtrace Data Format (blocks of 64 cells of 3 wavefront vectors [concatenated in 320 bits])				
	64x1 bits	64x1 bits	64x3 bits	
BT Data = *x*{D,C,B,A}	D wavefront Origins	I wavefront Origins	M wavefront Origins	
	319 240	239 160	159 80	79 0
	*x*D	*x*C	*x*B	*x*A

Figure 6.3: The format of writing the alignment and backtrace data in memory for the WFAsic.

Figure 6.4: Accelerator layout. The size is 1330um×1200um with all the connectivity on the right side.

6.4 Evaluation

6.4.1 ASIC Synthesis and Place and Route

The design parameters of the WFAsic accelerator can be configured to fulfill the desired design constraints. Due to the area restrictions of our ASIC, we configure the WFAsic accelerator with one Aligner module, 64 Extend and Compute sub-modules (parallel sections = 64), and support for input reads with lengths up to 10Kbp and error rates up to 10%.

The tools of synthesis, place and route, and verification are described in Chapter 3. In the post-synthesis netlist, the WFAsic accelerator reaches a frequency of 1.5GHz and requires an area of 1.107mm^2 (1.057mm^2 is cell area and 0.05mm^2 is net area). In the post-PnR netlist, the WFAsic accelerator reaches a frequency of 1.1GHz in typical corner with 0.8V supply and at 85°C, and it has a power consumption of 312mW. Figure 6.4 shows the layout of the WFAsic accelerator in the GF22FDX technology. The WFAsic accelerator occupies an area of

1.6mm^2 and uses 0.48MB of memory. The memories are implemented as register file memory macros. There are 260 memory macros that occupy 85% of the area. Two of them with a size of 256×128 are input and output FIFOs. Input sequences, M wavefront and I/D wavefront data are stored in 128 640×32, 66 640×15 and 64 256×30 memory macros, respectively.

6.4.2 FPGA Prototype Performance Results

We compare the performance of the WFAsic accelerator with a publicly available C implementation of the WFA [200] executed on the RISC-V CPU of the SoC. The comparison is done on an FPGA prototype of the chip. The performance is measured in clock cycles, regardless of the FPGA frequency. In the FPGA prototype, both the core and accelerator are running at the same frequency of 50MHz. However, in ASIC, they both achieve a frequency of 1.1GHz.

We evaluate the WFAsic accelerator with six different input sets of Table 3.1. Although the accelerator is designed for long sequences, we evaluate its performance for short (100bp), medium (1Kbp) and long (10Kbp) sequences with error rates of 5% and 10%. We generate synthetic input sets with random mismatches, insertions and deletions, using the same methodology as in [19, 173].

Figure 6.5 shows the speedup of the WFAsic accelerator with and without calculating the backtrace with respect to the execution of the CPU scalar code. The figure also compares the CPU vector code with the scalar code. The vector register size is 128 bits. The extend step is vectorized in a way that it compares 16 bases in parallel. Also, the compute step is vectorized using eight 16-bit elements per vector. Our accelerator achieves speedups over the CPU scalar code of 143× to 1076× without performing the backtrace, and of 2.8× to 344× when performing the backtrace. The next paragraphs explain the reasons of the different speedups obtained for the different read lengths and after enabling and disabling the computation of backtrace.

Figure 6.6 shows the scalability of the WFAsic accelerator with different numbers of Aligners over a design with only one Aligner. The available resources in the FPGA prototype are larger than in the final chip, so we can fit multiple Aligners and evaluate the scalability of the WFAsic accelerator on the FPGA. To this end, first we disable the backtrace to avoid memory bandwidth limitations. Results show that, for input sets with long sequences, the design scales perfectly. In particular, for the input sets of 10K-10% and 10K-5%, the accelerator with 10 Aligners provides speedups of 9.87× and 9.67× over the accelerator with one Aligner, respectively. This represents speedups of 10621× and 10062× over the WFA-CPU scalar code, respectively. The speedup is saturated with less Aligners for inputs with short sequences and

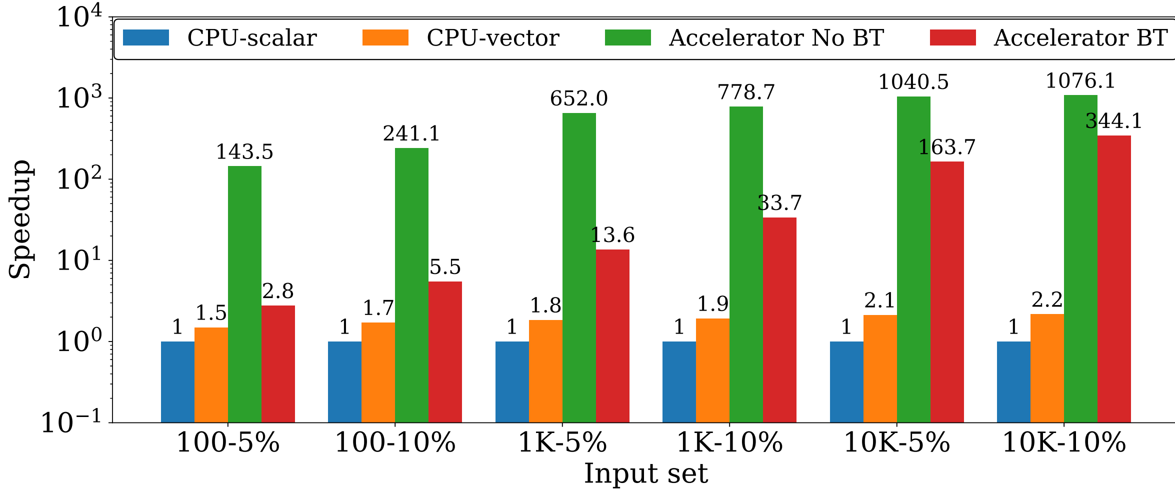


Figure 6.5: Speedup with respect to the CPU-scalar code.

smaller error rates. This is because the design is bound to the accelerator memory bandwidth for these inputs. This justification confirms the lower speedups of inputs with short lengths in Figure 6.5.

Next we explain the accelerator memory bandwidth restrictions, especially for short sequences, when having more than one Aligner in the accelerator. Table 6.1 shows, for each input, how many clock cycles are required to read a pair of sequences from main memory and to perform the alignment of the pair of sequences. Note that, first, the pairs of sequences are stored in the RAMs of the Aligners, and then the Aligners compute the alignments in parallel. Using Equation 6.1, the maximum efficient number of Aligners for each input set is calculated and shown in the last column of Table 6.1. For example, the design of 100-5% does not scale further than four Aligners because reading four pairs of sequences (4×75=300) takes more time than computing the four alignments in parallel (214 alignment cycles + 75 reading cycles = 289). Increasing the accelerator memory bandwidth would reduce the time for reading the sequences and, thus, improve the scalability of the designs for short reads.

$$MaxAligners = Roundup(\frac{Alignment_cycles}{Reading_cycles}) + 1 \tag{6.1}$$

When the backtrace is enabled, first the accelerator performs the alignment and then the CPU performs the backtrace. The backtrace time on the CPU dominates the total execution time, as it is much higher than the accelerator alignment time. This situation does not take place in the CPU executions of the WFA because the computations of the alignments is much

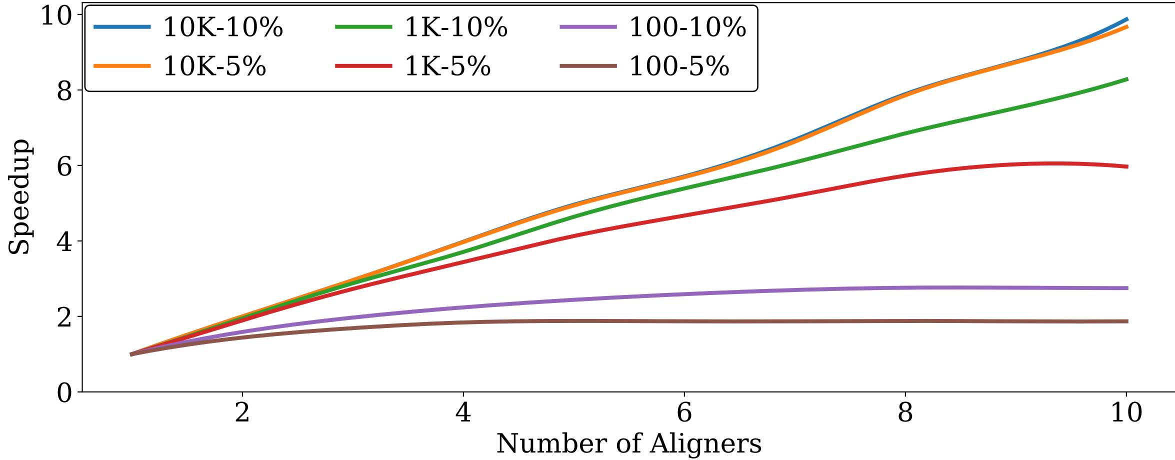


Figure 6.6: Speedup of adding Aligners with respect to one Aligner.

Table 6.1: Maximum number of Aligners for each input based on the execution cycles of reading and aligning one pair of reads.

Input		Alignment Cycles	Reading Cycles	Max Efficient Aligners
Length	Error Rate (%)			
100	5	214	75	4
100	10	327	75	6
1K	5	2541	376	8
1K	10	8461	376	24
10K	5	278083	3420	83
10K	10	937630	3420	276

slower. More importantly, the backtrace computation on the CPU is bound to the CPU memory bandwidth, which quickly becomes saturated. For these reasons, the speedups achieved by the WFAsic accelerator when the backtrace is disabled are higher than when the backtrace is enabled.

As mentioned earlier, due to area restrictions we are only able to fit one Aligner with 64 parallel sections in the WFAsic design. However, it is also possible to reduce the Aligner size by reducing the number of parallel sections and fit two smaller Aligners with 32 parallel sections in the chip. We have chosen the best configuration by comparing the performance of different configurations, WFAsic with one Aligner of 64 parallel sections versus WFAsic with two Aligners of 32 parallel sections. Note that as explained in Section 6.3.4, if there is one Aligner in the design, the time-consuming step of separating data of different alignments is not needed. The performance results shown in Figure 6.7 also compares both backtrace methods for the design with one Aligner and 64 parallel sections.

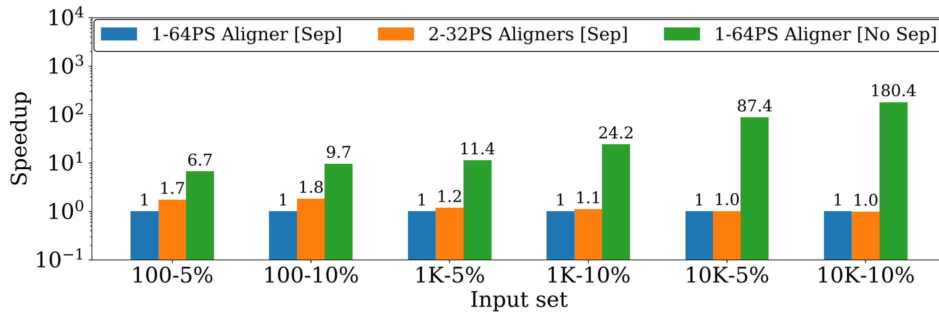


Figure 6.7: Performance comparison between WFAsic with one Aligner of 64 parallel sections (64PS) performing data separation (Sep), two Aligners of 32 parallel sections (32PS) performing data separation (Sep), and one Aligner of 64 parallel sections (64PS) without performing data separation (No Sep).

Figure 6.7 shows that by eliminating data separation step in 1-64PS Aligner design, it outperforms other configurations for all inputs, specially longer inputs. Comparing two other configurations which perform data separation, the design with two Aligners and 32 parallel sections, performs better for shorter reads. However, for longer reads the performance improvement is negligible. In principle, doubling the number of parallel section doubles the execution time. While in the case of very short reads this is not true. This is because for short reads the wavefront matrix is very small and most of the parallel sections are idle. So increasing parallel sections does not improve performance while increasing number of Aligners does.

We chose to have one Aligner with 64 parallel section, in which the backtrace method does not separate data of different alignments. Apart from the big performance improvement we get with this configuration, below is listed other reasons justifying why the design with one Aligner and 64 parallel sections, regardless of the backtrace method, is better than the one with two Aligners and 32 parallel sections.

- One Aligner with 32 parallel sections is only 1.5× smaller than one Aligner with 64 parallel sections. So using two Aligners with 32 parallel sections require more area than one Aligner with 64 parallel sections.

- Having one Aligner is more simple in terms of distributing inputs to, and collecting outputs from, the Aligner.

- Our target in this project is third generation sequencing technologies which provide longer reads where both accelerator designs, when performing data separation, perform equally fast.

Table 6.2 compares the GCUPS, the area, and the GCUPS per mm^2 of the WFAsic accelerator with other methods/platforms when aligning reads of 10Kbp.

Darwin is the state-of-the-art accelerator that uses a heuristic method which does not process the whole DP-matrix, but some tiles of it. Hence, the CUPS achieved by Darwin is calculated based on the peak performance reports of the tiles computations (20.8M tiles/sec) and the tile size (320×320) in the original paper [174]. The AMD EPYC is a high-end server-class processor with 64 cores. The table shows the GCUPS obtained when running the CPU implementation of the WFA algorithm on the AMD EPYC processor with 1 and 64 threads. The GCUPS of the WFAsic accelerator on the ASIC is estimated by scaling the cycle counts measured on the FPGA prototype to the ASIC frequency. For example, if the cycle count for aligning sequences in the FPGA prototype is 1.1 billion cycles, its execution time on an ASIC with a frequency of 1.1GHz is estimated to be one second. The GCUPS of WFA algorithm on AMD EPYC, GPU and ASIC are calculated for input sets with reads of length 10K and error rate of 5%. Note that the WFA algorithm running on the AMD EPYC, GPU and on the WFAsic avoids the full computation of the DP-matrix, but as this algorithm is an exact method, in Table 6.2, we compute the CUPS considering the equivalent number of DP cells that the SWG algorithm would need to compute the optimal alignment.

The last column of Table 6.2 shows GCUPS per mm^2, in which we can see that the WFAsic accelerator (with and without backtrace) outperforms Darwin, WFA-GPU and the CPU implementation of the WFA running on the high-end AMD EPYC processor. Note that Darwin achieves the highest total GCUPS, but with a much larger area budget than WFAsic. The AMD EPYC processor is the slowest of the three, both in terms of absolute GCUPs and of GCUPS per mm^2.

Table 6.3 compares the GCUPS per aligner of WFAsic and WFA-FPGA. As in the previous Section 5.5.5, the column *SWG Equivalent* shows the GCUPS considering the equivalent number of DP-matrix cells that SWG should calculate to do the alignment, while the column *Computed* shows the GCUPS considering the average actual number of computed wavefront vector cells by the WFA. For WFA-FPGA for short reads, we compare against the design which achieves the best GCUPS. However, for WFA-FPGA for long reads, we compare against the design with the same configuration as the WFAsic. Note that the computation of a WFA wavefront vector cell is more costly than the computation of a SWG DP-matrix cell because the former requires two operations, compute and extend, while the latter only requires the compute operation.

Table 6.2: GCUPS and area comparison of different platforms/methods aligning reads of 10Kbp.

Platform	**GCUPS**	**Area (mm^2)**	**GCUPS per mm^2**
Darwin [GACT[1] - heuristic]	2129	85.6	25
WFA-GPU[130] [NVIDIA GeForce 3080]	476	628	0.076
AMD EPYC[2] [WFA 1 thread]	7.5	1008	0.0074
AMD EPYC[2] [WFA 64 threads]	98	1008	0.0972
WFAsic [without backtrace]	390	1.6	244
WFAsic [with backtrace]	61	1.6	38

[1] Darwin is a full mapper. The GACT module of Darwin is responsible for the pairwise alignment, which is the focus of our work.
[2] The AMD EPYC processor contains 8 Core Complex Dies (CCDs) and a central I/O Die (IOD) [**CTE-AMD**]. The die size of each CDC is $74mm^2$ and that of IOD is $416mm^2$.

Table 6.3: GCUPS comparison of WFA-1FPGA and WFAsic per Aligner.

Platform	**GCUPS per Aligner**	
	SWG Equivalent	**Computed**
WFA-1FPGA (short reads: 300-3%)	22.7	0.2
WFA-1FPGA (long reads: 10K-5%)	48.5	1.1
WFAsic (without backtrace: 10K-5%)	390	8.8
WFAsic (with backtrace: 10K-5%)	61	1.4

6.5 Conclusions

This chapter presents the first WFA ASIC accelerator integrated in a RISC-V processor SoC. The accelerator is designed for long reads and evaluated for reads up to 10Kbp. WFAsic provides optimal results based on gap-affine scoring. The design is well parallelized by efficiently storing and distributing necessary data into multiple RAMs, so that 64 cells of the wavefront matrix are calculated in parallel. WFAsic is able to perform the alignment independently and in parallel with other CPU processes as it includes a DMA which has direct access to the main memory through the AXI-Full bus. Results show that the accelerator reaches speedups of up to 1076× and 344× when backtrace is disabled and enabled, respectively, over the CPU implementation of the WFA running on the RISC-V of the chip. In addition, the accelerator perfectly scales by increasing the number of the Aligners in the accelerator, if the accelerator-memory bandwidth is not saturated. The post-layout of the WFAsic, in GF22nm technology, reaches a frequency of 1.1GHz, fits in an area of $1.6mm^2$ and consumes a power of 312mW.

Chapter 7
Conclusions

This thesis has accelerated two important genomics application, k-mer counting and pairwise read alignment, using FPGAs and ASICs. The proposed accelerated designs have provided performance improvements as well as energy efficiency. This chapter details the main conclusions from the contributions of this thesis and then outlines the possibilities for future research they suggest. Finally, this chapter lists the publications resulting from this thesis and acknowledges the financial and technical support that made it possible.

7.1 Goals, Contributions and Main Conclusions

Although next generation and third generation sequencing technologies have significantly accelerated the pace of DNA sequencing, the process of DNA assembly remains a bottleneck in genomics research. Next generation sequencing technologies, such as Illumina, generate large volumes of short read data, while third generation technologies, such as Pacific Biosciences (PacBio), produce long read data. However, assembling these reads into a complete genome is still a computationally intensive process. To address this challenge, domain-specific genomics accelerators such as FPGAs and ASICs have been developed. These accelerators are specifically designed to handle the complex computations involved in DNA assembly, significantly reducing the time and cost involved in the process. By combining next generation and third generation sequencing technologies with specialized genomics accelerators, researchers can greatly accelerate their genomics research and make new discoveries that can lead to improvements in human health and medicine.

In the first contribution of this thesis we have targeted the k-mer counting application. In this work we proposed an FPGA-based hardware/software co-design to accelerate the k-mer counting, and implemented it in SMUFIN. SMUFIN is a genomics application for finding somatic mutations, and its first step is counting the frequency of k-mers appearance in all

reads of the dataset. This contribution not only includes designing an FPGA accelerator for k-mer counting, but also restructuring and modifying the SMUFIN C code to achieve better performance as well as using less memory. The software modifications consist of identifying and removing/modifying time-consuming functions, removing dependencies between parallel threads, reducing overheads and exploiting data compaction mechanisms. Compared to the baseline 160-threaded C implementation of the k-mer counting of SMUFIN, running on 40 cores of the POWER9 HPC machine, our co-design, using one Xilinx Virtex UltraScale+ (XCVU3P) FPGA running at 250MHz, is 2.14× faster while consuming 2.93× less energy and 1.57× less memory.

In the second contribution we have targeted the pairwise read alignment of short reads generated by next generation sequencing technologies. NGS technologies currently are the most widely used sequencing technologies in the market. In this work we designed the first WFA-FPGA accelerator for short reads, performing exact alignments based on the gap-affine WFA algorithm. Our design implements a hardware/software co-designed scheme, in which the FPGA accelerator by including multiple aligner cores, computes the alignment of multiple pairs of reads in parallel and performs the backtrace. However, in a novel approach it generates the backtrace results in a compacted form that eases CPU-FPGA communication. Then the CPU threads can unpack the compacted forms and achieve the final backtrace results in parallel. In the proposed design of the accelerator, the maximum supported read length and error score between the reads are configurable. It allows the design to be tailored to the characteristics of reads produced by various sequencing machines. By adjusting these two design parameters, the aligner resources can be optimized, which in turn, affects the number of parallel aligners that can be placed in the FPGA. We have evaluated our design with one and two FPGAs by applying seven different input sets with different read lengths of 100 to 300 bases and with different error rates of 3% to 8%. Compared to the 64-threaded WFA CPU-only implementation running on 16 cores of the POWER9 HPC machine, the FPGA accelerator, using Xilinx Virtex UltraScale Plus (XCVU37P) FPGAs running at 200MHz, achieves speedups of 4.5× to 8.8× with one FPGA, and of 8.2× to 13.5× with two FPGAs, while reducing the energy-to-solution by 6.1× to 9.7× with one FPGA, and by 11.4× to 14.6× with two FPGAs.

In the third contribution we have targeted the pairwise read alignment of long reads generated by third generation sequencing technologies. Third generation sequencing technologies are expected to be widely used in the future due to their developing benefits over the NGS technologies. In this work we have significantly modified the previous design of WFA-FPGA of short reads to make it suitable for long reads of up to 50K bases. To this end, to fit the

necessary data for doing the alignment of long reads in the FPGA, we have exploited on-chip FPGA RAMs instead of storing data in registers and using LUTs as in the design of short reads. We have intelligently distributed data in RAMs in a way that it maximizes the parallelism and minimizes the RAM usage. In addition, we also propose a reorganization of the tasks performed by the FPGA and by the CPU in the hardware/software co-design. In this design also the maximum supported read length, error score, and number of aligner are configurable. We have evaluated our design with one and two FPGAs by applying 15 different synthetic input sets and two real input sets. The length of the synthetic inputs ranges from 1K to 50K bases, and their error rates ranges from 5% to 20%. Compared to the 64-threaded WFA CPU-only implementation running on 16 cores of the POWER9 HPC machine, our co-design, using Xilinx Virtex UltraScale Plus (XCVU37P) FPGAs running at 200MHz, for synthetic input sets, achieves speedups of 2.6× to 5.6× with one FPGA and of 2.7× to 9.9× with two FPGAs, and energy-to-solution is reduced by 3.6× to 7.5× with one FPGA, and by 3.7× to 10.9× with two FPGAs. While, for real input sets, sequenced by PacBio machines, the accelerator design exhibits speedups of 3.0× to 3.9× and 5.6× to 6.6× with one and two FPGAs, respectively, and the energy improvements of 3.8× to 4.4× and 6.5× to 7.3× with one and two FPGAs, respectively.

In the forth contribution we have presented the first ASIC implementation of the WFA algorithm, WFAsic. This work presents the integration of the WFAsic accelerator in a Linux-capable RISC-V SoC. The WFAsic accelerator is configured using a standard Linux driver and API. In addition, the WFAsic accelerator runs as an independent process in parallel to other CPU processes. Integrating the WFAsic accelerator with the CPU in the same SoC provides great benefits to genomics applications, as it eliminates the need for external accelerators and their costly communication. Our WFAsic implementation includes one aligner core and supports read lengths up to 10Kbp and error rates up to 10%. However, these parameters are configurable depending on the available resources and input characteristics. After synthesis and PnR in GlobalFoundries 22nm technology, the WFAsic accelerator fits in an area of 1.6mm^2 and reaches a frequency of 1.1GHz. Although the WFAsic is designed for long reads, we have evaluated it with three input sets of length 100 to 10K bases and with error rates of 5% to 10%. The integrated WFAsic accelerator provides performance improvements of up to 1076× compared to the CPU implementation of the WFA running on Sargantana, the single-threaded 64 bit in-order Linux-capable RISC-V CPU of the chip running at the same frequency as the accelerator.

7.2 Future Work

The work presented in this thesis suggests many possible avenues for future work. Detailed below are four which stand out as of particular potential.

- In the first contribution of SMUFIN, the time-consuming step of generating Bloom filters is removed and the elimination of unique k-mers is postponed to the last step of the SMUFIN. By doing this, although we gained performance improvements, the application requires writing unnecessary data on disk and consequently using twice more disk space as needed. Hence, the performance is limited by I/O constraints and therefore the FPGA-CPU data transactions cannot benefit from a double buffering design. As a future work, removing unique k-mers intelligently before they are written to the disk, would remove the I/O constrains and push the performance more. To do this, unlike the original SMUFIN design, the Bloom filters generation is done at the same time as counting k-mers, instead of as prior step. In addition to k-mer generation, the time-consuming hash function of Bloom filters is offloaded to the FPGA, but the tables remain in the CPU as they do not fit in the FPGA memory. With this approach we aim at removing the I/O bottleneck, as less data needs to be written to disk, and enable a double buffering scheme that further improves performance.
- In second and third contributions we observed that the performance scales perfectly by increasing the number of FPGAs. Hence, implementing the WFA algorithm on a cluster of many FPGAs [209] can achieve significant performance improvements. Unlike traditional CPU-FPGA direct link connections, in this cluster the FPGAs and the CPUs can communicate with each other through a data center network. In this case, as both the resources and the bandwidth of each FPGA is smaller than in our FPGA platforms in our contributions, a smaller number of WFA cores could fit in each FPGA of the cluster. However, since the amount of FPGAs in the cluster is much larger (at least 64 FPGAs in the envisioned setup), a huge amount of parallelism can be achieved. This new perspective opens the door to aligning different reads against potential positions of the reference genome in parallel, or to minimize data transfers if all the FPGAs align the same one or two reads against different potential positions of the reference genome. This task requires to re-think the parallelization strategy of the previously proposed WFA accelerators for short and long reads, which in turn requires to re-think the whole communication and synchronization mechanisms between modules, as well as potentially

re-designing some modules to adapt them to the cluster characteristics while avoiding potential new bottlenecks.

- In the fourth contribution, our WFAsic accelerator is implemented in an SoC with one CPU core, without having access to the L2 cache. As a future work, the ASIC accelerator can be implemented in a multiple core SoC, with one accelerator per each core, and with direct connections to the L2 cache. This way multiple aligners work in parallel and receive input data faster as they read from L2 cache. In addition, the WFAsic could be modified based on the newly proposed improved WFA, WFA_bidirectional [210], in which less memory is required, and therefore WFAsic size reduces, and hence it occupies less area. These tasks require accelerator to implement the new methods of WFA_bidirectional and the cache coherency protocol.

- The WFA contributions have tested the WFA accelerator as an independent application, however the WFA could serve as the pairwise read aligner of a full mapper. Integrating the WFA accelerator in a full mapper would be interesting specially for the industrial section, as the improvements it may provide significantly changes the speed of read alignment. Implementing the WFA accelerator in a full mapper requires adapting and modifying the software code of the mapper in order to make it more suitable for a co-designed structure.

7.3 Publications

This section lists below the publications that resulted from the work on this thesis.

- Haghi A, Alvarez L, Polo J, Diamantopoulos D, Hagleitner C, Moreto M. A Hardware/-Software Co-Design of K-mer Counting Using a CAPI-Enabled FPGA. In 2020 30th International Conference on Field-Programmable Logic and Applications (FPL) 2020 Aug 31 (pp. 57-64). IEEE.

- Haghi A, Marco-Sola S, Alvarez L, Diamantopoulos D, Hagleitner C, Moreto M. An FPGA accelerator of the wavefront algorithm for genomics pairwise alignment. In 2021 31st International Conference on Field-Programmable Logic and Applications (FPL) 2021 Aug 30 (pp. 151-159). IEEE.

- Haghi A, Marco-Sola S, Alvarez L, Diamantopoulos D, Hagleitner C, Moreto M. WFA-FPGA: An efficient accelerator of the wavefront algorithm for short and long read genomics alignment. Future Generation Computer Systems. 2023 Dec 1;149:39-58.
- Haghi A, Alvarez L, Fornt J, de Haro Ruiz J.M, Figueras R, Doblas M, Marco-Sola S, Moreto M. WFAsic: A High-Performance ASIC Accelerator for DNA Sequence Alignment on a RISC-V SoC. In 2023 52nd International Conference on Parallel Processing (ICPP) 2023 Aug 7. ACM.

Author contributions to each paper are as follows:

- **Paper 1:**
  Abbas Haghi: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data Curation, Writing - Original Draft, Writing - Review & Editing.
  Lluc Alvarez: Conceptualization, Methodology, Writing - Original Draft, Writing - Review & Editing, Visualization, Supervision.
  Jordà Polo: Methodology, Software, Resources.
  Dionysios Diamantopoulos: Conceptualization, Methodology, Writing - Review & Editing, Visualization, Supervision.
  Christoph Hagleitner: Supervision, Project administration, Funding acquisition.
  Miquel Moreto: Methodology, Visualization, Supervision, Project administration, Funding acquisition.

- **Papers 2 & 3:**
  Abbas Haghi: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data Curation, Writing - Original Draft, Writing - Review & Editing.
  Lluc Alvarez: Conceptualization, Methodology, Writing - Original Draft, Writing - Review & Editing, Visualization, Supervision.
  Santiago Marco-Sola: Conceptualization, Methodology, Software, Resources, Supervision.
  Dionysios Diamantopoulos: Conceptualization, Methodology, Writing - Review & Editing, Visualization, Supervision.
  Christoph Hagleitner: Supervision, Project administration, Funding acquisition.
  Miquel Moreto: Methodology, Visualization, Supervision, Project administration, Funding acquisition.

- **Paper 4:**
  Abbas Haghi: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data Curation, Writing - Original Draft, Writing - Review & Editing.
  Lluc Alvarez: Conceptualization, Methodology, Writing - Original Draft, Writing - Review & Editing, Visualization, Supervision.
  Jordi Fornt: Support for GLS, Software, Review & Editing.
  Juan Miguel de Haro Ruiz: Support for FPGA prototyping, Review & Editing.
  Roger Figueras: Physical design, Review & Editing.
  Max Doblas: Support for the RISC-V core, Software.
  Santiago Marco-Sola: Conceptualization, Methodology, Software, Resources, Supervision.
  Miquel Moreto: Methodology, Visualization, Supervision, Project administration, Funding acquisition.

7.4 Financial and Technical Support

This thesis has been supported by the European HiPEAC Network of Excellence, by the Spanish Ministry of Science and Innovation MCIN/AEI/10.13039/501100011033 (contracts TIN2015-65316-P, PID2019-107255GB-C21 and TED2021-132634A-I00), by the Generalitat de Catalunya (contracts 2017-SGR-1328 and 2021-SGR-00763), by the European Union within the framework of the ERDF of Catalonia 2014-2020 under the DRAC project (contract 001-P-001723), by the European NextGenerationEU/PRTR, by the Lenovo-BSC Contract-Framework (2022), and by the IBM/BSC Deep Learning Center initiative.

Bibliography

[1] Karl Rupp. *50-year trends in microprocessors*. URL: https://github.com/karlrupp/microprocessor-trend-data/tree/master/50yrs.

[2] Darrell Boggs et al. "The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology." In: *Intel Technology Journal* 8.1 (2004).

[3] Nikhil Bansal, Kirk Pruhs, and Cliff Stein. "Speed scaling for weighted flow time". In: *SIAM Journal on Computing* 39.4 (2010), pp. 1294–1308.

[4] Roddy Urquhart. *Semiconductor Scaling is Failing*. White paper. March, 2022. URL: https://codasip.com/papers/semiconductor-scaling-is-failing/.

[5] William J Dally, Yatish Turakhia, and Song Han. "Domain-specific hardware accelerators". In: *Communications of the ACM* 63.7 (2020), pp. 48–57.

[6] Erin D Pleasance et al. "A comprehensive catalogue of somatic mutations from a human cancer genome". In: *Nature* 463.7278 (2010), pp. 191–196.

[7] André Lacour et al. "Genome-wide significant risk factors for Alzheimer's disease: role in progression to dementia due to Alzheimer's disease among subjects with mild cognitive impairment". In: *Molecular psychiatry* 22.1 (2017), pp. 153–160.

[8] Yangrae Cho et al. "Prevalence of rare genetic variations and their implications in NGS-data interpretation". In: *Scientific reports* 7.1 (2017), p. 9810.

[9] Niklas Krumm et al. "Excess of rare, inherited truncating mutations in autism". In: *Nature genetics* 47.6 (2015), pp. 582–588.

[10] Olga Spichenok et al. "Prediction of eye and skin color in diverse populations using seven SNPs". In: *Forensic Science International: Genetics* 5.5 (2011), pp. 472–478.

[11] Chimpanzee Sequencing and Analysis Consortium Waterson Robert H. waterston@ gs. washington. edu Lander Eric S. lander@ broad. mit. edu Wilson Richard K. rwilson@ watson. wustl. edu. “Initial sequence of the chimpanzee genome and comparison with the human genome”. In: *Nature* 437.7055 (2005), pp. 69–87.

[12] Cory Y McLean et al. “Human-specific loss of regulatory DNA and the evolution of human-specific traits”. In: *Nature* 471.7337 (2011), pp. 216–219.

[13] Margaret A Hamburg and Francis S Collins. “The path to personalized medicine”. In: *New England Journal of Medicine* 363.4 (2010), pp. 301–304.

[14] Sahand Salamat and Tajana Rosing. “FPGA Acceleration of Sequence Alignment: A Survey”. In: *arXiv preprint arXiv:2002.02394* (2020).

[15] *NovaSeq 6000 Sequencing System.* URL: https://www.illumina.com/systems/sequencing-platforms/novaseq.html.

[16] Mohammed Alser et al. “Accelerating genome analysis: a primer on an ongoing journey”. In: *IEEE Micro* 40.5 (2020), pp. 65–75.

[17] *DNA Sequencing Costs: Data.* URL: https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data.

[18] Valentıé Moncunill et al. “Comprehensive characterization of complex structural variations in cancer by directly comparing genome sequence reads”. In: *Nature biotechnology* 32.11 (2014), p. 1106.

[19] Santiago Marco-Sola et al. “Fast gap-affine pairwise alignment using the wavefront algorithm”. In: *Bioinformatics* 37.4 (2021), pp. 456–463.

[20] Vıéctor Soria-Pardos et al. “Sargantana: A 1 GHz+ In-Order RISC-V Processor with SIMD Vector Extensions in 22nm FD-SOI”. In: *2022 25th Euromicro Conference on Digital System Design (DSD)*. IEEE. 2022, pp. 254–261.

[21] *Factsheet: Deoxyribonucleic Acid (DNA)*. national human genome research institute. URL: https://www.genome.gov/about-genomics/fact-sheets/Deoxyribonucleic-Acid-Fact-Sheet.

[22] Santiago Marco-Sola. “Efficient approximate string matching techniques for sequence alignment”. PhD thesis. Universitat Politècnica de Catalunya (UPC), 2017.

[23] Tobias Pascal Loka. “Advanced Strategies for Alignment-based Real-time Analysis and Data Protection in Next-Generation Sequencing”. PhD thesis. Freien Universität Berlin, 2020.

[24] *The Genome Reference Consortium.* URL: https://www.ncbi.nlm.nih.gov/grc/human.

[25] Bonnie Berger, Noah M. Daniels, and YU Y.WILLIAM. "Algorithm advances take advantages of the structure of massive biological data landscape". In: *Commun Can* 59.8 (2016), pp. 71–78.

[26] Boluwatife A Adewale. "Will long-read sequencing technologies replace short-read sequencing technologies in the next 10 years?" In: *African journal of laboratory medicine* 9.1 (2020), pp. 1–5.

[27] Nathaniel S McVicar. "FPGA Accelerated Bioinformatics: Alignment, Classification, Homology and Counting". PhD thesis. University of Washington, 2018.

[28] *Illumina. Measuring sequencing accuracy.* URL: https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/quality-scores.html.

[29] David Sims et al. "Sequencing depth and coverage: key considerations in genomic analyses". In: *Nature Reviews Genetics* 15.2 (2014), pp. 121–132.

[30] Eric S Lander et al. "Initial sequencing and analysis of the human genome". In: *nature* 409.6822 (2001), pp. 860–921.

[31] *Illumina. Coverage depth recommendations.* URL: https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/coverage.html.

[32] *Your Essential Guide to Different File Formats in Bioinformatics.* URL: https://www.formbio.com/blog/your-essential-guide-different-file-formats-bioinformatics.

[33] Xingyu Liao et al. "Current challenges and solutions of de novo assembly". In: *Quantitative Biology* 7 (2019), pp. 90–109.

[34] Abdul Rafay Khan et al. "A comprehensive study of de novo genome assemblers: current challenges and future prospective". In: *Evolutionary Bioinformatics* 14 (2018), p. 1176934318758650.

[35] Sergey Koren et al. "Hybrid error correction and de novo assembly of single-molecule sequencing reads". In: *Nature biotechnology* 30.7 (2012), pp. 693–700.

[36] Xiaoqiu Huang et al. "PCAP: a whole-genome assembly program". In: *Genome research* 13.9 (2003), pp. 2164–2170.

[37] Todd J Treangen et al. "Next generation sequence assembly with AMOS". In: *Current Protocols in Bioinformatics* 33.1 (2011), pp. 11–8.

[38] David B Jaffe et al. “Whole-genome sequence assembly for mammalian genomes: Arachne 2”. In: *Genome research* 13.1 (2003), pp. 91–96.

[39] Eugene W Myers et al. “A whole-genome assembly of Drosophila”. In: *Science* 287.5461 (2000), pp. 2196–2204.

[40] Jang-il Sohn and Jin-Wu Nam. “The present and future of de novo whole-genome assembly”. In: *Briefings in bioinformatics* 19.1 (2018), pp. 23–40.

[41] Rayan Chikhi and Guillaume Rizk. “Space-efficient and exact de Bruijn graph representation based on a Bloom filter”. In: *Algorithms for Molecular Biology* 8.1 (2013), pp. 1–9.

[42] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. “An Eulerian path approach to DNA fragment assembly”. In: *Proceedings of the national academy of sciences* 98.17 (2001), pp. 9748–9753.

[43] Daniel R Zerbino and Ewan Birney. “Velvet: algorithms for de novo short read assembly using de Bruijn graphs”. In: *Genome research* 18.5 (2008), pp. 821–829.

[44] Mark J Chaisson, Dumitru Brinza, and Pavel A Pevzner. “De novo fragment assembly with short mate-paired reads: Does the read length matter?” In: *Genome research* 19.2 (2009), pp. 336–346.

[45] Jonathan Butler et al. “ALLPATHS: de novo assembly of whole-genome shotgun microreads”. In: *Genome research* 18.5 (2008), pp. 810–820.

[46] Jared T Simpson et al. “ABySS: a parallel assembler for short read sequence data”. In: *Genome research* 19.6 (2009), pp. 1117–1123.

[47] Shaun D Jackman et al. “ABySS 2.0: resource-efficient assembly of large genomes using a Bloom filter”. In: *Genome research* 27.5 (2017), pp. 768–777.

[48] Yu Peng et al. “IDBA–a practical iterative de Bruijn graph de novo assembler”. In: *Annual international conference on research in computational molecular biology*. Springer. 2010, pp. 426–440.

[49] Binghang Liu et al. “Estimation of genomic characteristics by analyzing k-mer frequency in de novo genome projects”. In: *arXiv preprint arXiv:1308.2012* (2013).

[50] Sebastian Deorowicz, Agnieszka Debudaj-Grabysz, and Szymon Grabowski. “Disk-based k-mer counting on a PC”. In: *BMC bioinformatics* 14.1 (2013), p. 160.

[51] Robert C Edgar. "MUSCLE: multiple sequence alignment with high accuracy and high throughput". In: *Nucleic acids research* 32.5 (2004), pp. 1792–1797.

[52] Guillaume Marçais and Carl Kingsford. "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers". In: *Bioinformatics* 27.6 (2011), pp. 764–770.

[53] Pall Melsted and Jonathan K Pritchard. "Efficient counting of k-mers in DNA sequences using a bloom filter". In: *BMC bioinformatics* 12.1 (2011), p. 333.

[54] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. "DSK: k-mer counting with very low memory usage". In: *Bioinformatics* 29.5 (2013), pp. 652–653.

[55] Yang Li and Xifeng Yan. "MSPKmerCounter: a fast and memory efficient approach for k-mer counting". In: *arXiv preprint arXiv:1505.06550* (2015).

[56] Rajat Shuvro Roy, Debashish Bhattacharya, and Alexander Schliep. "Turtle: Identifying frequent k-mers with cache-efficient algorithms". In: *Bioinformatics* 30.14 (2014), pp. 1950–1957.

[57] Sebastian Deorowicz et al. "KMC 2: fast and resource-frugal k-mer counting". In: *Bioinformatics* 31.10 (2015), pp. 1569–1576.

[58] Stephen F Altschul et al. "Basic local alignment search tool". In: *Journal of Molecular Biology* 215.3 (1990), pp. 403–410.

[59] Heng Li. "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM". In: *arXiv preprint arXiv:1303.3997* (2013).

[60] Heng Li. "Minimap2: pairwise alignment for nucleotide sequences". In: *Bioinformatics* 34.18 (2018), pp. 3094–3100.

[61] Santiago Marco-Sola et al. "The GEM mapper: fast, accurate and versatile alignment by filtration". In: *Nature methods* 9.12 (2012), p. 1185.

[62] Santiago Marco-Sola and Paolo Ribeca. "Efficient Alignment of Illumina-Like High-Throughput Sequencing Reads with the GEnomic Multi-tool (GEM) Mapper". In: *Current Protocols in Bioinformatics* 50.1 (2015), pp. 11–13.

[63] *Lecture Notes in Sequence Alignment.* URL: http://www.cs.cmu.edu/~durand/03-711/2017/Lectures/Sequence-Alignment-2017.pdf.

[64] Ankit Agrawal and Xiaoqiu Huang. "Pairwise statistical significance of local sequence alignment using multiple parameter sets and empirical justification of parameter set change penalty". In: *BMC bioinformatics*. Vol. 10. Springer. 2009, pp. 1–9.

[65] Vladimir I Levenshtein et al. “Binary codes capable of correcting deletions, insertions, and reversals”. In: *Soviet physics doklady*. Vol. 10. 8. Soviet Union. 1966, pp. 707–710.

[66] Saul B Needleman and Christian D Wunsch. “A general method applicable to the search for similarities in the amino acid sequence of two proteins”. In: *Journal of molecular biology* 48.3 (1970), pp. 443–453.

[67] Temple F Smith, Michael S Waterman, et al. “Identification of common molecular subsequences”. In: *Journal of Molecular Biology* 147.1 (1981), pp. 195–197.

[68] Osamu Gotoh. “An improved algorithm for matching biological sequences”. In: *Journal of Molecular Biology* 162.3 (1982), pp. 705–708.

[69] Ho-Cheung Ng. “FPGA acceleration of DNA sequence alignment: design analysis and optimization”. PhD thesis. Imperial College London, 2021.

[70] Graham Singer. *The History of the Modern Graphics Processor.* December, 2022. URL: https://www.techspot.com/article/650-history-of-the-gpu/.

[71] Peter N Glaskowsky. *NVIDIA’s Fermi: the first complete GPU computing architecture*. White paper. 2009.

[72] Gabriel Campeanu. “GPU Support for Component-based Development of Embedded Systems”. PhD thesis. Mälardalen University, 2018.

[73] *NVIDIA A100 Tensor Core GPU Architecture*. Tech. rep. NVIDIA, 2020. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[74] *CUDA Refresher: The CUDA Programming Model.* June, 2020. URL: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/.

[75] Dion Harris. *NVIDIA Hopper GPU Architecture.* March, 2022. URL: https://blogs.nvidia.com/blog/2022/03/22/nvidia-hopper-accelerates-dynamic-programming-using-dpx-instructions/.

[76] *NVIDIA H100 Tensor.* URL: https://www.nvidia.com/en-us/data-center/h100/.

[77] Luca Cadenelli. “Hardware/software co-design for data-intensive genomics workloads”. PhD thesis. Universitat Politècnica de Catalunya, 2019.

[78] Quim Aguado-Puig et al. “Accelerating Edit-Distance Sequence Alignment on GPU Using the Wavefront Algorithm”. In: *IEEE access* 10 (2022), pp. 63782–63796.

[79] Mark LaPedus. *In-Memory Vs. Near-Memory Computing.* Februery, 2019. URL: https://semiengineering.com/in-memory-vs-near-memory-computing/.

[80] Shaahin Angizi. "Processing-in-memory for data-intensive applications, from device to algorithm". PhD thesis. Arizona State University, 2021.

[81] Rahul Awati. *Processing in memory.* URL: https://www.techtarget.com/searchbusinessanalytics/definition/processing-in-memory-PIM.

[82] *In-Memory Processing.* URL: https://hazelcast.com/glossary/in-memory-processing/.

[83] Ping Chi et al. "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory". In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 27–39.

[84] Ming Cheng et al. "Time: A training-in-memory architecture for memristor-based deep neural networks". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. 2017, pp. 1–6.

[85] Shuangchen Li et al. "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories". In: *Proceedings of the 53rd Annual Design Automation Conference*. 2016, pp. 1–6.

[86] Zhezhi He et al. "Leveraging dual-mode magnetic crossbar for ultra-low energy in-memory data encryption". In: *Proceedings of the on Great Lakes Symposium on VLSI 2017*. 2017, pp. 83–88.

[87] Shihui Yin et al. "XNOR-SRAM: In-memory computing SRAM macro for binary/ternary deep neural networks". In: *IEEE Journal of Solid-State Circuits* 55.6 (2020), pp. 1733–1743.

[88] Zhewei Jiang et al. "C3SRAM: An in-memory-computing SRAM macro based on robust capacitive coupling computing mechanism". In: *IEEE Journal of Solid-State Circuits* 55.7 (2020), pp. 1888–1897.

[89] Xiaoyu Sun et al. "XNOR-RRAM: A scalable and parallel resistive synaptic architecture for binary neural networks". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 1423–1428.

[90] Shaahin Angizi et al. "Cmp-pim: an energy-efficient comparator-based processing-in-memory neural network accelerator". In: *Proceedings of the 55th Annual Design Automation Conference*. 2018, pp. 1–6.

[91] Shaahin Angizi et al. “IMCE: Energy-efficient bit-wise in-memory convolution engine for deep neural network”. In: *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2018, pp. 111–116.

[92] Shaizeen Aga et al. “Compute caches”. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2017, pp. 481–492.

[93] Charles Eckert et al. “Neural cache: Bit-serial in-cache acceleration of deep neural networks”. In: *2018 ACM/IEEE 45Th annual international symposium on computer architecture (ISCA)*. IEEE. 2018, pp. 383–396.

[94] Vivek Seshadri et al. “Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology”. In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 2017, pp. 273–287.

[95] Shuangchen Li et al. “Drisa: A dram-based reconfigurable in-situ accelerator”. In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 2017, pp. 288–301.

[96] Benjamin C Lee et al. “Architecting phase change memory as a scalable dram alternative”. In: *Proceedings of the 36th annual international symposium on Computer architecture*. 2009, pp. 2–13.

[97] Mohsen Imani, Yeseong Kim, and Tajana Rosing. “Mpim: Multi-purpose in-memory processing using configurable resistive memory”. In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2017, pp. 757–763.

[98] Juan Gómez-Luna et al. “Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system”. In: *IEEE Access* 10 (2022), pp. 52565–52608.

[99] Sukhan Lee et al. “Hardware architecture and software stack for PIM based on commercial DRAM technology: Industrial product”. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2021, pp. 43–56.

[100] Yongkee Kwon et al. “System architecture and software stack for GDDR6-AiM”. In: *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE. 2022, pp. 1–25.

[101] Ian Kuon, Russell Tessier, Jonathan Rose, et al. “FPGA architecture: Survey and challenges”. In: *Foundations and Trends® in Electronic Design Automation* 2.2 (2008), pp. 135–253.

[102] Yupeng Chen. "Design and analysis of bioinformatics algorithms on an FPGA platform". PhD thesis. 2014.

[103] *FPGAs: structure, elements and configuration.* URL: https://community.element14.com/technologies/fpga-group/b/blog/posts/fpgas-structure-elements-and-configuration.

[104] Ajay Kumar Singh. "Power Efficient Data-Aware SRAM Cell for SRAM-Based FPGA Architecture". In: *Field-Programmable Gate Array*. IntechOpen, 2017.

[105] Jeremy Soh. "A scalable, portable, FPGA-based implementation of the Unscented Kalman Filter". PhD thesis. The University of Sydney, 2017.

[106] ARM Holdings. *AMBA AXI and ACE Protocol Specification*. Tech. rep. Tech. rep. 2011. url: https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi _protocol_spec.pdf, 2013.

[107] AXI Xilinx. "Reference Guide, UG761 (v13. 1)". In: *URL https://www.xilinx .com/support/documentation/ip_documentation/ug761_axi_reference _guide.pdf* (2011).

[108] *RISC-V Specifications*. URL: https://github.com/riscv.

[109] *History of RISC-V*. URL: https://riscv.org/about/history/.

[110] Andrew Waterman et al. "The risc-v instruction set manual, volume i: Base user-level isa". In: *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* 116 (2011).

[111] Matthew Connatser. *What is RISC-V*. May, 2022. URL: https://www.digitaltrends.com/computing/what-is-risc-v/.

[112] Trong-Thuc HOANG and Cong-Kha PHAM. *RISC-V-based System-on-Chip (SoC) Fully Equipped with Cryptographic Accelerators for Transport Layer Security (TLS) 1.3*.

[113] Farzad Farshchi, Qijing Huang, and Heechul Yun. "Integrating NVIDIA deep learning accelerator (NVDLA) with RISC-V SoC on FireSim". In: *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. IEEE. 2019, pp. 21–25.

[114] L Calicchia et al. "Digital signal processing accelerator for RISC-V". In: *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE. 2019, pp. 703–706.

[115] *Extending RISC-V ISA With a Custom Instruction Set Extension*. URL: https://www.design-reuse.com/articles/46237/extending-risc-v-isa-with-a-custom-instruction-set-extension.html.

[116] Hao Cheng et al. “RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2023), pp. 193–237.

[117] Seyed Kian Mousavikia et al. “Instruction Set Extension of a RiscV Based SoC for Driver Drowsiness Detection”. In: *IEEE Access* 10 (2022), pp. 58151–58162.

[118] Marius Erbert, Steffen Rechner, and Matthias Müller-Hannemann. “Gerbil: a fast and memory-efficient k-mer counter with GPU-support”. In: *Algorithms for Molecular Biology* 12.1 (2017), p. 9.

[119] Huiren Li, Anand Ramachandran, and Deming Chen. “GPU Acceleration of Advanced k-mer Counting for Computational Genomics”. In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2018, pp. 1–4.

[120] Nicola Cadenelli, Jordà Polo, and David Carrera. “Accelerating K-mer frequency counting with GPU and non-volatile memory”. In: *2017 IEEE 19th International Conference on High Performance Computing and Communications*. IEEE. 2017, pp. 434–441.

[121] Wenqin Huangfu et al. “Nest: Dimm based near-data-processing accelerator for k-mer counting”. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. 2020, pp. 1–9.

[122] Biresh Kumar Joardar et al. “NoC-enabled software/hardware co-design framework for accelerating k-mer counting”. In: *Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip*. 2019, pp. 1–8.

[123] Nicola Cadenelli et al. “Considerations in using OpenCL on GPUs and FPGAs for throughput-oriented genomics workloads”. In: *Future Generation Computer Systems* 94 (2019), pp. 148–159.

[124] Nathaniel Mcvicar, Chih-Ching Lin, and Scott Hauck. “K-mer counting using Bloom filters with an FPGA-attached HMC”. In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2017, pp. 203–210.

[125] Minh Pham, Yicheng Tu, and Xiaoyi Lv. "Accelerating BWA-MEM Read Mapping on GPUs". In: *Proceedings of the 37th International Conference on Supercomputing*. 2023, pp. 155–166.

[126] Harisankar Sadasivan et al. "Accelerating Minimap2 for accurate long read alignment on GPUs". In: *bioRxiv* (2022).

[127] Sita Rani and OP Gupta. "CLUS_GPU-BLASTP: accelerated protein sequence alignment using GPU-enabled cluster". In: *The Journal of Supercomputing* 73.10 (2017), pp. 4580–4595.

[128] André Müller et al. "AnySeq/GPU: A Novel Approach for Faster Sequence Alignment on GPUs". In: *arXiv preprint arXiv:2205.07610* (2022).

[129] André Müller et al. "AnySeq: a high performance sequence alignment library based on partial evaluation". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2020, pp. 1030–1040.

[130] Quim Aguado-Puig et al. "WFA-GPU: Gap-affine pairwise alignment using GPUs". In: *bioRxiv* (2022).

[131] Giulia Gerometta, Alberto Zeni, and Marco D Santambrogio. "TSUNAMI: A GPU implementation of the WFA algorithm". In: *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2023, pp. 150–161.

[132] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. "CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions". In: *BMC bioinformatics* 14.1 (2013), pp. 1–10.

[133] Bertil Schmidt et al. "CUDASW++ 4.0: Ultra-fast GPU-based Smith-Waterman Protein Sequence Database Search". In: *bioRxiv* (2023), pp. 2023–10.

[134] Sven Warris et al. "pyPaSWAS: Python-based multi-core CPU and GPU sequence alignment". In: *PLoS One* 13.1 (2018), e0190279.

[135] Liang-Tsung Huang et al. "Improving the mapping of Smith-Waterman sequence database searches onto CUDA-enabled GPUs". In: *BioMed research international* 2015 (2015).

[136] Jacek Blazewicz et al. "Protein alignment algorithms with an efficient backtracking routine on multiple GPUs". In: *BMC bioinformatics* 12.1 (2011), pp. 1–17.

[137] Mohammed A Shehab et al. "A hybrid CPU-GPU implementation to accelerate multiple pairwise protein sequence alignment". In: *2017 8th International Conference on Information and Communication Systems (ICICS)*. IEEE. 2017, pp. 12–17.

[138] Farzaneh Zokaee, Hamid R Zarandi, and Lei Jiang. "Aligner: A process-in-memory architecture for short read alignment in reRAMs". In: *IEEE Computer Architecture Letters* 17.2 (2018), pp. 237–240.

[139] Wenqin Huangfu et al. "Radar: a 3D-reRAM based DNA alignment accelerator architecture". In: *Proceedings of the 55th Annual Design Automation Conference*. 2018, pp. 1–6.

[140] Zamshed I Chowdhury et al. "A DNA read alignment accelerator based on computational RAM". In: *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits* 6.1 (2020), pp. 80–88.

[141] Nika Mansouri Ghiasi et al. "GenStore: A High-Performance and Energy-Efficient In-Storage Computing System for Genome Sequence Analysis". In: *arXiv preprint arXiv:2202.10400* (2022).

[142] Ting Wu et al. "RePAIR: a ReRAM-based processing-in-memory accelerator for indel realignment". In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2022, pp. 400–405.

[143] Roman Kaplan, Leonid Yavits, and Ran Ginosasr. "BioSEAL: In-Memory Biological Sequence Alignment Accelerator for Large-Scale Genomic Data". In: *Proceedings of the 13th ACM International Systems and Storage Conference*. 2020, pp. 36–48.

[144] Roman Kaplan et al. "A resistive CAM processing-in-storage architecture for DNA sequence alignment". In: *IEEE Micro* 37.4 (2017), pp. 20–28.

[145] Saransh Gupta et al. "RAPID: A ReRAM processing in-memory architecture for DNA sequence alignment". In: *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE. 2019, pp. 1–6.

[146] Safaa Diab et al. "High-throughput pairwise alignment with the wavefront algorithm using processing-in-memory". In: *arXiv preprint arXiv:2204.02085* (2022).

[147] Safaa Diab et al. "A framework for high-throughput sequence alignment using real processing-in-memory systems". In: *Bioinformatics* 39.5 (2023), btad155.

[148] DS Nurdin, MN Isa, and SH Goh. "DNA sequence alignment: A review of hardware accelerators and a new core architecture". In: *International Conference on Electronic Design (ICED)*. IEEE. 2016, pp. 264–268.

[149] Laiq Hasan and Zaid Al-Ars. "An overview of hardware-based acceleration of biological sequence alignment". In: *Computational Biology and Applied Bioinformatics* (2011), pp. 187–202.

[150] Brian Hill et al. "Precision medicine and FPGA technology: Challenges and opportunities". In: *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE. 2017, pp. 655–658.

[151] Licheng Guo et al. "Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between FPGA and GPU". In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019, pp. 127–135.

[152] Yu-Ting Chen et al. "A novel high-throughput acceleration engine for read alignment". In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2015, pp. 199–202.

[153] James Arram et al. "Reconfigurable acceleration of short read mapping". In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2013, pp. 210–217.

[154] Ho-Cheung Ng et al. "Acceleration of Short Read Alignment with Runtime Reconfiguration". In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE. 2020, pp. 256–262.

[155] Daichi Fujiki et al. "SeedEx: A Genome Sequencing Accelerator for Optimal Alignments in Subminimal Space". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 937–950.

[156] Ernst Joachim Houtgast et al. "An FPGA-based systolic array to accelerate the BWA-MEM genomic mapping algorithm". In: *2015 international conference on embedded computer systems: Architectures, modeling, and simulation (samos)*. IEEE. 2015, pp. 221–227.

[157] Subho Sankar Banerjee et al. "Asap: Accelerated short-read alignment on programmable hardware". In: *IEEE Transactions on Computers* 68.3 (2018), pp. 331–346.

[158] Damla Senol Cali et al. "Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 951–966.

[159] Damla Senol Cali et al. "SeGraM: a universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping". In: *arXiv preprint arXiv:2205.05883* (2022).

[160] Carlos AC Jorge et al. "A CPU-FPGA heterogeneous approach for biological sequence comparison using high-level synthesis". In: *Concurrency and Computation: Practice and Experience* (2020), e6007.

[161] Scott Lloyd and Quinn O Snell. "Hardware accelerated sequence alignment with traceback". In: *International Journal of Reconfigurable Computing* (2009).

[162] Riadh Ben Abdelhamid and Yoshiki Yamaguchi. "A Block-Based Systolic Array on an HBM2 FPGA for DNA Sequence Alignment". In: *International Symposium on Applied Reconfigurable Computing*. 2020, pp. 298–313.

[163] Ernst Houtgast, Vlad-Mihai Sima, and Zaid Al-Ars. "High performance streaming Smith-Waterman implementation with implicit synchronization on intel FPGA using OpenCL". In: *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE)*. IEEE. 2017, pp. 492–496.

[164] Lorenzo Di Tucci et al. "Architectural optimizations for high performance and energy efficient Smith-Waterman implementation on FPGAs using OpenCL". In: *Design, Automation & Test in Europe Conference (DATE)*. 2017, pp. 716–721.

[165] Barry Strengholt and Matthijs Brobbel. *Acceleration of the Smith-Waterman algorithm for DNA sequence alignment using an FPGA platform*. 2013.

[166] Juan M Marmolejo-Tejada et al. "Hardware implementation of the Smith-Waterman algorithm using a systolic architecture". In: *2014 IEEE 5th Latin American Symposium on Circuits and Systems*. IEEE. 2014, pp. 1–4.

[167] Lars Wienbrandt. "Bioinformatics applications on the FPGA-based high-performance computer RIVYERA". In: *High-Performance Computing Using FPGAs*. Springer, 2013, pp. 81–103.

[168] EP Vermij. *Genetic sequence alignment on a supercomputing platform*. 2011.

[169] Chi Wai Yu et al. “A Smith-Waterman systolic cell”. In: *International Conference on Field Programmable Logic and Applications*. Springer. 2003, pp. 375–384.

[170] Kiran Puttegowda et al. “A run-time reconfigurable system for gene-sequence searching”. In: *16th International Conference on VLSI Design, 2003. Proceedings.* IEEE. 2003, pp. 561–566.

[171] Tom Van Court and Martin C Herbordt. “Families of FPGA-Based Algorithms for Approximate String Matching.” In: *ASAP*. 2004, pp. 354–364.

[172] Xia Fei et al. “FPGASW: Accelerating large-scale Smith-Waterman sequence alignment application with backtracking on FPGA linear systolic array”. In: *Interdisciplinary Sciences: Computational Life Sciences* 10.1 (2018), pp. 176–188.

[173] Yi-Lun Liao et al. “Adaptively Banded Smith-Waterman Algorithm for Long Reads and Its Hardware Accelerator”. In: *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2018, pp. 1–9.

[174] Yatish Turakhia, Gill Bejerano, and William J Dally. “Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly”. In: *ACM SIGPLAN Notices* 53.2 (2018), pp. 199–213.

[175] Yatish Turakhia et al. “Darwin: A hardware-acceleration framework for genomic sequence alignment”. In: *bioRxiv* (2017), p. 092171.

[176] Yatish Turakhia et al. “Darwin-WGA: A co-processor provides increased sensitivity in whole genome alignments with high speedup”. In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2019, pp. 359–372.

[177] Enzo Rucci et al. “SWIFOLD: Smith-Waterman implementation on FPGA with OpenCL for long DNA sequences”. In: *BMC systems biology* 12.5 (2018), p. 96.

[178] Cuong Pham-Quoc, Binh Kieu-Do, and Tran Ngoc Thinh. “A high-performance FPGA-based BWA-MEM DNA sequence alignment”. In: *Concurrency and Computation: Practice and Experience* 33.2 (2021), e5328.

[179] Konstantina Koliogeorgi et al. “Dataflow acceleration of Smith-Waterman with traceback for high throughput next generation sequencing”. In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2019, pp. 74–80.

[180] Enzo Rucci et al. "OSWALD: OpenCL Smith-Waterman on Altera's FPGA for Large Protein Databases". In: *The International Journal of High Performance Computing Applications* 32.3 (2018), pp. 337–350.

[181] Yoshiki Yamaguchi, Hung Kuen Tsoi, and Wayne Luk. "FPGA-based Smith-Waterman algorithm: Analysis and novel design". In: *International Symposium on Applied Reconfigurable Computing*. Springer. 2011, pp. 181–192.

[182] Xianyang Jiang et al. "A reconfigurable accelerator for Smith-Waterman algorithm". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 54.12 (2007), pp. 1077–1081.

[183] Isaac TS Li, Warren Shum, and Kevin Truong. "160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA)". In: *BMC bioinformatics* 8.1 (2007), p. 185.

[184] Jeff Allred et al. "Smith-Waterman implementation on a FSB-FPGA module using the Intel Accelerator Abstraction Layer". In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–4.

[185] Peiheng Zhang, Guangming Tan, and Guang R Gao. "Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform". In: *International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA)*. 2007, pp. 39–48.

[186] Khaled Benkrid, Ying Liu, and AbdSamad Benkrid. "A highly parameterized and efficient FPGA-based skeleton for pairwise biological sequence alignment". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17.4 (2009), pp. 561–570.

[187] Sean O Settle et al. "High-performance dynamic programming on FPGAs with OpenCL". In: *IEEE High Performance Extreme Computing Conference (HPEC)*. 2013, pp. 1–6.

[188] Peng Chen et al. "Accelerating the next generation long read mapping with the FPGA-based system". In: *IEEE/ACM transactions on computational biology and bioinformatics* 11.5 (2014), pp. 840–852.

[189] Daichi Fujiki et al. "Genax: A genome sequencing accelerator". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 69–82.

[190] Jing-Ping Wu et al. “A Memory-Efficient Accelerator for DNA Sequence Alignment with Two-Piece Affine Gap Tracebacks”. In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2021, pp. 1–4.

[191] TOP500. *The 500 most powerful commercially available computer systems*. URL: https://www.top500.org/lists/top500/2022/06.

[192] Jeffrey Stuecheli et al. “CAPI: A coherent accelerator processor interface”. In: *IBM Journal of Research and Development* 59.1 (2015), pp. 7–1.

[193] Jeffrey Stuecheli et al. “IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI”. In: *IBM Journal of Research and Development* 62.4/5 (2018), pp. 8–1.

[194] IBM. *CAPI2 and SNAP repository*. URL: https://github.com/open-power/snap.

[195] IBM. *How is data managed in the SNAP environment?* English. Version 1.0. IBM. Apr. 30, 2018. 21 pp. URL: https://github.com/open-power/snap/blob/master/doc/AN_CAPI_SNAP-How_is_data_managed.pdf.

[196] IBM. *OpenCAPI repository*. URL: https://github.com/OpenCAPI/oc-accel.

[197] Alpha Data. *ADM-PCIE-9V3 board*. URL: https://www.alpha-data.com/product/adm-pcie-9v3/.

[198] Alpha Data. *ADM-PCIE-9H7 board*. URL: https://www.alpha-data.com/product/adm-pcie-9h7/.

[199] AMD. *Alveo U280 FPGA board*. URL: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[200] Santiago Marco-Sola. *WFA open source C implementation*. URL: https://github.com/smarco/WFA.

[201] Hak-Min Kim et al. “Comparative analysis of 7 short-read sequencing platforms using the Korean Reference Genome: MGI and Illumina sequencing benchmark for whole-genome sequencing”. In: *GigaScience* 10.3 (Mar. 2021).

[202] U.S. Food and Drug Administration. *PacBio HIFI input set*. URL: https://precision.fda.gov/challenges/10.

[203] Chunlin Xiao. *PacBio CCS input set*. URL: https://github.com/genome-in-a-bottle/giab_data_indexes.

[204] Rick Wertenbroek and Yann Thoma. "k-mer counting with FPGAs and HMC in-memory operations". In: *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE. 2018, pp. 233–240.

[205] David Carrera Perez et al. *A computer-implemented and reference-free method for identifying variants in nucleic acid sequences*. US Patent App. 16/315,982. Jan. 2020.

[206] Todd Rosedahl et al. "Power/performance controlling techniques in OpenPOWER". In: *International Conference on High Performance Computing*. Springer. 2017, pp. 275–289.

[207] Abbas Haghi. *WFA FPGA implementation*. URL: https://gitlab.bsc.es/ahaghi/wfa%5C_fpga%5C_accelerator.

[208] *UltraScale Architecture Memory Resources User Guide*. English. Version Version 1.13. Xilinx. 139 pp. September 24, 2021.

[209] IBM. *CloudFPGA*. URL: https://www.zurich.ibm.com/cci/cloudFPGA/.

[210] Santiago Marco-Sola et al. "Optimal gap-affine alignment in O(s) space". In: *Bioinformatics* 39.2 (2023), btad074.