Performance characterization and acceleration of genome-mapping tools on HPC environments

Christos Konstantinos Matzoros

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya

A thesis submitted for the degree of

Master in Innovation and Research in Informatics (MIRI:HPC)

October, 2022

Advisor: Santiago Marco-Solá

Tutor: Miquel Moretó Planas

To my family

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor, Professor Santiago Marco-Solá, for his continuous support during the years of my master's degree. His guidance and excellent feedback were pivotal for the completion of this work. I learned a lot from him, and he inspired me to become a more competent engineer. His trust, continuous support, and inspiration, even when seeing me struggling at times, are some of the qualities I will forever be grateful for. I would also like to show my warmest appreciation to my tutor, Professor Miquel Moretó Planas, for his significant and constructive suggestions during the planning and development of this work. I will always be grateful to both of them for allowing me to work under their wing in a great working environment parallel with my studies. They allowed me to contribute to additional research projects and collaborate with great people in Barcelona Supercomputing Center (BSC). I would also wish to express my gratitude to my co-worker and friend Quim Aguado for contributing with his ideas and engaging in a part of my thesis.

Finally, I would like to thank my family for their everlasting love and encouragement throughout my studies. This endeavor would not be possible without their unconditional support. To my friends, who stood with me during my darkest times and offered me unforgettable moments, I consider them part of my family too.

Abstract

Nowadays, the efficient analysis and exploitation of genomic information is paramount to future advancements in the healthcare sector, such as better diagnosis techniques and the development of improved disease treatments. In the past decades, the exponential increase in the biological data production has fostered the development of more efficient genomic pipelines. For that, modern genome analysis requires better and more scalable algorithms, and improved high-performance implementations that can exploit current hardware accelerators. For most genome analysis pipelines, sequence mapping is one of the most computationally intensive and time-consuming processing stages. The ultimate goal of this work is to propose techniques to accelerate read mapping, leveraging novel algorithms and hardware vector extensions.

In this thesis, we present a thorough performance characterization of the most widely-used genome-mapping tools and propose acceleration techniques that can effectively improve the performance of these tools. To that end, first, we identify the most time-consuming kernels, their performance bottlenecks, and the underlying causes of inefficiency. Afterwards, we design and implement an accelerated version of one of the most time-consuming steps: pairwise sequence alignment. For that, we propose to replace the classical dynamic-programming algorithm, used within these tools, with the recently proposed wavefront alignment algorithm (WFA). Moreover, we design and implement the first fully-vectorized version of the WFA, leveraging Intel's AVX2 and AVX-512 instructions, to further accelerate sequence-to-sequence alignment. As a result, we demonstrate that our vectorized WFA implementation outperforms the original scalar WFA implementation between 1.1×-2.4×. In turn, this renders speedups from 2.4× up to 826.7× compared to the most widely-used alignment algorithm, KSW2 (used within Minimap2 and Bwa-Mem2). We conclude that these tools can be significantly accelerated by selecting better algorithms (like the WFA) and leveraging fine-tuned implementations that can exploit hardware resources available in current high performance computing (HPC) processors.

Contents

Acknowledgments **ii**

Abstract **iii**

1 Introduction **1**

1.1 Context 1

1.2 Motivation 3

1.3 Goals and Contributions 3

1.4 Outline 4

2 Background **5**

2.1 Basic Biology 5

2.2 Sequencing 7

2.3 Genome Analysis Pipelines 9

2.4 Mapping Stages 12

2.4.1 Indexing 12

2.4.2 Seeding and Filtering 13

2.4.3 Seed Chaining and Extension 14

2.4.4 Pairwise Alignment 15

2.5 Genome Mappers 16

2.5.1 Minimap2 16

2.5.2 Bwa-Mem2 18

2.5.3 Bowtie2 19

3 Experimental Evaluation of Genome Mappers **20**
3.1 Experimental Setup 20
3.1.1 Machines 20
3.1.2 Tools 22
3.1.3 Datasets 23
3.2 Scalability Analysis 24
3.3 Bottleneck Analysis 30
3.4 Microarchitecture Exploration 32
3.5 Summary 35

4 Accelerating Sequence Alignment **37**
4.1 Classical Sequence Alignment Algorithms 37
4.2 Wavefront Alignment Algorithm (WFA) 39
4.2.1 Wavefront Furthest-Reaching Diagonals 39
4.2.2 Alignment Algorithm using Wavefronts 41
4.3 Exploiting SIMD Paralellism in WFA 43
4.4 Vectorization Using AVX-512 Intrinsics 48
4.4.1 Initial Approach 49
4.4.2 Implementation 50
4.4.3 Optimizations 54
4.4.3.1 Hybrid Implementation 54
4.4.3.2 Loop Unrolling of Remaining Diagonals 56
4.4.3.3 Other Code Improvements 57
4.4.4 Performance Evaluation 58
4.5 Vectorization using AVX2 Intrinsics 61
4.5.1 Implementation 61
4.5.2 Unrolling a Second Iteration of Vectorized Code 63
4.5.3 Performance Evaluation 66
4.6 Comparison of Vectorized WFA Implementations 67
4.7 Comparison of Sequence Alignment Implementations 68

5 Conclusions **71**
5.0.1 Publications 72
5.1 Future work 72

5.2 Financial and Technical Support 73

6 Appendices **74**

List of Figures

2.1 DNA double helix and nucleotides [49] 6
2.2 Major stages of genome analysis pipelines 11
2.3 Minimap2 main components 17

3.1 Marenostrum 4 node [11] 21
3.2 Scalability plot of Bowtie2 using Illumina real dataset 25
3.3 Scalability plot of Bowtie2 using Mason2 simulated dataset 26
3.4 Scalability plot of Bwa-Mem2 using Illumina real dataset 27
3.5 Scalability plot of Bwa-Mem2 using Mason2 simulated dataset . . 28
3.6 Scalability plot of Minimap2 using real Oxford Nanopore dataset . 29
3.7 Scalability plot of Minimap2 using real PacBio dataset 29
3.8 Scalability plot of Minimap2 using simulated Pbsim2 dataset . . . 30
3.9 Bottleneck analysis of genomic tools executions 31
3.10 Chart of VTune's high-level metrics in Microarchitecture Exploration Analysis 33
3.11 Microarchitecture exploration analysis chart of the executions . . 34

4.1 Dependencies between wavefronts as to compute one element of the next wavefront 41
4.2 Evolution of register space 48
4.3 AVX-512 Data Types 49
4.4 Abstract of AVX-512 vectorization strategy 50
4.5 AVX2 Data Types 61
4.6 Execution trace of vectorized part with AVX2 64
4.7 Execution trace of vectorized part with AVX2 using loop unrolling 66

4.8 Theoretical percentage decrease based on the initial bottlenecks characterization and potential overall speedup of genome-mapping tools using the vectorized WFA algorithm 70

List of Tables

2.1 Principal characteristics of major sequencing methods [6] 9

3.1 Real Datasets 24
3.2 Simulated Datasets 24

4.1 Performance metrics for the different stages of optimizations (Instructions and Clocktics are expressed in Billions) 60
4.2 Performance evaluation of the first version (optimized) with AVX2 (Instructions and Clocktics are expressed in Billions, CPI) 63
4.3 Performance evaluation of the loop-unrolling version of AVX2 (Instructions and Clocktics are expressed in Billions) 67
4.4 Comparison of AVX implementations (Instructions and Clocktics are expressed in Billions) 68
4.5 Execution Time of KSW and WFA implementations 69
4.6 Speedups of WFA versions with respect to KSW2 69

Chapter 1

Introduction

1.1 Context

Genomic sequencing [14] is the method of deciphering the genetic material found in individual organisms (humans, plants, animals) or viruses. Sequencing genomic molecules enable research and development in bio-medicine and other life sciences through its rapidly growing presence in medicine, outbreak tracing, and understanding of pathogens [6]. The successful sequencing of the human genome [40] is propelling these advancements further.

In bioinformatics, sequence analysis [61] is the term that describes the computational analysis of a DNA, RNA, or peptide sequence in order to understand its features, biological function, structure, or evolution. The outcomes of investigating the sequences information can provide us with valuable insights. It unlocks the mysteries of how the genes, and eventually, how the cells of all known organisms behave under various conditions. Using an efficient way to process that information through the genomic pipelines, could lead to advancements in the health sector, e.g., by improving the therapies or the prescribed medications. We define a bioinformatics pipeline [63] as the set of bioinformatics algorithms we execute in a predefined sequence to process sequence data. Clinical laboratories rely on resource-intensive data processing pipelines to analyze sequencing data [63].

The developments we previously mentioned, are also driven by the introduction of High-Throughput Sequencing (HTS) analysis technologies that have dramatically reduced the cost of DNA sequencing [6]. In the past decade, they have become crucial in studies of genomics [16]. Sequencing information was traditionally being elucidated using a low throughput technique called Sanger sequencing. In 2008, the National Human Genome Research Institute (NGHRI) created a DNA sequencing technology initiative [62] aimed at achieving a 1000 dollars human genome for the following decade [67]. That started the high-throughput sequencing era, which led to the successful sequencing of a large part of the human genome [26] and the standardization of a reference genome that is still in use. HTS technologies can sequence multiple DNA molecules in parallel, sequencing hundreds of millions of DNA molecules at a time [16]. A general feature of all HTS technologies is the generation of a substantial amount of data that need to be processed efficiently. For example, the Illumina [32] platforms generate up to 100 gigabases of sequence data per lane (HiSeq 4000) or up to 15Gb (MiSeq) [20], using a parallel sequencing approach.

An essential step in most genomic analysis pipelines [63] is sequence alignment, which refers to the alignment (a.k.a. mapping) of the generated reads to a reference sequence, such as the human genome. Sequence alignment is a process of finding for each base pair in each sequencing read, the precise or approximate location in the reference genome [18]. The alignment step is the bottleneck in the genomic analysis pipelines [4]. This becomes more evident as the amount of data created from HTS technologies increases at a greater rate and needs to be processed as fast as possible.

We now need more than ever to make genome analysis more efficient. We need to read and analyze genomes accurately and efficiently enough to scale the analysis to population level [6]. There currently exist major computational bottlenecks that we need to tackle. This thesis attempts to identify some of these bottlenecks as well as to contribute to the acceleration of the sequence analysis process.

1.2 Motivation

Nowadays, the main challenge is to reconstruct the complete genome of an individual using the relatively short sequences that current HTS machines produce [74]. We need to efficiently find the actual location of each read in a potentially large reference while distinguishing between technical sequencing errors and true genetic variation within the sample. The outstanding amount of data produced by HTS technologies demand acceleration of the alignment process.

The available general processing resources are insufficient as they cannot process the sequencing information in a viable way. It is necessary to take advantage of high-performance computing machines [66]. For that, we need to ensure that these aligners are properly exploiting the available resources in high-performance computing machines.

1.3 Goals and Contributions

This work aims to characterize the performance of the most widely-used mapping tools in a high-end machine (Marenostrum 4) and propose techniques to accelerate alignment algorithms and mapping tools. We performed a detailed analysis of Minimap2 [43], Bwa-Mem2 [76], and Bowtie2 [41], three of the most widely used genome mappers. This characterization provided us useful insights about the scalability, the bottlenecks as well as the hardware usage issues of these genome mappers. The evaluation indicated the need for acceleration of the alignment process of the genome mappers. For that, we propose employing a more efficient alignment algorithm (the Wavefront Alignment WFA algorithm [46]), for which we design and implement a SIMD-accelerated version using Intel's AVX2 [35], and AVX-512 [34] intrinsics. As a result, we demonstrate that our vectorized WFA implementation outperforms the original non-vectorized WFA implementation between 1.1-2.4x times. That allows speedups for the vectorized WFA from 2.4 up to 826.7 with respect to the alignment algorithm of genome-mapping tools (KSW2). Finally, the tools have potential speedups from 1.3 to 2.9 if we substitute the KSW2 algorithm with the vectorized WFA.

1.4 Outline

The rest of this thesis is organized as follows:

Chapter 2 first provides a background on basic biology concepts. We describe the main stages of genome analysis pipelines, from the sequencing of the DNA up to the alignment process. We provide information about the main parts of the genome mapping tools. We explore the various mapping tools that are widely used and their distinctive characteristics. We finally describe the wavefront algorithm, a state-of-the-art sequence alignment algorithm.

Chapter 3 introduces the basic setup on which we will conduct our experimentation. It also provides scalability, bottleneck, and microarchitecture analysis of three widely used genome mappers.

Chapter 4 describes how we were driven to select more efficient algorithms (WFA) over the traditional dynamic programming method used in the genome-mapping tools (KSW2). It describes how we achieved to accelerate the wavefront algorithm using vectorization. We give a detailed analysis of two implementations, one using AVX2 and the other using AVX-512 intrinsics, as well as the optimizations that formed the final implementations. We conclude with a comparison of the implementations.

Chapter 5 concludes with a summary of our main findings and presents directions for future work.

Chapter 2

Background

This chapter provides a background on basic biology concepts and the primary stages of genome analysis pipelines. We also provide information regarding the major stages of the genome-mapping tools and their distinctive characteristics.

2.1 Basic Biology

All living beings, from viruses to humans, seem very diverse at first glance. In reality, all life forms are driven by the same molecular processes and share many similarities in their basis [73]. They are all made out of cells except viruses which also imitate the cell's functions [75]. Cells are considered the basic building blocks of all living things and have many parts, each with a different function [50]. They are made by a group of biological molecules responsible for a cell's structural and functional activities.

The primary biological molecules are carbohydrates, proteins, lipids, and nucleic acids [73]. Proteins and nucleic acids are vital for the survival of organisms. Nucleotides are the building blocks of nucleic acids. Nucleic acids (found in DNA and RNA) have the remarkable property of storing an organism's genetic code. The genetic code is the sequence of nucleotides that determines the amino acid sequence of proteins, which are critical to life [9].

DNA [54] is made up of two strands of nucleotides. Four types of nucleotides are attached to form a structure called a double helix. The DNA has two strands. These are anti-parallel and are linked by hydrogen bonding based on complementary base pairing (Figure 2.1). Adenine (A) pairs with thymine (T), while guanine (G) pairs with cytosine (C).

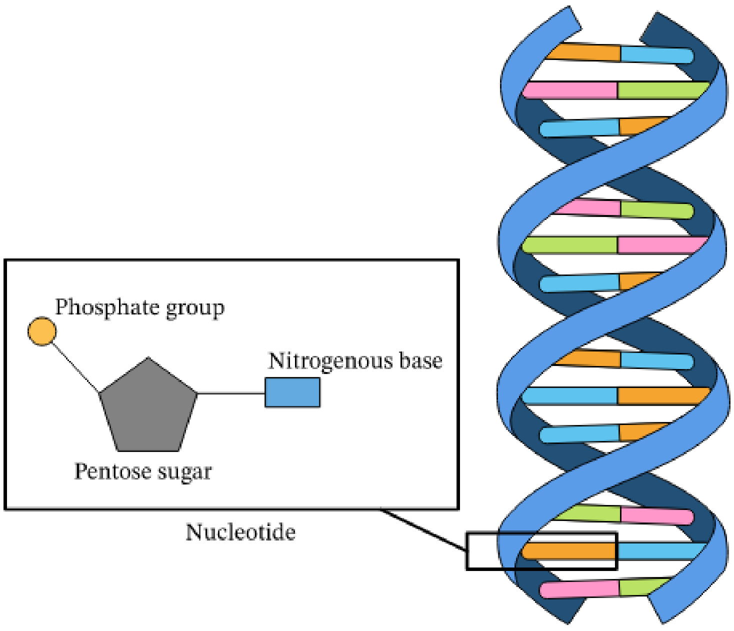

Figure 2.1: DNA double helix and nucleotides [49]

Genes [55] are found on chromosomes and they are made of DNA. Different genes determine the various traits of an organism. A gene has several parts. In most genes, the protein-making instructions are broken up into sections called exons. These are interspersed with introns, sections of not useful DNA. A chromosome [53] is a chunk of a genome that contains some of an organism's genes. Chromosomes help a cell to keep a large amount of genetic information in a structured way. An organism's whole set of nuclear DNA is referred to as the genome. It contains genes, which are packaged in chromosomes and affect specific characteristics of the organism.

Investigating the structure and behavior of the DNA, such as the ordering of its nucleotides, can provide us with valuable insights. It unlocks the mysteries of how the genes, and eventually, how the cells of all known organisms behave under various conditions. Using an efficient way to acquire and process that information could lead to advancements in the health sector, e.g., by improving the therapies or the prescribed medications.

2.2 Sequencing

We call genomic sequencing [14] the method that scientists use to decipher the genetic material found in individual organisms (humans, animals and others) or viruses. We can use this method to find changes in areas of the genome, which can help scientists understand how specific diseases form. Scientists can use genomic sequencing results to diagnose and treat various diseases. The first draft of the human genome was finished in 2001 by the Human Genome Project [33], an international scientific research project to define the base pairs that make up human DNA. Whole-genome sequencing technology can analyze entire genomes, revealing the information in them and exhibiting the complexity and heterogeneity of the genome.

Over the last decade, there has been a significant shift away from the Sanger sequencing technology for genome analysis [25]. The Sanger Method [65] also known as the "chain termination" method, is a method for determining the nucleotide sequence of DNA. It had dominated the industry for almost two decades [29]. The limitations of Sanger sequencing showed a need for new and improved technologies for sequencing large numbers of human genomes. The major limitation of the Sanger method is the sequencing volume, as it only sequences a single DNA fragment at a time. Also it can only sequence short pieces of DNA [23].

Newer methods are referred to as High-Throughput Sequencing (HTS) methods. HTS has several advantages over Sanger sequencing. HTS can produce high-throughput data from multiple samples in parallel per run [7]. Moreover, HTS produces more accurate sequencing data than Sanger DNA sequencing. It took about ten years to sequence the first draft human genome at a cost of several millions of dollars. In contrast, nowadays, it is possible to sequence the human genome within a single day at the cost of under one thousand US dollars or even less [22].

The diversity of HTS characteristics makes it likely that multiple platforms coexist in the marketplace. As a result, some platforms have advantages and disadvantages for particular applications over others. There are many different sequencing technologies, but we will focus only on the most used. These are the Illumina sequencing technology [32], the Pacific Biosciences sequencing technology [59], and the Oxford Nanopore Technologies [58]. Table 2.1 presents the principal characteristics of each sequencing method.

Illumina sequencing [32] is a second-generation sequencing method. It uses reversible dye terminators technology to detect the sequence of DNA molecules. It has high accuracy (about 99.9%) and creates sort sequences of 100 to 300 bp (base pairs) [6]. The advantages of this technology are that it is simple, scalable, and has a high yield. The main disadvantage is that it requires expensive equipment.

PacBio's [59] SMRT (Single Molecule, Real-Time) technology takes advantage of the natural process of DNA replication, which is a highly efficient and accurate process. SMRT technology occurs in real time. The average read length starts from 10KB bases up to 30KB bases. PacBio reads typically have a high error rate. However, their errors tend to be random, so if the same region is sequenced several times, the errors average out, resulting in a "consensus" sequence that gives an average accuracy rate of 99.9% [6]. Its advantages are that it is fast and contains informative data, and the main disadvantage is that it has high error rates.

Nanopore sequencing [58] is a third-generation sequencing technique. It uses a Nanopore to detect the sequence of DNA molecules. It has 90% to 98% accuracy and creates the most extended read lengths, from 100 bp to the current record of 2M bp [6]. The advantage of Nanopore sequencing is that it produces ultra-long reads. The main disadvantage of this technology is that it tends to be error-prone.

Principal Characteristics	**Illumina Technology**	**Pacific Biosciences**	**Oxford Nanopore Technologies**
Instrument cost (X constant)	3X	1.6X	X
Read length	100–300	10K-30K	100–2M
Read length in a single data file	Fixed	Modest Variability	Variable
Accuracy	99.9 %	99.9 %	90 %-98 %
Sequencing run time	44 h	30 h	72 h

Table 2.1: Principal characteristics of major sequencing methods [6]

2.3 Genome Analysis Pipelines

There are many challenges associated with selecting and implementing the right set of tools in basic research or clinical settings. Bioinformatics analyses are complex, multi-step processes comprised of multiple software applications [21]. Many academic efforts are spent improving the statistical accuracy of particular approaches leading to constant changes in the most appropriate pipeline that can be assembled at any given point in time. A genome analysis pipeline [60] comprises bioinformatics algorithms and tools executed in a predefined order to process genomic sequencing data. A pipeline progressively processes a large amount of sequence data and their associated metadata through a series of transformations using various software components, environments, and databases. A typical clinical implementation of a bioinformatics pipeline is automated and requires proper quality control to ensure the generated data are accurate, robust, and reproducible. Each step of a clinical sequencing pipeline emits information that can be used as metrics for bioinformatics pipeline quality control. We present the primary pipeline stages in Figure 2.2.

Most genome analysis pipelines start with sequencing data acquisition. After that, there is software to analyze the sensor data to predict the individual bases. Base-calling is the process that converts sensor data from the sequencing platform and distinguishes the sequence of nucleotides for each fragment of DNA in the sample prepared for analysis. The read sequences are usually stored in a FASTQ file format [51, 60]. Sequence alignment [43, 45, 70, 76] is the process of determining the region of the reference genome where the read originated. One, multiple, or no location can be assigned for each sequence read. This computationally intensive process outputs a genome location (i.e., chromosome, position, and strand) and a mapping score. The highest (or lowest depending on the algorithm) score indicated the best alignment for a specific read. The location can be used to calculate the proportion of mapped reads and the depth of the sequencing experiment (i.e., coverage). The sequence alignment output data is usually stored in a sequence alignment map (SAM) file format or a binary, compressed alignment file format (i.e., BAM). BAM files retain the same information as SAM files. The main difference is that they are in a binary file format. BAM files are smaller and more efficient than SAM files, saving time and reducing costs of computation and storage [6].

In many cases, the information of a single sequence is insufficient to derive the proper alignment of indels into the reference genome. This results in variations between the reference and the reads near the misaligned parts of the sequence [17]. The realignment step corrects these artifacts. This step refines the initial alignments by re-aligning all the sequences in a region together and identifying suspicious regions [60]. Base quality score recalibration (BQSR) uses this information as a covariate along with other known areas of variation. BQSR adjusts the base quality scores of sequencing reads using an empirical error model [39]. Realignment of mapped reads and recalibration of base quality scores before SNV calling proved to be crucial to accurate variant calling [60].

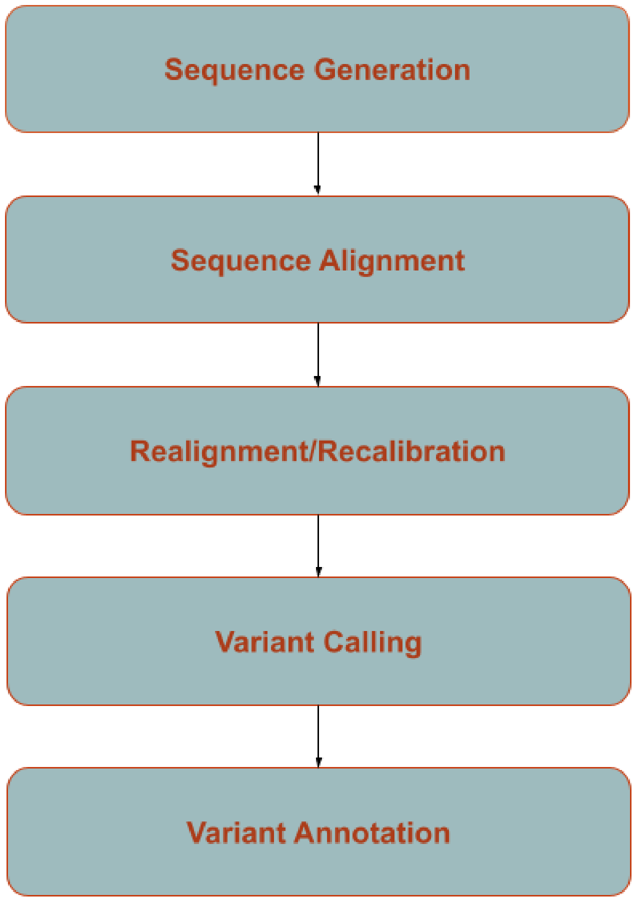

Figure 2.2: Major stages of genome analysis pipelines

We call variant Calling the differentiation of “true variants” from “noise” for a given sample [8]. Once we have a pre-processed, analysis-ready bam file using realignment and recalibration, we can begin the variant discovery process. Tools such as GATK perform variant calling. They can call SNPs and indels simultaneously via local de-novo assembly of haplotypes [27]. In other words, whenever the program finds a region displaying signs of variation, it discards the current mapping information and completely reassembles the reads in that region. This step is designed to maximize sensitivity to minimize false negatives, i.e., failing to identify actual variants. This brings us to the final stage, called Variant Annotation.

Variant Annotation [60] aims to identify the function and effect of all identified

SNPs. In this phase, the biological information is extracted. Variant identification generates a detailed catalog of variations in a genome, and among other applications, this process allows the diagnosis of known diseases or health conditions.

2.4 Mapping Stages

This thesis's work focuses on the most computationally intensive step of most genome analysis pipelines: the mapping stage. The process of sequence mapping involves different algorithms and data structures implemented within current genome mapping tools. In the following, we present the most widely used techniques employed within the most widely used mapping tools.

2.4.1 Indexing

The majority of genome mapping tools follow similar stages throughout their execution. Nevertheless, these stages have many differences between them in how they are implemented most of the time. Mappers can use different algorithms or data structures to implement the same concept. Most genome mapping tools use a computational procedure known as 'indexing' to accelerate their mapping algorithms. Indexing aims to quickly locate genomic subsequences in the reference genome [5].

In the indexing step, the mapper builds an extensive index database from a reference genome or a set of reads. After extracting the seeds from the reference genome sequence, the mapper stores them in a data structure along with their occurrence locations in the reference genome. Most prominent data-structured used for genome indexing include hash tables, suffix trees, suffix arrays, and FM-indexes.

Recent reports [5] show that hashing is the most popular indexing technique used by alignment tools. In hashing, large keys are transformed into short keys by applying hash functions. Then, the values are stored in a data structure called a hash table. The idea of hashing is to distribute entries of (key, value) form uniformly across an array. Each element is assigned a key. We can access the element in O(1) time using that key. The algorithm (hash function) computes an index that indicates where an entry can be found or inserted using the key.

In our case, the hash table is a data structure that stores the seeds and their corresponding locations in the reference genome. These regions are also known as k-mers. After receiving genomic reads from the sequencing process, the mapper extracts read-seeds from each read. Then it uses these read-seeds as a key to query the hash table index. The hash table returns a list of all occurrence locations of the read-seed in the reference genome. The simplicity and the short indexing time explain its popularity.

The second most popular type of indexing is the one that uses suffix-tree-based techniques. We define a suffix tree as a representation of a trie corresponding to the suffixes of a given string where all nodes with one child are merged with their parents [68]. In our case, a suffix tree is a tree-like data structure where each branch represents different genome suffixes. When two suffixes of the genome share the same prefix, the prefix needs to be stored only once. The leaf nodes of the suffix tree store all the locations of the specific unique suffix in the reference genome. Contrary to a hash table, a suffix tree supports searching for exact and inexact match seeds by traversing the tree branches from the root to a leaf node.

2.4.2 Seeding and Filtering

The second step in genome mapping is finding each read's potential positions in the reference sequence. This method is known as seeding. During seeding, the aligner first finds sub-strings of a DNA read that are exactly matching (or with a small error) in the genome at one or more than one place [77]. The genome mapper first extracts the read-seeds from each read sequence. Then by querying the index database, it determines the occurrences of each extracted read-seed in the reference sequence and their locations.

The determined seed locations are used to reduce the search space from the whole reference sequence to only the nearby region of each seed location. Filtering is the method of identifying and extracting only the regions of the reference genome that are likely to be similar to each of the read sequences. This method is applied before pairwise alignment, and the goal is to discard seed locations that pairwise alignment would consider a poor match [38]. Many mapping tools try this technique to avoid wasting computation on unnecessary alignments.

2.4.3 Seed Chaining and Extension

For each read-seed, the algorithm can also select only a small number of seeds that are apart from each other rather than selecting a large number of them. Most algorithms that support this approach attempt to restrict the number of differences at the gaps to avoid aligning a read to highly different regions in the reference genome [3]. The mapper performs this approach using seed extension followed by seed chaining. After locating an exact match between a read and the reference genome, the read alignment algorithm tries to extend the matching seed in both directions to the point where there are no more exact matches. These are called maximal exact matches (MEMs). Second, the algorithm explores the gaps between neighboring extended seeds in the reference genome and applies a pairwise alignment, a process to build a longer chain of these adjacent extended seeds.

The seed length and the seed type can affect the number of possible seed locations in the reference sequence [5]. For short seeds, the number of such locations is vast when considering the human genome's magnitude. The repetitive nature of genome sequences has, as a result, the high frequency of appearances of the same seeds on the reference. Many possible locations for short seeds impose a significant computational load. Thus, most read alignment algorithms apply heuristics to bypass searching all the seed locations in the reference genome [12, 13, 15]. Longer seed lengths reduce the number of possible seed locations in the reference genome and the number of chosen seeds from each read. These advantages come at the cost of a possible reduction in alignment sensitivity. To increase the seed length without reducing the alignment sensitivity, seeds can be generated as spaced seeds.

2.4.4 Pairwise Alignment

The last step is called pairwise alignment. This method identifies the similarity/differences and other relationships between pieces (local or global) of all extracted reads and each of the corresponding regions of the reference sequence [1, 2, 30, 47]. It can be used in the seed-extension part that we explained earlier. The algorithm should find the best possible alignment and determine the minimum number of differences between the two sequences and their locations. The algorithm also tries to determine the type of these differences. Each pairwise sequence alignment problem is solved by maximizing (or minimizing) an alignment score.

The alignment score uses a function called gap-penalty. Introducing gaps can allow an alignment algorithm to match additional terms compared to zero gap alignment. However, minimizing gaps in an alignment is vital to create a proper alignment. Too many gaps can make an alignment to become pointless. Gap penalties are used to adapt alignment scores based on the number and length of gaps.

The three main gap penalty types are the constant (edit distance), linear, and affine [6]. In the constant, a fixed negative score is given to every gap. The linear gap penalty considers the length 'L' of each insertion or deletion in the gap. If the penalty for each inserted or deleted element is 'P' and the length of the gap 'L', the total gap penalty would be the product of the two PL. The gap-affine model is the most widely used and combines the components in both the constant and linear gap penalty, taking the form $O + E \cdot L$. 'O' is known as the gap opening penalty, 'E' the gap extension penalty, and 'L' the length of the gap. Gap opening represents the cost required to open a gap of any length, and gap extension is the cost to extend the length of an existing gap by one. Affine gap scores are more general than linear and the edit distance costs, but are more costly to compute by a constant factor [6].

There are three main classifications of pairwise alignment [19]. First, we have global alignment. Here, the algorithm attempts to align two sequences entirely, from beginning to end, aligning every character in each sequence only once. The global alignment will contain all the characters from the query and the target sequences. This option is suitable if the two sequences are similar and have approximately the same length. The second is called local alignment. This method determines the best alignment between a subsequence of the query and a subsequence of the target sequences. This option is suitable for aligning more diverging sequences. Lastly, we have semi-global alignment, which is a variant of global alignment. This option allows for gaps to exist at the edges of one of the sequences.

Pairwise alignment algorithms can be categorized as dynamic programming (DP) and non-DP-based algorithms. The DP-based algorithms can be implemented as local alignment via Smith-Waterman [69] , global alignment via Needleman-Wunsch [52], Gotoh [28], or variations of them [24]. In that case, the entirety of one sequence is aligned to one of the ends of the other sequence. The non-DP algorithms include Hamming distance and the Rabin-Karp algorithm [37]. DP-based algorithms are favored over non-DP algorithms when one is interested in finding genetic substitutions, insertions, and deletions. The local alignment algorithm is generally preferred over global alignment when only a fraction of the read is expected to match some reference genome regions.

Lastly, the alignment process generates the alignment file, which contains alignment information such as the exact number of differences, the location of each difference, and their type.

2.5 Genome Mappers

The following sections describe the main characteristics of the investigated genome-mapping tools: Minimap2, Bwa-Mem2, and Bowtie2.

2.5.1 Minimap2

Minimap2 (mm2) [43] is a state-of-the-art sequence mapper and aligner. The mapper is used mostly for long-read sequencing platforms like Oxford Nanopore Technologies (ONT) and Pacific Biosciences (PacBio). Figure 2.3 depicts the main components of mm2. Like most full genome aligners, its algorithm is based on the classic seed-chain-align model. It has an offline pre-processing step to build an index from a reference sequence. The reference genome is indexed in the offline pre-processing step. It uses a hash table with the popular k-mer samples called minimizers. These minimizers are used as the key and minimizer locations on the reference as the values.

The next step is seeding. This step identifies short fixed-length exact matches (minimizers) between a read and a reference sequence. When mm2 processes a read, the extracted minimizers from the read are used to search the index for exact matches. These exact matches (anchors) are then sorted based on position in the reference.

Chaining takes the sorted anchors as the input and identifies collinear ordered subsets of anchors called chains such that no anchor is used in more than one chain. Minimap2 implements chaining using 1-dimensional dynamic programming where a complex problem is recursively broken down into simpler sub-problems. Chaining sub-selects a few regions (chains) on the target reference and reduces the work for the next step of base-level alignment.

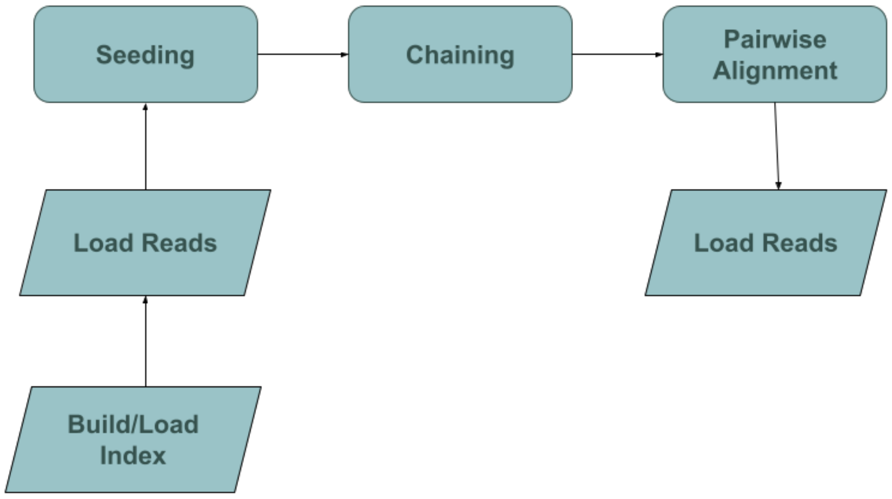

Figure 2.3: Minimap2 main components

Further, if base-level alignment is requested, the mapper applies 2-dimensional dynamic programming (Smith-Waterman [69], Suzuki-Kazahara[71]) to extend from the ends of chains to close the gaps between neighboring anchors in the chains. Minimap2 can map long noisy DNA/cDNA/mRNA reads, short, accurate genomics reads, find overlaps between long reads, and align to a whole reference genome or genome assembly.

2.5.2 Bwa-Mem2

Bwa-Mem2 [76] is another well-known tool for mapping short reads to larger reference sequences. The algorithm is composed of five main kernels. First is the SMEM kernel, which is responsible for the seeding. It searches for super maximal exact matches (SMEMs) between the read and the reference using FM-Index. Then it outputs SA intervals of the SMEMs. Maximal exact matches, called MEMs, are the exact matches between substrings of two different sequences that cannot be further extended in either of their directions. By extension, we call SMEM the MEM that is not contained in any other MEM on the read sequence.

The SAL kernel is next, where a suffix array lookup is carried out using the SA intervals to get the coordinates in the reference sequence. Following that, we have the CHAIN kernel, which chains collinear seeds close to each other. Chain filtering attempts to decrease the unsuccessful seed extension at the later step by filtering out short chains primarily contained in a long chain and much worse than the long chain.

BSW kernel is responsible for extending the seeds we filtered in the previous step. It uses a banded Smith-Waterman (BSW) alignment algorithm that computes only a diagonal band of the dynamic programming matrix. The last kernel, SAM-FORM, is responsible for formatting the output in the SAM format.

Bwa-mem2 is an improved version of Bwa-Mem [42], where they use architecture-aware optimizations to speed up the three main kernels (SMEM, SAL, and BSW). BSW and SMEM are instruction bound, and SMEM is also partially memory latency bound [76]. Bwa-Mem2 makes SAL kernel an insignificant contributor to the overall execution time.

2.5.3 Bowtie2

Bowtie 2 [41] is also a tool for aligning sequencing reads to long reference sequences. It is mainly used to align short reads of about 50 up to 300s of characters to relatively long genomes. Bowtie 2 extends the FM-Index method of Bowtie to allow gapped alignment by splitting the algorithm into two stages. First is the ungapped seed-finding stage, which benefits from the speed and memory efficiency of the FM-Index, and a gapped extension stage that utilizes dynamic programming and takes advantage of the efficiency of SIMD parallel processing available on processors.

Bowtie 2 is performed in four steps. For every read, we need to align; first, it extracts exact matches (seeds) from the read and its reverse complement. Then, the extracted substrings are aligned exactly to the reference using the FM-Index. Third, seed alignments are prioritized, and their positions in the reference genome are calculated from the index. Finally, the exact matches are extended into full alignments using an accelerated Smith-Waterman implementation [24].

Chapter 3

Experimental Evaluation of Genome Mappers

This chapter introduces the basic setup on which we conducted our characterization analysis. It also provides a scalability, bottleneck, and microarchitecture analysis using the Intel VTune profiler, for of three widely used genome mappers (Minimap2 [43], Bowtie2 [41], and Bwa-Mem2 [76]).

3.1 Experimental Setup

In the subsequent subsections, we present the characteristics of the machine we utilized for this thesis's experimentation. We document the tools that we used and their versions. These include, among others, profilers as well as simulators for creating the datasets. Finally, we provide the specifications of the datasets we use and how we acquired them. These datasets were used as input for the various analysis throughout this thesis.

3.1.1 Machines

For the experimentation part, we used the processing power of Marenostrum4 nodes that belongs to the Barcelona Supercomputing Center. MareNostrum4 is a supercomputer composed of Intel Xeon Platinum processors (Skylake). This system is composed of SD530 Compute Racks, an Intel Omni-Path high-performance network interconnect, and a SuSE Linux Enterprise Server operating system [10]. Its current Linpack Rmax Performance is 6.4708 Petaflops. This general-purpose block consists of 48 racks housing 3456 nodes. We will focus on the characteristics of one compute node as we use only a few for our experiments.

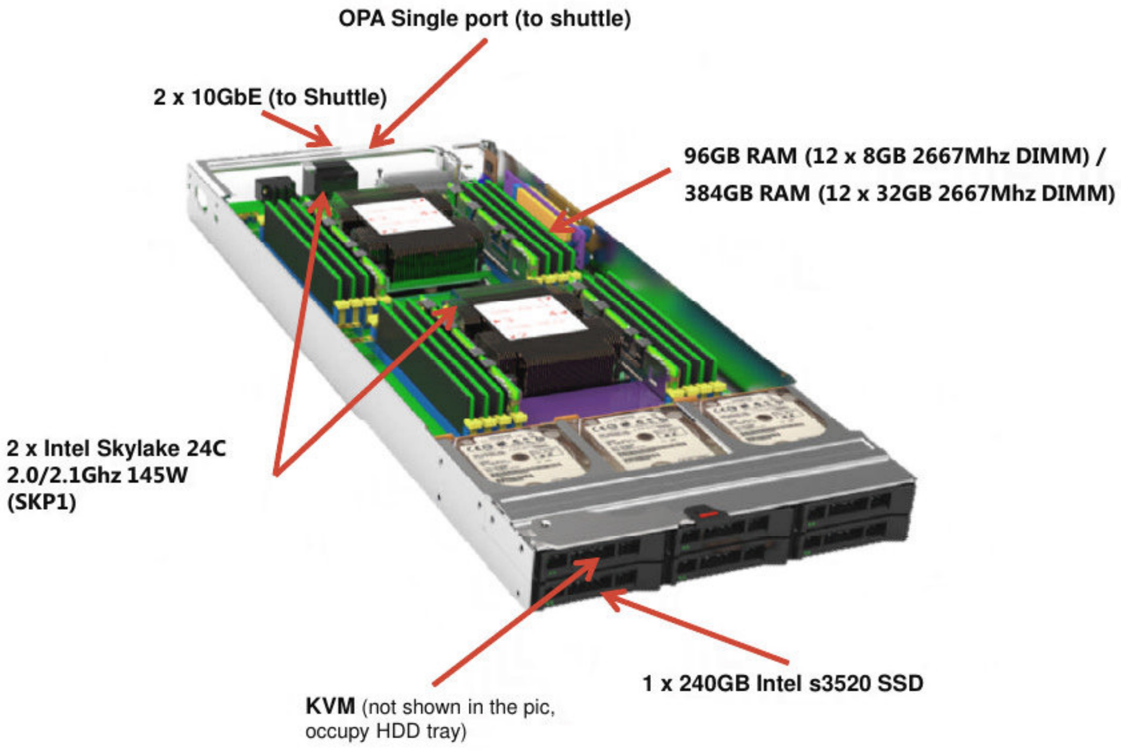
OPA Single port (to shuttle)
2 x 10GbE (to Shuttle)
96GB RAM (12 x 8GB 2667Mhz DIMM) /
384GB RAM (12 x 32GB 2667Mhz DIMM)
2 x Intel Skylake 24C
2.0/2.1Ghz 145W
(SKP1)
1 x 240GB Intel s3520 SSD
KVM (not shown in the pic, occupy HDD tray)

Figure 3.1: Marenostrum 4 node [11]

Figure 3.1 illustrates a mn4 compute node. Each one is equipped with 2 x Intel Xeon Platinum 8160 CPUs with 24 cores each at 2.10GHz for a total of 48 cores per node. The cache hierarchy consists of L1d and L1i caches of 32K, L2 cache of size 1024K, and L3 cache of 33792K. It contains a 96GB main memory with 1.88 GB available per core. The available interconnection networks are a 100 Gbit/s Intel Omni-Path HFI Silicon 100 Series PCI-E adapter or a 10 Gbit Ethernet. There is also available a 200 GB local SSD for temporary storage during jobs.

The processors support well-known vectorization instructions such as SSE, AVX up to AVX–512.

3.1.2 Tools

We use several tools for this thesis, such as genome mappers, profilers, and dataset simulators. We will now introduce each of them and the exact versions we used.

Starting with the genome mappers, we used Minimap2 [43], Bowtie2 [41], and Bwa-Mem2 [76]. Minimap2 is a versatile sequence alignment program that aligns DNA or mRNA sequences to a large reference database. The usual use case is mapping PacBio or Oxford Nanopore genomic reads to the human genome, but it is not limited to that. It offers many other functions, such as finding overlaps between long reads with error rates up to 15% or aligning Illumina single- or paired-end reads. For Minimap2, we use the Minimap2-2.24 release.

We continue with Bowtie2 mapper, a tool for aligning sequencing reads to long reference sequences. Contrary to Minimap2, which specializes in long reads, Bowtie2 is particularly good at aligning reads of about 50 up to 1000s of bases to relatively long genomes. We use version 2.4.5 for that tool. Next is Bwa-Mem2 genome tool, which, similarly to Bwa-Mem2, is also suitable for sequence reads up to 100 base pairs. The primary use case is mapping DNA sequences against a large reference genome. We use version 0.7.17 for our research.

In our experiments, we need sequence reads as input for the genome mappers. To feed them with sequences, we rely on real datasets that are open to the public. To better control the input data, we also used sequence simulators to produce simulated sequences, particularly pbsim2 [56, 57] and mason2 [31, 44]. For long reads, we use pbsim2. It is a long reads simulator that implements sampling-based or model-based simulations. It specializes in producing Continuous Long Reads (CLRs) of PacBio, and Nanopore reads. We also use Mason2, which simulates HTS reads given a genome, and optionally, a VCF file with variants for a given donor to use as the source. We used Version 2.0.0-beta1.

For the performance analysis of the genome-mapping tools, we mainly use the Intel® VTune™ Profiler [36]. It is a well-known profiler used for various types of analysis, such as hotspots analysis, microarchitecture exploration, memory access analysis, and others. We used version 2021.7.1, which is provided with Intel® oneAPI Base Toolkit.

3.1.3 Datasets

For our experimentation, we use both real datasets and simulated ones. The real datasets belong to three well-known technologies, which can also be grouped based on the length of their reads. For short reads, we use Illumina technology, while for long reads, we use PacBio and Oxford Nanopore technologies. Table 3.1 describes the characteristics of the real datasets used for our experimentation.

Furthermore, we used two read simulators to create simulated datasets. First, we have pbsim2, which simulates Continuous Long Reads (CLRs) of PacBio, and Nanopore reads. In our case, we simulated PacBio reads. Then we also used mason2, a read simulator software for Illumina, 454, and Sanger reads. For mason2, we decided to simulate Illumina reads. We have four more datasets, D1, D2, D3 and D4 which are created using the dataset generator offered from the WFA library. D1 and D2 both contain a number of 10K sequences with an average of 10K bases length each. The difference is that D1 has an error of 5% while D2 has an error of 15%. D3 and D4 both contain a number of 1M sequences with an average of 150 bases length each. The difference is that D3 has an error of 5% while D4 has an error of 15%. Table 3.2 describes the characteristics of the specific simulated datasets used for our experimentation. We use the Genome Reference Consortium Human Build 38 (GRCh38) as the reference dataset. It is composed of genomic sequences, primarily finished clones that were sequenced as part of the Human Genome Project.

Technology	**Number of Sequences**	**Length min/max/average**	**Number of Bases**	**Input Size**
Illumina	10M	148/148/148	148M	3.3G
PacBio	100K	24/55495/6779.23	678M	1.3G
Oxford Nanopore	100K	1/353743/9270	927M	1.8G

Table 3.1: Real Datasets

Simulated Technology (or Name)	**Number of Sequences**	**Length min/max/average**	**Sampled From**	**Number of Bases**	**Input Size**
pbsim2	100K	10K/10K/10K	PacBio	1000M	1.9G
mason2	10M	150/150/150	Illumina	150M	3.1G
D1	10K	9913/10K/10074	-	100M	0.2G
D2	10K	9867/10K/10128	-	100M	0.2G
D3	1M	143/150/159	-	150M	0.3G
D4	1M	133/150/169	-	150M	0.3G

Table 3.2: Simulated Datasets

3.2 Scalability Analysis

We start our analysis by exploring the scalability of the genomic-mapping tools. We use scalability to indicate the ability of hardware and software to handle larger amounts of work by enabling the usage more resources, in this case, processors. The scalability can be analyzed by estimating how its performance varies as a function of the input size growth and the number of processors. We attempt to measure how the genome mappers scale using the available datasets. We measure strong scalability which indicates how the solution time varies with the number of processors for a fixed total problem size. To extract this information we compare the ideal speedup to the actual speedup that is computed by the elapsed time of executions in the Marenostrum 4 node starting with one thread up to 48 threads.

We start with Bowtie2 using the two available datasets of short reads, the real Illumina data and the simulated dataset from mason2. As we see from the scalability plots in Figure 3.2 and Figure 3.3, we get similar speedups for both datasets. The application scales very good up the maximum available number of threads. For 48 threads, we get a speedup of 44.13 with the ideal being 48. This difference indicate that is a small room for improvements to improve the scalability.

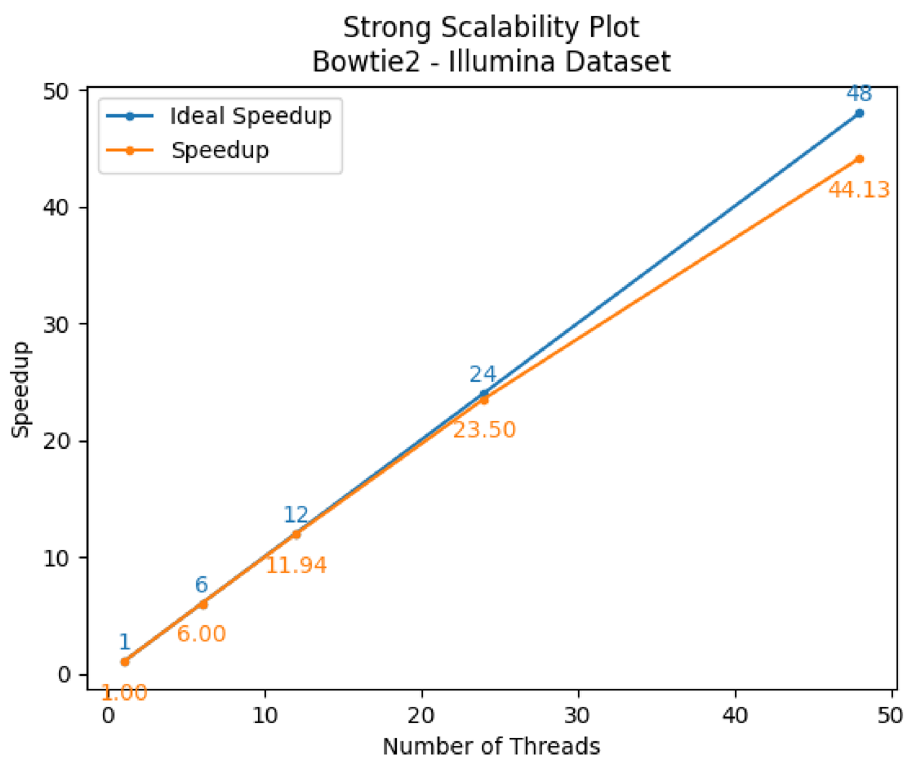

Figure 3.2: Scalability plot of Bowtie2 using Illumina real dataset

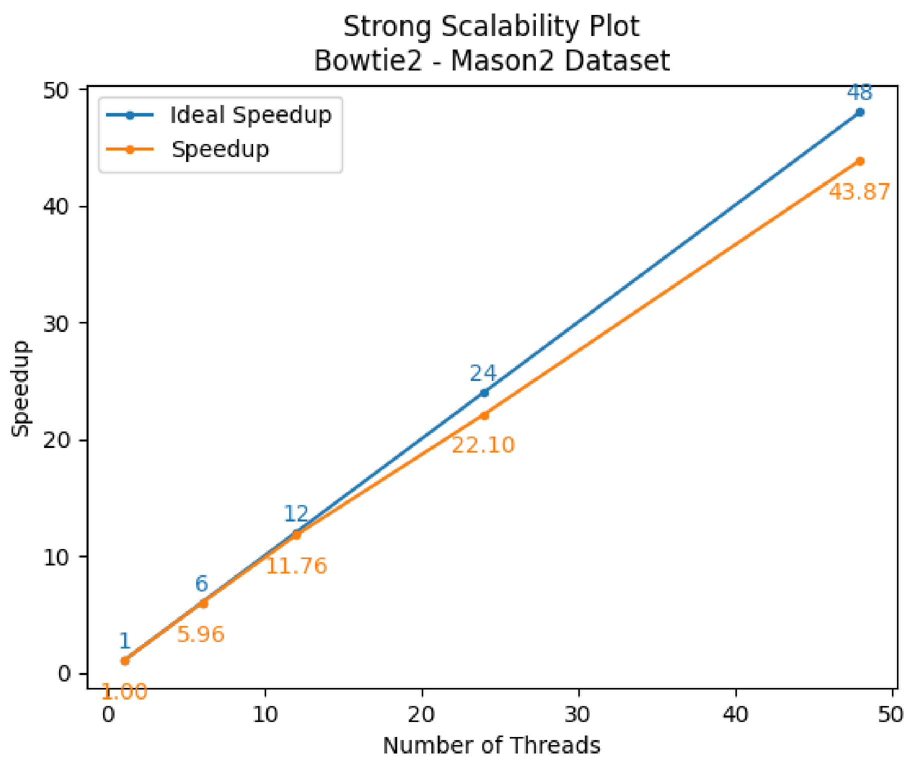

Figure 3.3: Scalability plot of Bowtie2 using Mason2 simulated dataset

Next, we use the same datasets to measure the scalability of Bwa-Mem2. From Figures 3.4 and 3.5, we can detect a scalability problem that is evident early on after using twelve of the available threads. The divergence between the ideal Speedup and the actual Speedup increases dramatically as we approach the maximum number of available threads.

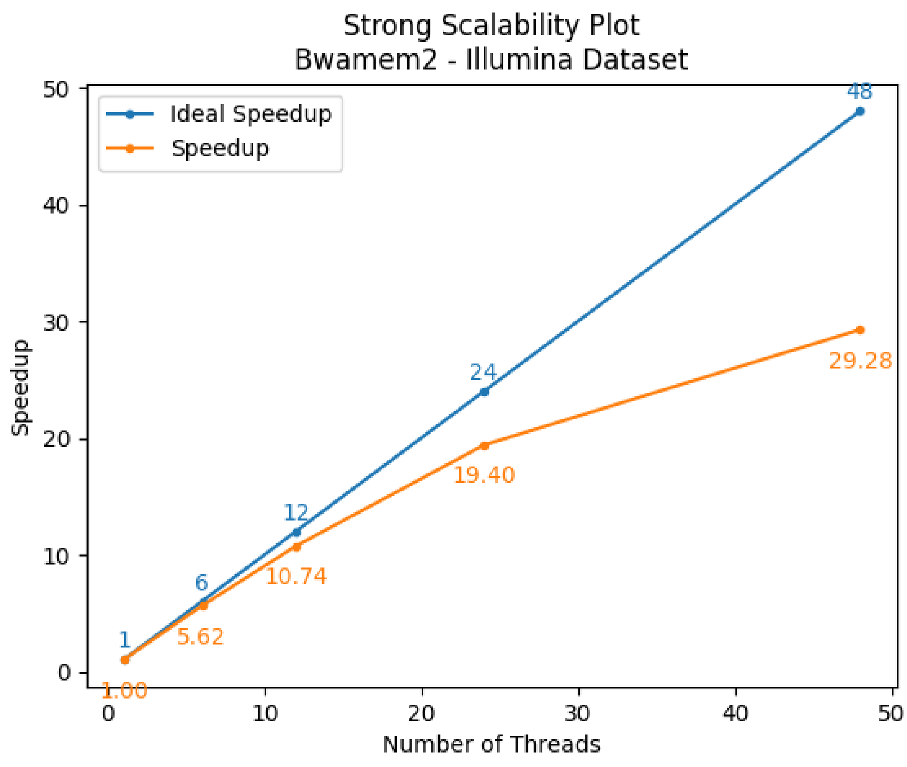

Figure 3.4: Scalability plot of Bwa-Mem2 using Illumina real dataset

The application is inefficient for the maximum number of threads as the actual Speedup is far from ideal, resulting in a parallel efficiency as small as 0.61 for the Illumina dataset and 0.66 for the simulated dataset. Parallel efficiency is a metric of the utilization of the resources of the improved system defined as $E = \frac{\text{Speedup}}{\text{Number of Processors}}$. The ideal value for efficiency is equal to one. The result indicates that parts of the application do not scale well.

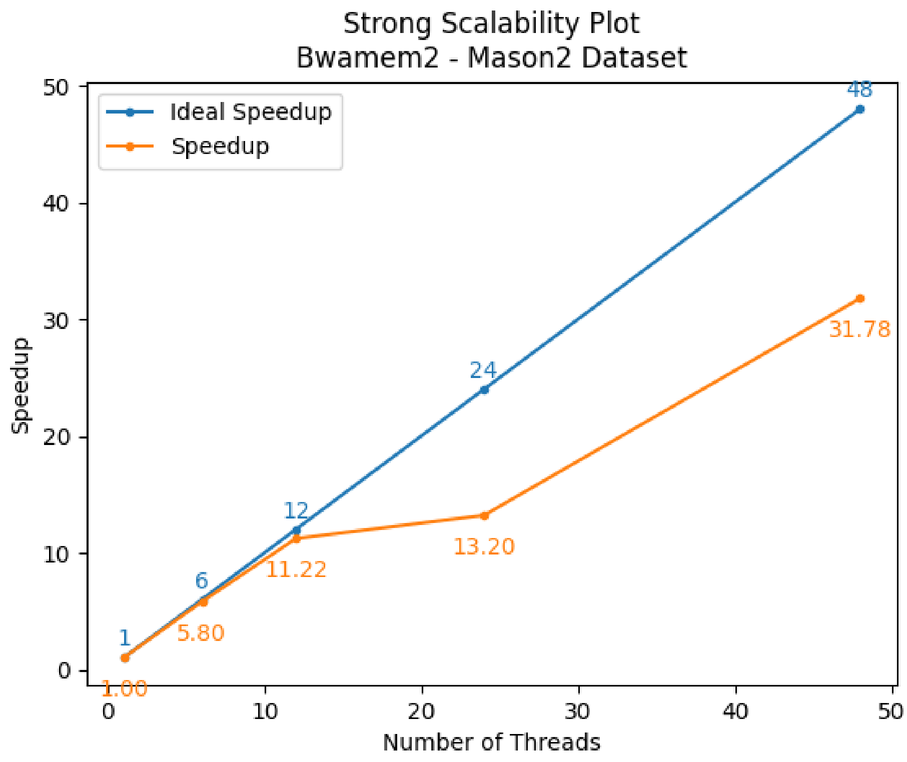

Figure 3.5: Scalability plot of Bwa-Mem2 using Mason2 simulated dataset

We continue with Minimap2, using the long read datasets, the real PacBio and Oxford Nanopore datasets, and the simulated dataset created using pbsim2. Figures 3.6, 3.7 and 3.8 show the results of the scalability analysis for the three different datasets in the prior mentioned order. All of the cases appear to have a scalability problem. This seems to happen for the first two datasets when using over twelve threads, while for the Pbsim2 dataset, this seems to appear earlier just by using over six threads. Also, the Pbsim2 dataset gives us a speedup for the maximum number of threads equal to 21.66, which is the worst compared to all the previous cases for this number of threads.

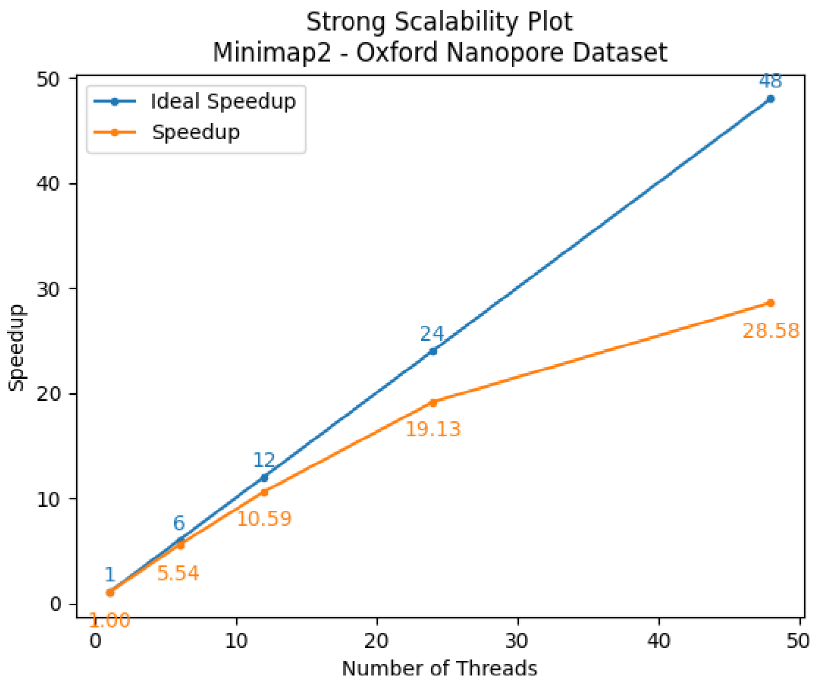

Figure 3.6: Scalability plot of Minimap2 using real Oxford Nanopore dataset

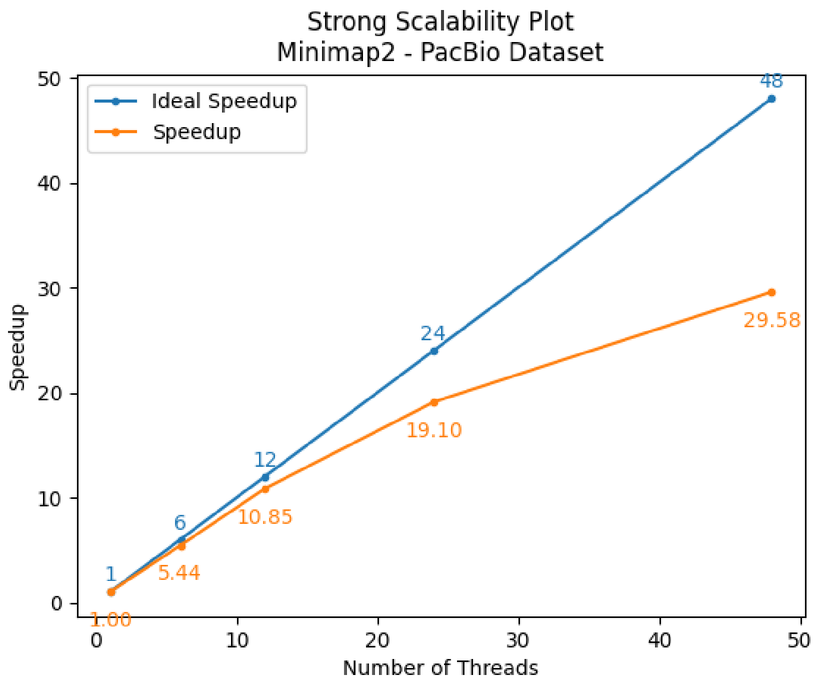

Figure 3.7: Scalability plot of Minimap2 using real PacBio dataset

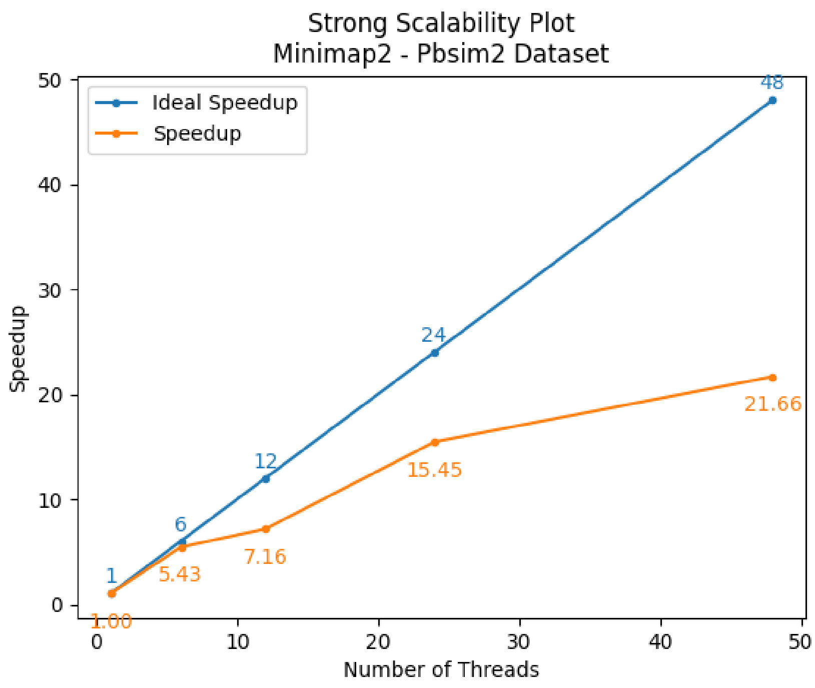

Figure 3.8: Scalability plot of Minimap2 using simulated Pbsim2 dataset

3.3 Bottleneck Analysis

In Bottleneck Analysis, we identify the most time-consuming functions of our genomic tools. We used the Hotspots Analysis option that is offered by the Intel VTune profiler to identify these functions. For our experiments, we used all the available threads (i.e., 48) on the node. We run the analysis for Bowtie2, Bwa-Mem2, and Minimap2 with the combinations of our previously defined input sequences. We grouped the most time-consuming function of each tool into three groups attending to the main three steps of genome mapping tools: seeding, chaining/filtering, and alignment/extension.

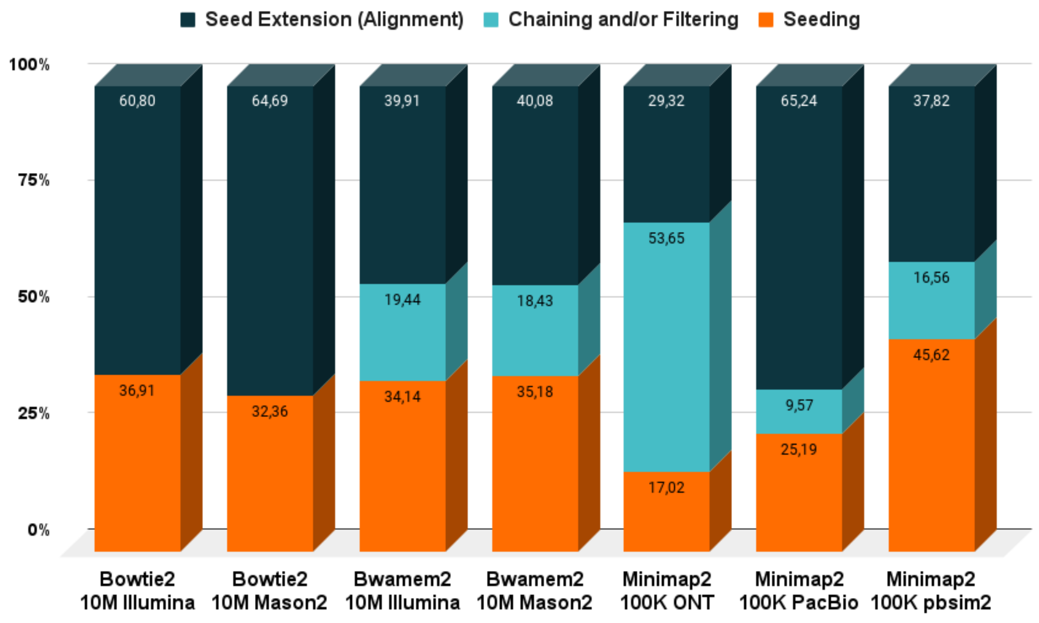

Figure 3.9: Bottleneck analysis of genomic tools executions

Figure 3.9 contains the results of the bottleneck analysis. For Bowtie2, both the real and the simulated datasets provide similar results. We can see that the alignment part is the most time-consuming. With Bwa-Mem2, we see that for both datasets, the seeding and the alignment parts share a similar percentage of execution time with the alignment stage being slightly larger. Interestingly, in Minimap2, the time distribution of the three modules varied across all the input data types. For instance, the chaining was the most time-consuming step for the ONT datasets, whereas PacBio CLR datasets required spending the majority of the time in the alignment phase. In that case of ONT reads, the chaining part is the most consuming and is attributed to the irregularity of workload of the ONT reads, which vary in read lengths [64]. The results obtained for Minimap2 and the datasets we have tested agree with previous research [64] published.

3.4 Microarchitecture Exploration

Microarchitecture exploration analysis of VTune collects a complete list of profiling events for analyzing a typical application. It calculates a collection of predefined ratios used for the metrics and identifies hardware-level performance problems. Roughly speaking, we divide superscalar processors into the front-end and back-end parts. In front-end the instructions are fetched and decoded into the operations. In back-end the required computation is performed. Each cycle, the front-end generates these operations (Uops) and places them into pipeline slots that then move through the back-end. For some time interval, vtune determines the maximum number of pipeline slots that could have been filled and issued during that time interval. This analysis splits all pipeline slots into four main categories: Front-End bound, Bad Speculation, Retired, and Back-End Bound. Figure 3.10 presents a two-fold chart that ends up with the four leaf categories used as high-level performance metrics.

In more detail, Front-End represents the first part of the processor core responsible for fetching operations that the Back-End part will later execute. Basically, the Front-End fetches instructions from the memory subsystem and decodes them into micro-ops (uOps). Front-End Bound metric defines a slots fraction where the processor's Front-End undersupplies its Back-End when there is no Back-End stall. For instance, we can categorize stalls due to instruction-cache misses as Front-End Bound.

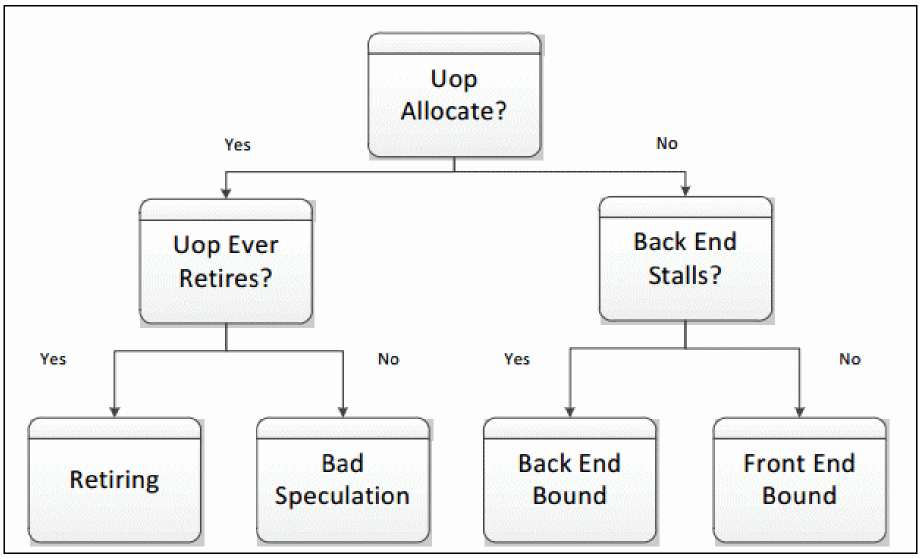

Figure 3.10: Chart of VTune's high-level metrics in Microarchitecture Exploration Analysis

The 'Bad Speculation' metric represents the percentage of pipeline slots wasted due to incorrect speculations. These can be slots used to issue uOps that do not eventually get retired. Also, it may be slots for which the recovery from earlier incorrect speculation has blocked the issue pipeline. For instance, we can categorize the wasted work due to mispredicted branches as Bad Speculation.

The 'Retiring' metric expresses the percentage of pipeline slots utilized by useful work, which describes the number of issued uOps that eventually retire. Maximizing Retiring typically increases the instructions per cycle. When we achieve the maximum possible number of uOps retired per cycle, the Retiring metric equals 100%.

Back-End represents the portion of the processor where an out-of-order scheduler dispatches uOps into execution units. Once completed, these uOps get retired according to program order. Back-End Bound metric expresses the percentage of pipeline slots where no uOps are being delivered due to a lack of required resources in the Back-End. For instance, we can classify as Back-End Bound the stalls due to data-cache misses.

Back-End Bound is further divided into two main categories: Memory Bound and Core Bound. Memory Bound measures the percentage of slots where the pipeline could be stalled due to demand load or store instructions. Core Bound represents how much Core non-memory issues were a bottleneck. Shortages in hardware compute resources or dependencies of software's instructions are categorized under Core Bound.

We applied this type of analysis for Bowtie2, Bwa-Mem2 and Minimap2, with the combinations of some of our predefined inputs sequences. Figure 3.11 depicts the fractions of slots that we ascribe to one of the five metric categories: Front-End Bound, Memory-Bound, Core-Bound, Bad Speculation, and Retiring.

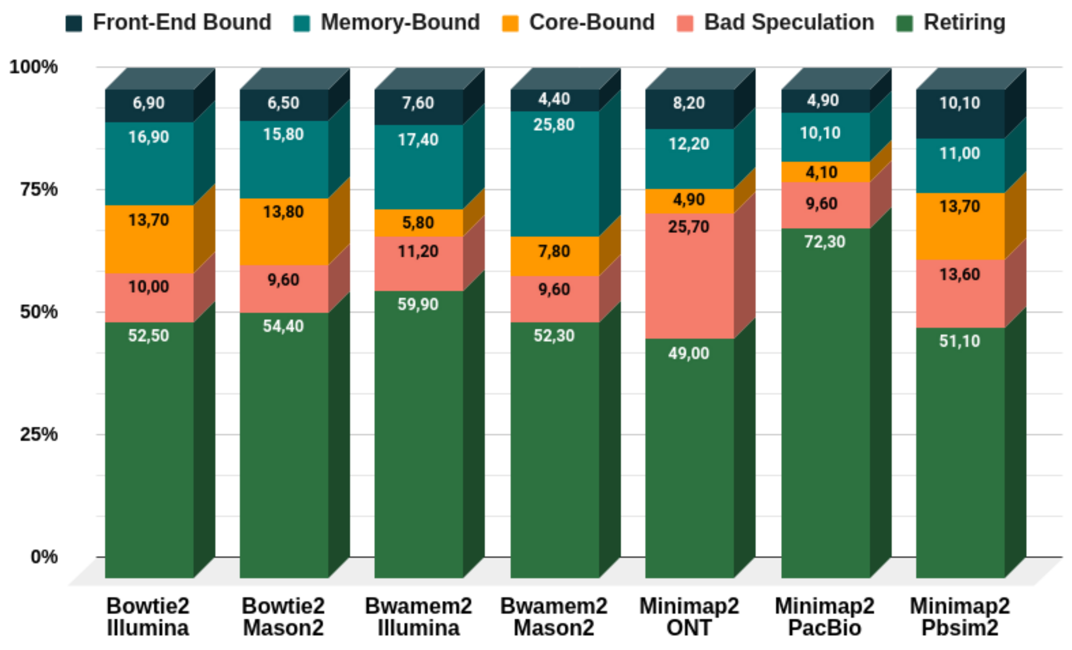

Figure 3.11: Microarchitecture exploration analysis chart of the executions

Again, the Bowtie2 executions follow the same behavior for both of the datasets. It is apparent that the retiring slot constitutes the most significant percentage of all. From VTune, we found that in Bowtie2, the functions that constitute the alignment part have a large value for the retiring metric. As we found that the majority of time is spent on the alignment part, it makes sense to have a larger value for the retiring metric for the total execution.

We see similar results for Bwa-Mem2. Note that the percentage of time spent on the seeding part for Bwa-Mem2 is similar to the one with Bowtie2. For Minimap2, we can see some differences. We know that the execution of the PacBio dataset spends most of the time during the alignment step in comparison with the other two datasets. For that dataset, we see a larger value on the retiring metric for the whole execution. We can see that there is a correlation where the alignment part, in general, has a higher Retiring metric value. From our exploration, we have concluded that the seeding part is more memory bound in comparison with the alignment part. This can be explained as the mapper needs to query the index data structure, which results in a sizeable memory footprint and irregular memory accesses.

3.5 Summary

In this chapter we performed a detailed performance analysis of widely used genome-mapping tools (Minimap2, Bwa-Mem2, and Bowtie2). This analysis offered insights into the mapping tools' scalability, bottlenecks, and microarchitecture behavior, in a Marenostrum 4 node. We also tested with datasets from different sequencing technologies. The result was that Bowtie2 scales very well up to the maximum of 48 threads tried in total. On the other hand, Bwa-Mem2, as well as Minimap2, starts having scalability problems after the use of 12 threads.

The bottleneck analysis indicated that most of the time, the alignment step, is the most consuming part of the execution. The only exception is when we use ONT reads in Minimap2. In that case, the chaining part is the most consuming and is attributed to the irregularity of workload of the ONT reads, which vary in read lengths [64]. The microarchitecture exploration indicated that the percentage of pipeline slots utilized by useful work (which describes the number of issued uOps that eventually retire) is very high in the alignment step. This can is attributed to the large number of computations during that step. On the other hand, the percentage of slots where the pipeline could be stalled due to demand load or store instructions (memory bound) is higher in the seeding part. This is usually due to the fact that, during the seeding stage, accesses to the genome's index (having a large memory footprint) often put pressure on the memory hierarchy and lead to memory inefficiencies.

Chapter 4

Accelerating Sequence Alignment

In the previous chapter, we discussed the results of the experimental evaluation for the genome-mapping tools. These results indicate that the alignment step (i.e., pairwise alignment or sequence to sequence alignment) is one of the most time-consuming step during most executions. In the following sections, we propose strategies for the acceleration of the alignment step.

4.1 Classical Sequence Alignment Algorithms

In most cases, the pairwise alignment problem is solved using some variation of the Needleman–Wunsch (NW) algorithm [52] (i.e., using gap-linear penalties) or the Smith–Waterman–Gotoh (SWG) algorithm [28] (i.e., using gap-affine penalties). These solutions are based on dynamic programming (DP) and require computing some recurrence equations on a DP matrix and, then, perform a traceback on the matrix to retrieve the optimal alignment.

Let us assume that we have two sequences consisting of characters A,T,G,C. The query $q = q_0q_1q_2...q_{n-1}$ which is a string of length $|q| = n$ and the text $t = t_0t_1t_2...t_{m-1}$ of length $|t| = m$. We define the pairwise global alignment problem as the computation of the alignment from $(0, 0)$ to (n, m) of a DP matrix, with minimum penalty score, allowing matching bases, substitutions, insertions and deletions. Under the gap-affine model, the alignment penalty score is computed based on the values of $\{a, x, o, e\}$, where a and x correspond to the penalty of matching or mismatching two bases, respectively, and the gap-penalty function is expressed as the linear function $g(n) = o + n \cdot e$, where o is the open-gap penalty, n is the length of the gap and e is the extend-gap penalty. Eq. 4.1 shows the recurrence relations of the DP matrix used in the SWG algorithm. Note that the majority of applications use strictly positive penalties ($x, o, e > 0$) and the mismatch penalty x stays the same throughout the whole alignment.

$$\begin{cases} I_{v,h} & = \min\{M_{v,h-1} + o + e, I_{v,h-1} + e\} \\ D_{v,h} & = \min\{M_{v-1,h} + o + e, D_{v-1,h} + e\} \\ M_{v,h} & = \min\{I_{v,h}, D_{v,h}, M_{v-1,h-1} + s(q_{v-1}, t_{h-1})\} \end{cases} \tag{4.1}$$

where,

$$s(v,h) = \left\{ \begin{array}{ll} a, & q_v = t_h \\ x, & q_v \neq t_h \end{array} \right\} \text{ for } 0 \leq v < n \text{ and } 0 \leq h < m$$

As a result, DP-based algorithms require quadratic time and memory on the length of the sequences to compute the optimal alignment. The quadratic execution time of these classical approaches quickly becomes the execution bottleneck, and these methods fail to scale with longer read lengths [47]. Moreover, intrinsic dependencies on the DP recurrences limit the effectiveness of vectorization approaches.

Both the KSW2 [72] algorithm (found at the heart of Minimap2 and Bwa-Mem) and Farrar's Smith-Waterman implementation [24] (found within Bowtie2) are based on traditional DP algorithms. These algorithms have limited capabilities for vectorization. For that reason, we are interested in a more flexible algorithm that not only accelerates the alignment process, but also enables the usage of SIMD instructions.

We propose to replace the classical DP-based algorithm with the recently proposed wavefront alignment algorithm (WFA) [47, 48]. WFA is an alignment algorithm that computes the exact alignment (i.e., computes the optimal alignment) between two sequences using gap-affine penalties. Unlike the traditional DP-based algorithms, this algorithm can be vectorized, enabling the efficient exploitation of SIMD instructions found on modern processors. In the following, we present the WFA, we propose a fully vectorized implementation, and we evaluate its performance compared to the scalar WFA implementation and other classical DP-based algorithms.

4.2 Wavefront Alignment Algorithm (WFA)

In short, WFA computes partial alignments of increasing score s until the optimal alignment is found. The WFA algorithm takes advantage of homologous regions between sequences to accelerate the alignment process. As a result, the WFA algorithm largely outperforms other state-of-the-art methods, requiring $O(n^s)$ time and $O(s^2)$ memory (where n is the length of the sequence and s is the optimal alignment score). In the following, we explain in more detail the algorithm and its properties.

4.2.1 Wavefront Furthest-Reaching Diagonals

WFA algorithm is based on the observation that diagonals on the DP-matrix have monotonically increasing scores. Intuitively, to compute the minimum score on a given cell of the DP-matrix, it is sufficient to compute the cells with a smaller score, as the values along each diagonal always increase. More formally, $\forall s$ (scores) and $\forall k$ (diagonals) we define as further reaching point $\mathcal{F}_{s,k}$, the DP-cell, on diagonal k and with score s, that is more far-off from the beginning of the diagonal. Also, a point $p = (v, h)$, in the diagonal $k = h - v$, is further than other $p' = (v - 1, h - 1)$ $(p' < p)$ in the same diagonal. For a given score s we define $\widetilde{I}_{s,k}$, $\widetilde{D}_{s,k}$, and $\widetilde{M}_{s,k}$ as the offsets in the diagonal to the f.r. point $\mathcal{F}_{s,k}$ for each of the three SWG matrices I, D, and M.

For a given score s, we define the s-wavefront (WF_s) as the set of all the further reaching points with score s, which is the set of offsets $\widetilde{I}_{s,k}$, $\widetilde{D}_{s,k}$, and $\widetilde{M}_{s,k}$, $\forall k$. Likewise, we call $\widetilde{I}_s$, $\widetilde{D}_s$, and $\widetilde{M}_s$, the components of the wavefront WF_s.

Assuming each component of a wavefront s ($\widetilde{I}_s$, $\widetilde{D}_s$, $\widetilde{M}_s$) is a vector of offsets centered over the main diagonal ($k = 0$), let $\widetilde{M}^{hi}$ denote the index of the rightmost diagonal in the component, and $\widetilde{M}^{lo}$ the index of the lowest.

The goal is to find the minimum score s, so as any of the f.r. points of WF_s reaches the point (n,m) in the DP matrix. We observe that for any s, the f.r. points of WF_s can only be originated from points whose score is $s-o$, $s-e$, $s-x$ or from a previous point with the same score s followed by matches along the diagonal. Essentially we can compute the set of f.r. points of WF_s using WF_{s-o}, WF_{s-e}, and WF_{s-x}. Considering only insertions, deletions, and mismatches; we can redefine Eq. 4.1 in terms of offsets to f.r. points (Eq. 4.2). Derived from Eq. 4.2, Figure 4.1 denotes the dependencies between wavefronts as to compute one element of the next wavefront.

$$\begin{aligned}
\widetilde{I}_{s,k} &= \max \left\{ \begin{array}{ll} \widetilde{M}_{s-o-e,k-1} & \text{(Open insertion)} \\ \widetilde{I}_{s-e,k-1} & \text{(Extend insertion)} \end{array} \right\} + 1 \\
\widetilde{D}_{s,k} &= \max \left\{ \begin{array}{ll} \widetilde{M}_{s-o-e,k+1} & \text{(Open deletion)} \\ \widetilde{D}_{s-e,k+1} & \text{(Extend deletion)} \end{array} \right\} \\
\widetilde{M}_{s,k} &= \max \left\{ \begin{array}{ll} \widetilde{M}_{s-x,k} + 1 & \text{(Substitution)} \\ \widetilde{I}_{s,k} & \text{(Insertion)} \\ \widetilde{D}_{s,k} & \text{(Deletion)} \end{array} \right\}
\end{aligned} \tag{4.2}$$

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

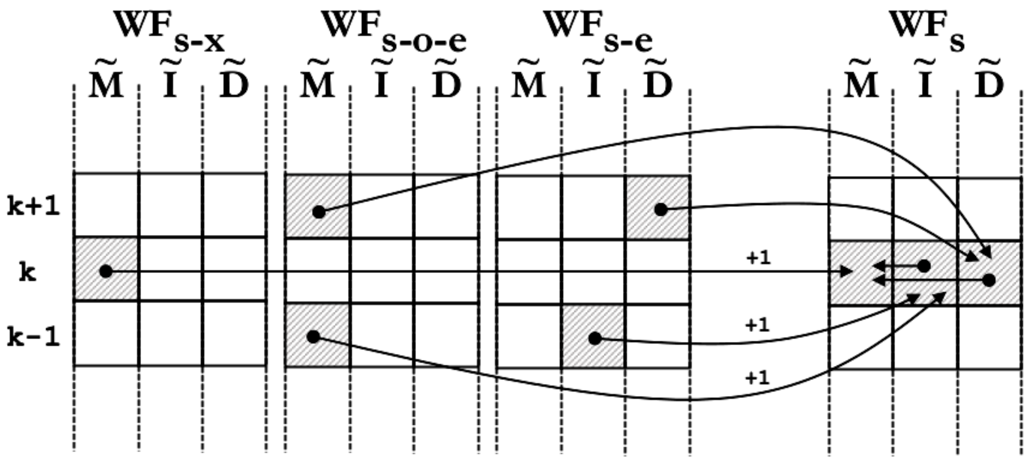

Figure 4.1: Dependencies between wavefronts as to compute one element of the next wavefront

4.2.2 Alignment Algorithm using Wavefronts

The WFA algorithm (Algorithm 1) progressively computes wavefronts of increasing score until the cell (n, m) is reached. $\mathcal{A}_k$ denotes the diagonal that the cell (n,m) belongs to. $\mathcal{A}_{offset}$ is initialized with the offset of cell (n,m) from the start. WFA starts by initializing $\widetilde{M}_{0,0} = 0$ and the best score s to 0.

Iteratively, for each score s, the algorithm extends the points $\widetilde{M}_{s,k}$ following matching characters along the diagonals, using WF_EXTEND$(\widetilde{M}_s, q, t)$ (Algorithm 2). Then, it checks whether any of the resulting f.r. points of wavefront WF_s reaches (n, m). If not, the algorithm proceeds to compute the next wavefront WF_{s+1} using WF_NEXT$(\widetilde{D}, \widetilde{I}, \widetilde{M}, q, t, s)$ (Algorithm 3, which applies Eq. 4.2), and iterates again.

Each new wavefront grows to span over one more diagonal on each end (i.e., $\widetilde{M}_s^{hi}$ and $\widetilde{M}_s^{lo}$) compared to the wavefronts it depends on. As a result, the size of each subsequent wavefront increases proportional to the alignment score between the sequences. Hence, the algorithm requires $O(s^2)$ memory to store all wavefronts. Also, note that extending a wavefront (WF_EXTEND function) is bounded by the number of diagonal matching characters $(\max\{n, m\})$ and the length of the wavefront. Similarly, the function WF_NEXT computes each next wavefront in

Algorithm 1: Gap-affine WFA algorithm algorithm

```
Input: $q, t$ strings, $p = \{x, o, e\}$ gap-affine penalties
Output: Gap-affine alignment $\mathcal{A}$ between $q$ and $t$ under $p$ penalties
Function WF_ALIGN($q, t, P$) begin
    // Diagonal and offset to $(n, m)$
    $\mathcal{A}_k \leftarrow (m - n)$ ;
    $\mathcal{A}_{offset} \leftarrow m$ ;
    // Initial conditions
    $\widetilde{M}_{0,0} \leftarrow 0$ ;
    // Incremental computation of wavefronts
    $s \leftarrow 0$ ;
    while true do
        // Exact extend s-wavefront
        WF_EXTEND($\widetilde{M}_s, q, t$) ;
        // Check exit condition
        if ($\widetilde{M}_{s,\mathcal{A}_k} \geq \mathcal{A}_{offset}$) then break ;
        // Compute wavefront for the next score
        $s \leftarrow s + 1$ ;
        WF_NEXT($\widetilde{M}, \widetilde{I}, \widetilde{D}, q, t, s$)
    // Backtrace alignment
    $\mathcal{A} \leftarrow$WF_BACKTRACE($\widetilde{M}, \widetilde{I}, \widetilde{D}, q, t, s$) return $\mathcal{A}$
```

time proportional to the wavefront length. Therefore, the running time of the WFA algorithm, to compute an alignment of score s, is bounded in the worst case by $O(\max\{n, m\} \cdot s)$, or $O(ns)$ assuming that the sequences have the same length.

Algorithm 3: Compute next wavefront

```
Input: $\widetilde{M}, \widetilde{I}, \widetilde{D}$ wavefronts, $q, t$ strings, $s$ score
Function WF_NEXT($\widetilde{M}, \widetilde{I}, \widetilde{D}, q, t, s$) begin
    $hi \leftarrow \max\{\widetilde{M}^{hi}_{s-x}, \widetilde{M}^{hi}_{s-o-e}, \widetilde{I}^{hi}_{s-e}, \widetilde{D}^{hi}_{s-e}\} + 1$ ;
    $lo \leftarrow \min\{\widetilde{M}^{lo}_{s-x}, \widetilde{M}^{lo}_{s-o-e}, \widetilde{I}^{lo}_{s-e}, \widetilde{D}^{lo}_{s-e}\} - 1$ ;
    for $k \leftarrow lo$ to $hi$ do
        $\widetilde{I}_{s,k} \leftarrow \max\{\widetilde{M}_{s-o-e,k-1}, \widetilde{I}_{s-e,k-1}\} + 1$ ;
        $\widetilde{D}_{s,k} \leftarrow \max\{\widetilde{M}_{s-o-e,k+1}, \widetilde{D}_{s-e,k+1}\}$ ;
        $\widetilde{M}_{s,k} \leftarrow \max\{\widetilde{M}_{s-x,k} + 1, \widetilde{I}_{s,k}, \widetilde{D}_{s,k}\}$ ;
```

Once the algorithm computes a wavefront that reaches (n, m), and therefore, it

Algorithm 2: Wavefront extend

```
Input: $\widetilde{M}_s$ wavefront, $q, t$ strings
Function WF_EXTEND($\widetilde{M}, q, t$) begin
    for $k \leftarrow \widetilde{M}^{lo}$ to $\widetilde{M}^{hi}$ do
        $v \leftarrow \widetilde{M}_{s,k} - k$ ;
        $h \leftarrow \widetilde{M}_{s,k}$ ;
        while $p[v] == t[h]$ do
            $\widetilde{M}_{s,k} \leftarrow \widetilde{M}_{s,k} + 1$ ;
            $v \leftarrow v + 1$ ;
            $h \leftarrow h + 1$ ;
```

can retrieve back the path that leads from $(0, 0)$ to (n, m) (backtracing). WFA's backtrace is performed across the wavefronts' offsets instead of using the DP matrix scores. On each step of the backtrace, the function determines which f.r. point, from the previous wavefronts, originated the current offset. The difference between the actual offset and the source is the total amount of matching characters between the two positions.

4.3 Exploiting SIMD Paralellism in WFA

As described previously, the wavefront algorithm splits its operation into two steps. The algorithm uses the wavefront-extend kernel when it needs to extend the offsets along the diagonal to the f.r. points based on the matching characters for each score s. The second part of the computation is the wavefront-compute kernel, which computes the next wavefront anytime it is needed.

First, we examine the option of vectorizing the code of the compute wavefront kernel as depicted in Listing 4.1. First, the code fetches the offsets and performs a loop peeling. After the two steps mentioned above, an opportunity appears to exploit parallelism. That is because all the code's diagonals (from lo to hi) have to compute the subsequent offsets. There is no loop-carried dependency as we always read from previous offsets arrays and store it to the next_offsets array. Two MAX operations comprise the body of the loop. This type of loops can even be auto-vectorized by the compiler if the user instructs the compiler to do so and assess that there are no dependencies between iterations. With ***#pragma GCC ivdep***, the programmer we can assert that there are no loop-carried dependencies that would prevent consecutive iterations of the following loop from being executed concurrently with SIMD instructions. Note that, if the compiler could not prove that vectorizing is safe due to data dependency, we could do assert so by using this directive.

```c
void edit_wavefronts_compute_wavefront(
  edit_wavefront_t* const wavefront,
  edit_wavefront_t* const next_wavefront,
  const int pattern_length, const int text_length, const int
   distance) {

  // Fetch wavefronts
  const int hi = wavefront->hi, lo = wavefront->lo;
  next_wavefront->hi = hi+1;
  next_wavefront->lo = lo-1;
  // Fetch offsets
  ewf_offset_t* const offsets = wavefront->offsets;
  ewf_offset_t* const next_offsets = next_wavefront->offsets;
  // Loop peeling (k=lo-1)
  next_offsets[lo-1] = offsets[lo];
  // Loop peeling (k=lo)
  const ewf_offset_t bottom_upper_del =
    ((lo+1) <= hi) ? offsets[lo+1] : -1;
  next_offsets[lo] = MAX(offsets[lo]+1,bottom_upper_del);
  // Compute next wavefront starting point
  int k;
  #pragma GCC ivdep
  for (k=lo+1;k<=hi-1;++k) {
     // const int del = offsets[k+1]; // Upper
     // const int sub = offsets[k] + 1; // Mid
     // const int ins = offsets[k-1] + 1; // Lower
     // next_offsets[k] = MAX(sub,ins,del); // MAX
    const ewf_offset_t max_ins_sub =
        MAX(offsets[k],offsets[k-1]) + 1;
    next_offsets[k] = MAX(max_ins_sub,offsets[k+1]);
  }
  // Loop peeling
  const ewf_offset_t top_lower_ins =
    (lo <= (hi-1)) ? offsets[hi-1] : -1;
  next_offsets[hi] = MAX(offsets[hi],top_lower_ins) + 1;
  next_offsets[hi+1] = offsets[hi] + 1;
}
```

Listing 4.1: Compute Wavefront (Edit Distance)

Using the ivdep directive enables the compiler to proceed without considering loop-carried dependencies. That way, we do not need to use SIMD intrinsics or other vectorization technique that would be much more time-consuming and error-prone. Alas, this is not quite the same for extend step of the WFA. The wavefront-extend kernel can not leverage the auto-vectorization features as its operation is more complex.

Listing 4.2 contains the sequential code of wavefront extend. For every diagonal k, the algorithm first fetches the offset on the current offset on the specific position. Then it acquires the pointers to the pattern and text blocks specified in the current diagonal. The next step is to compare 64-bit blocks (i.e., 8 characters) of both sequences: pattern and text. If all of them are the equal, the code compares the next chunk of the sequences. Each iteration increases the value of the offset on that diagonal by four. The algorithm repeats this process iteratively using a while-loop until the algorithm detects the first difference in the comparison. That will mean that the algorithm will exit the while loop and, after that, it will have to determine the actual number of equal characters in the chunk before the first difference. The algorithm repeats the same process for each of the diagonals in the wavefront.

For every diagonal k, the number of 64-bit blocks of sequences the algorithm needs to compare is unknown and, in most cases, irregular across diagonals. This irregularity is the main obstacle preventing the automatic vectorization by the compiler. Another problem is that the algorithm performs a gather operation in each iteration to bring from memory the chunks of the sequences that need to be compared. The compiler will attempt to gather and scatter memory positions, but this operations can be very inefficient in many processors. The only way around this issue is to explicitly use vector intrinsics, template classes, or assembly language. In this case, we have opted for doing it manually using intrinsics.

```c
void wavefront_extend(
  wavefront_aligner_t* const wf_aligner,
  wavefront_t* const mwavefront,
  const int lo,
  const int hi) {

  wf_offset_t* const offsets = mwavefront->offsets;
  int k;

  for (k=lo;k<=hi;++k) {
    // Fetch offset
    const wf_offset_t offset = offsets[k];
    if (offset == WAVEFRONT_OFFSET_NULL) continue;

    // Fetch pattern/text blocks
    uint64_t* pattern_blocks =
        (uint64_t*)(wf_aligner->pattern+WAVEFRONT_V(k,offsets[k])
  );
    uint64_t* text_blocks =
        (uint64_t*)(wf_aligner->text+WAVEFRONT_H(k,offsets[k]));

    // Compare 64-bits blocks
    uint64_t cmp = *pattern_blocks ^ *text_blocks;
    while (__builtin_expect(cmp==0,0)) {
      // Increment offset (full block)
      offsets[k] += 8;
      // Next blocks
      ++pattern_blocks;
      ++text_blocks;
      // Compare
      cmp = *pattern_blocks ^ *text_blocks;
    }

    // Count equal characters
    const int equal_right_bits = __builtin_ctzl(cmp);
    const int equal_chars = DIV_FLOOR(equal_right_bits,8);
    offsets[k] += equal_chars;
  }
}
```

Listing 4.2: Wavefront Extend Initial Code

Vectorization is a vital tool for improving the performance of code running on modern CPUs. It is the process of transforming an algorithm from performing on a single value at a time to operating on a larger set of values at once. Modern CPUs provide explicit support for vector operations where a single instruction is applied to multiple data (i.e., SIMD). There is a range of alternatives and tools for implementing vectorization. An established way to perform vectorization is using vector intrinsics. As for Intel's processors, vector registers appeared in 1997 with MMX instruction set, having 80-bit registers. After that, SSE instruction sets were released with 128-bit registers. In 2011, Intel released the Sandy Bridge architecture with the AVX instruction set with 256-bit registers. In 2016 the first AVX-512 CPU was released, with 512-bit registers.

Figure 4.2 depicts the evolution of register space. As a part of SSE, Intel introduced the 128-bit XMM [0,15] space in the 64-bit execution mode. In 2011 Intel extended the register space with YMM[0, 15] registers in the 64-bit mode to support the 256-bit AVX and AVX2 extensions. The lower 128 bits that belong to the YMM registers are aliased to the respective 128-bit XMM registers. Intel extended the register space with ZMM[0,31] AVX-512 register space in 64-bit mode. This ISA extension provides 32 512-bit wide registers [zmm0,zmm31]. Similarly, the lower 256 bits of the ZMM registers are aliased to the respective 256-bit YMM registers and the lower128 bits to the respective 128-bit XMM registers.

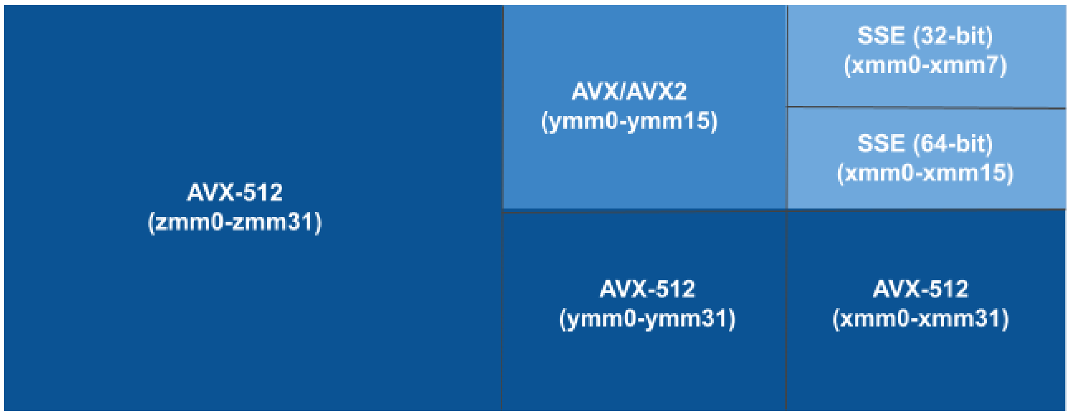

Figure 4.2: Evolution of register space

In this work, we decided to study the case of using Advanced Vector Extensions (AVX) to vectorize and accelerate the wavefront algorithm. In particular, we explore the use of AVX2 and AVX-512. The folling chapters contain the steps we took to achieve the final versions bot for AVX2 and AVX-512, as well as a final performance assessment.

4.4 Vectorization Using AVX-512 Intrinsics

Intel AVX-512 is a vectorization extension to the Intel x86 instruction set that can perform single instruction multiple data (SIMD) instructions over vectors of 512 bits. Intel AVX-512 doubles the width of the vector register in comparison to its predecessor AVX2, doubling the number of registers to decrease latency further. It also contains additional optimizations to accelerate tasks for modern workloads further. Three primary vector data types are specified, as shown in Figure 4.3. First is $\_\_m512$, which accounts for a 512-bit vector containing 16 floats. Then we have $\_\_m512d$, which describes a 512-bit vector containing eight doubles. Finally, we have $\_\_m512i$ 512-bit vector containing integers. An integer vector type can contain any type of integer, from chars to shorts to unsigned long longs. An _m512i may contain sixty-four chars, thirty-two shorts, Sixteen ints, or eight longs. These integers can be signed or unsigned.

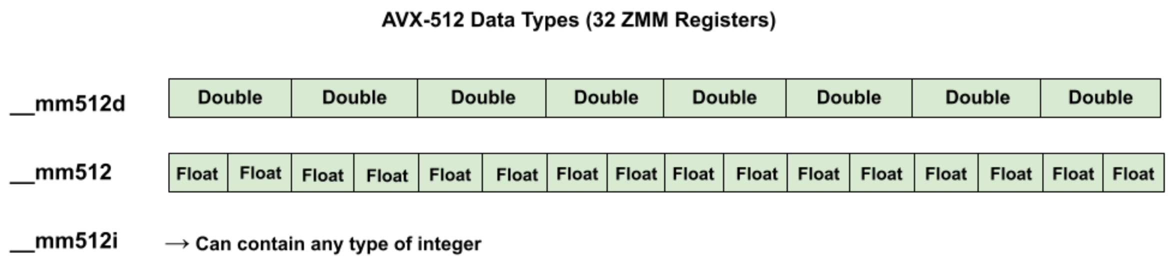
AVX-512 Data Types (32 ZMM Registers)
__mm512d
Double Double Double Double Double Double Double Double
__mm512
Float Float Float Float Float Float Float Float Float Float Float Float Float Float Float Float
__mm512i
→ Can contain any type of integer

Figure 4.3: AVX-512 Data Types

4.4.1 Initial Approach

Figure 4.4 depicts the first attempt to vectorize the extend-wavefront kernel. The idea is to take blocks of sixteen wavefront elements, representing the diagonals' offsets, and try to parallelize the computation using vectorization intrinsics. We can fit the values of the offsets of the diagonals into a vector register that can represent sixteen integer values (a total of 512 bits) for sixteen of the diagonals. For each diagonal k, the code fetches the correct part of text and pattern in order to compare them using vectors.

We can compare a maximum of four characters each time by fetching only 32 bits (4 characters) to fill in the vector register for all sixteen diagonals. The comparison for all diagonals returns a vector of integers representing the position of the first difference of the strings compared for each diagonal. For example, if we have a difference between the first characters of the sub-strings, we will give the zero value, but if the sub-strings are identical, it will get the value four (which represents that there is no difference).

Using that vector, we update the offsets of the wavefront and deactivate the lanes for which we have already found a difference. The algorithm repeats the process for the lanes (diagonals) in which the comparison returns the value four. When the algorithm reaches a point where no diagonal can extend more, it stores the new offsets and continues with the successive 16 (or fewer) diagonals. The following section explains the AVX-512 intrinsics used to vectorize the code.

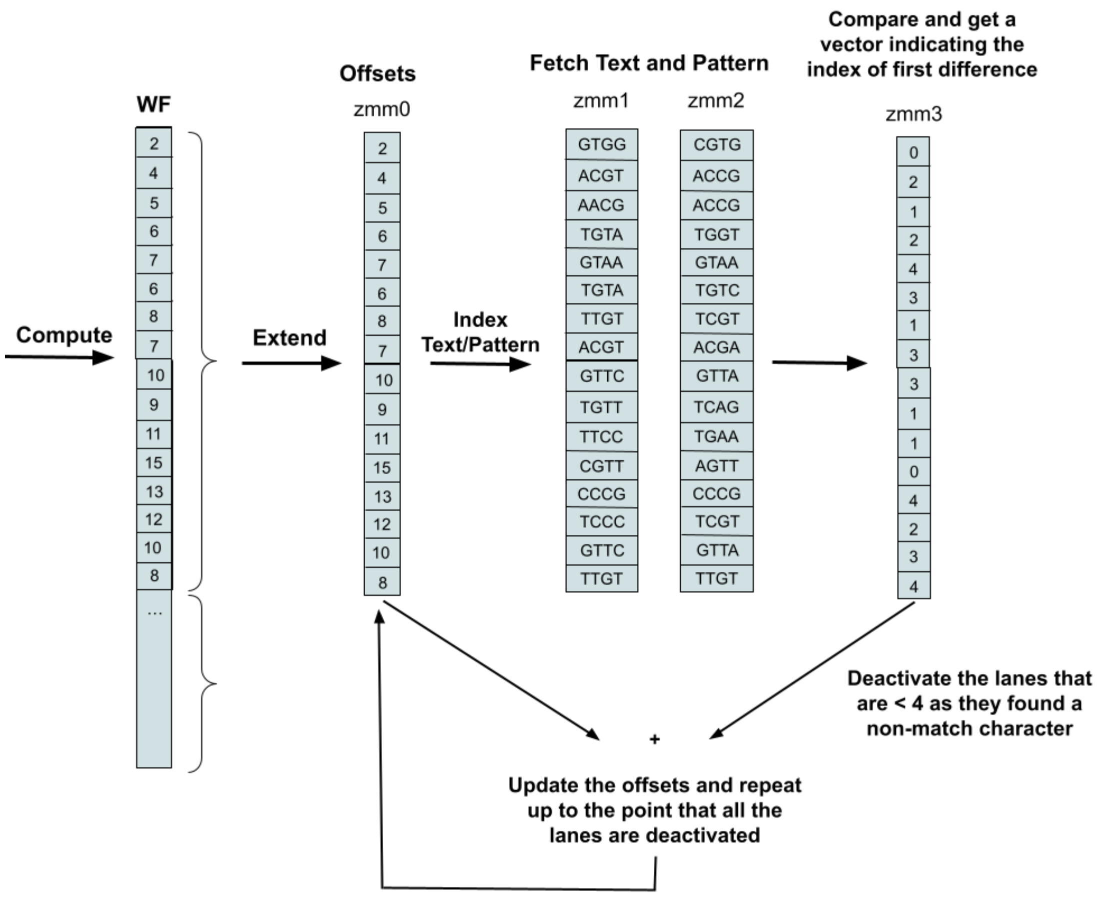
Compute
WF
Extend
Offsets
zmm0
Index Text/Pattern
Fetch Text and Pattern
zmm1
zmm2
Compare and get a vector indicating the index of first difference
zmm3
Deactivate the lanes that are < 4 as they found a non-match character
+
Update the offsets and repeat up to the point that all the lanes are deactivated

Figure 4.4: Abstract of AVX-512 vectorization strategy

4.4.2 Implementation

Listing 4.3 presents the first stage of the AVX-512 vectorized code implementation. After the initialization of the necessary parameters (offsets,k_min,k_max), there is a need to declare and initialize some important helping vectors that are used later in the code. We have, for example, the vector "$fours$", which contains a vector with just the value of four in each lane. The intrinsic $\_mm512\_set1\_epi32$ broadcasts 32-bit integer a to all elements of a vector. One of these vectors is also the $vecShuffle$ vector. We need to store 64 integers that will be used as indexes to a shuffle operation. In this scenario, we can use $\_mm512\_set\_epi8$, which sets packed 8-bit integers in the vectors with the supplied values.

We continue with the for-loop, which acts on packs of sixteen diagonals each time to parallelize their computation using vectorization. Having initialized ”k” with the minimum diagonal of the current repetition, we first initialize a vector *ks* that contains the indexes of consecutive diagonals. We also initialize a new vector *kmax_vector* that contains in each element the value of *k_max*. Using the intrinsic *_mm512_cmple_epi32_mask* we then compare the elements of vector *ks* that contains the indexes of the diagonals, with the elements of *kmax_vector* that contain the *k_max* value, for less-than-or-equal, and store the results in mask vector *mask*. We use the mask vector to deactivate the lanes where the index is over *k_max*.

Using the intrinsic *_mm512_maskz_loadu_epi32* we can then load from memory the offsets of the wavefront that corresponds to the current diagonals into the new vector called *offsets_vector*, using the vector mask of the previous step. The algorithm will use the *offsets_vector* to find the correct h and v values for each offset, which indicate the starting position in the strings of text and pattern accordingly.

The next step is to iteratively compare parts of the pattern and the text, as described in the previous section. We use a while-loop where in each iteration, we check if all the lanes are deactivated, which means that there is no diagonal left that needs to be extended. We check using the *_mm512_mask2int* function, which converts the bit mask vector into an integer. The transformation to an integer happens after performing first a zero extend computation on the 16-bit mask. When the value zero is returned, all the diagonals have finished with their computation, and we need to exit the while-loop.

Inside the while-loop, we first need each diagonal to fetch the parts of the pattern and text to compare them. As described in the previous section, each iteration of the while-loop compares four characters of pattern and text for each diagonal. These chunks of four characters equal 32 bits. Different indices for each diagonal, from where we need to fetch, force us to use a gather paradigm. We achieve that using the *_mm512_mask_i32gather_epi32* intrinsic, which gathers 32-bit integers from memory using 32-bit indices. These 32-bit integers represent a chunk of four consecutive characters of the pattern or text. It also uses the mask in order to perform this operation only for the lanes that are still activated.

After extracting the parts of pattern and text, a problem arises when we need to compare them: the intrinsic for the gather stores the integers with the reversed endianness of what we need to represent the four characters accurately. In order to change the endianness of each element of the vector, we use the $\_mm512\_shuffle\_epi8$ intrinsic, which shuffles packed 8-bit integers in the vector according to the shuffle control mask. Using the correct indices can allow us to virtually change the order of the represented characters by changing the endianness of the integer.

In order to compare the text and the pattern, we can perform an XOR operation using the intrinsic $\_mm512\_maskz\_xor\_epi32$. That produces a 1 in the character's position that differs between the two strings. Then by counting the zeros before we encounter the first '1', we can see how far the extension reached for each diagonal and also to which diagonals will need to continue the extension or be deactivated. We use the intrinsic $\_mm512\_maskz\_lzcnt\_epi32$, which counts the number of leading zero bits in each packed 32-bit integer in the vector. This information is stored in the vector called clz_vector.

With the clz vector, we will need to divide each element by eight to get the number of equal characters. We store these numbers in the $equal\_chars$ vector. We use the intrinsic $\_mm512\_maskz\_srli\_epi32$, which shifts packed 32-bit integers in the vector right by a specified number (here by three). This vector contains the number of positions where each diagonal extended its offset for that iteration. We can then add these values to the corresponding elements of $offsets\_vector$ to update the offsets. To prepare the computation for the next iteration of the while-loop, we also update the h and v vectors. At the end of the while-loop, we have to update the mask and deactivate the lanes where the elements in $equal\_chars$ are smaller or equal to four.

After we finally exit the while-loop, we store the offsets from the vector register back in the memory. The forenamed procedure will be repeated for the next chunk of diagonals until we reach k_max.

```
void extend_wavefront_vectorized(
    edit_wavefront_t* const wavefront,
    const char* const pattern,
    const int pattern_length,
    const char* const text,
    const int text_length,
    const int distance) {

    // Parameters
    ewf_offset_t* const offsets = wavefront->offsets;
    const int k_min = wavefront->lo;
    const int k_max = wavefront->hi;

    // Extend diagonally each wavefront point
    int k;

    // Helping vectors
    const __m512i fours = _mm512_set1_epi32(4);
    const __m512i pattern_length_vector = _mm512_set1_epi32(pattern_length);
    const __m512i text_length_vector = _mm512_set1_epi32(text_length);
    const __m512i zero_vector = _mm512_setzero_epi32();
    const __m512i vecShuffle =
    _mm512_set_epi8(60,61,62,63, 56,57,58,59, 52,53,54,55, 48,49,50,51,
                    44,45,46,47, 40,41,42,43, 36,37,38,39, 32,33,34,35,
                    28,29,30,31, 24,25,26,27, 20,21,22,23, 16,17,18,19,
                    12,13,14,15, 8,9,10,11,   4,5,6,7,     0,1,2,3);

    // Compute 16 elements at the same time, increment the k accordingly
    // Assume 32 bits elements
    const int elems_per_register = 16;
    for (k=k_min;k<=k_max;k+=elems_per_register) {
      // Create a vector register holding the diagonal numbers
      __m512i ks = _mm512_set_epi32 (k+15,k+14,k+13,k+12,k+11,k+10,k+9,k+8,
                                     k+7,k+6,k+5,k+4,k+3,k+2,k+1,k);

      // Create the mask used for comparing which elements are <= k_max
      __m512i kmax_vector = _mm512_set1_epi32 (k_max);
      __mmask16 mask = _mm512_cmple_epi32_mask (ks,kmax_vector);

      // Compute h and v
      // load 16 elements starting from offsets[k]
      __m512i offsets_vector = _mm512_maskz_loadu_epi32 (mask, &offsets[k]);
      __m512i h_vector = offsets_vector;
      __m512i v_vector = _mm512_maskz_sub_epi32 (mask, offsets_vector, ks);

      // Instrinsic to check if there's any bit set in the mask
      while(_mm512_mask2int(mask) != 0 ){
        // Gather 32 bits of text/pattern for each k
        __m512i pattern_vector = _mm512_mask_i32gather_epi32(zero_vector, mask,
                                                   v_vector, &pattern[0], 1);

        __m512i text_vector = _mm512_mask_i32gather_epi32(zero_vector, mask,
                                              h_vector, &text[0], 1);

        // Change endianess to make the xor + clz character comparison
        pattern_vector = _mm512_shuffle_epi8(pattern_vector, vecShuffle);
        text_vector = _mm512_shuffle_epi8(text_vector, vecShuffle);

        // Compare bases on each lane and get leading zeros on each lane
        __m512i xor_result_vector = _mm512_maskz_xor_epi32( mask,pattern_vector,
                                                     text_vector);
        __m512i clz_vector = _mm512_maskz_lzcnt_epi32 (mask,xor_result_vector);

        // Divide clz by 8 (1 character) to get the number of equal characters
        __m512i equal_chars =  _mm512_maskz_srli_epi32 (mask,clz_vector,3);

        //update offsets using equal_chars
        offsets_vector = _mm512_mask_add_epi32 (offsets_vector, mask,
                                          offsets_vector, equal_chars);

        // v+4 h+4
        v_vector = _mm512_maskz_add_epi32 (mask, v_vector, fours);
        h_vector = _mm512_maskz_add_epi32 (mask, h_vector, fours);

        // Only lanes with equal_chars == 4 continue (no mismatch found yet)
        mask = _mm512_mask_cmpeq_epi32_mask(mask, equal_chars, fours);
      }

      _mm512_storeu_epi32 (&offsets[k], offsets_vector);

    }
}
```

Listing 4.3: First Attempt of AVX-512 Implementation

4.4.3 Optimizations

In this section, we present the optimizations that improved the efficiency of our implementation. We implemented specific techniques and minor code alternations that helped us reach the final version of the AVX-512 implementation for the extension part of the wavefront algorithm.

4.4.3.1 Hybrid Implementation

After a statistical analysis using various input datasets, it was shown that, on average most of the lanes are deactivated from the first iteration of the while-loop.

Specifically, we created four representative datasets, which we used to measure the average percentage of times we needed more than one iteration on the vectorized while-loop. We found that the code needs to do a second iteration 5.43% of the time on average, while it needs to do a third iteration 0.03% of the time on average. Thus, after the first iteration, there is a big chance that they are only some (or none) of the offsets that still need to be extended. Using a vectorized code after the first iteration causes unnecessary overhead, which the remaining amount of work cannot mitigate. Therefore, we decided to implement another version where we only do one iteration of the while-loop to be vectorized. After the first iteration, the remaining diagonals that have not finished the extension continue to extend their offsets sequentially.

Listing 4.4 shows the altered part of the implementation. We use a while-loop to iterate through the diagonals. For efficiency reasons, we do not want to iterate through all the sixteen diagonals, only in the ones that are needed to be extended. For that reason, we use the following strategy. First, we use the *__builtin_ctz* function, which returns the number of trailing zeros in the 16-bit mask. That number also indicates the diagonal we need to extend in the current chunk of 16 diagonals. In order to access the correct global diagonal, we have to add this number (used as an offset) to the current minimum diagonal. Then, we just sequentially extend for the selected diagonal. After storing the new offsets for the specific diagonal, we then have to change the bit of the mask to zero for the diagonal that we investigated. Then we repeat the same procedure. That way, we only do the necessary repetitions.

```
void extend_wavefront_vectorized(...) {
  ...
  for (k=k_min;k<=k_max;k+=elems_per_register) {
    //Create a vector register holding the diagonal numbers
    //Create the mask for comparing which elements are <= k_max
    //Compute h and v
    //Load 16 elements starting from offsets[k]

    // ONE ITERATION OF VECTORIZED CODE

    //Gather 32 bits of text/pattern for each
    //Change endianness to make the xor + clz character comparison
    //Compare bases on each lane and get leading zeros on each lane
    //Divide clz by 8 to get the number of equal characters
    //Update offsets
    //v=v+4, h=h+4
    //Update mask
    //Store the new offset values back to the memory

    // SEQUENTIALLY

    int mask = _mm512_mask2int(vector_mask);
    while (mask != 0) {
      // tz contains the number of trailing 0s in the 16 bit mask
      int tz = __builtin_ctz(mask);
      // curr_k = current min k + the current selected diagonal
      int curr_k = k + tz;
      ...
      // Count equal characters
      const int equal_right_bits = __builtin_ctzl(cmp);
      const int equal_chs = DIV_FLOOR(equal_right_bits,8);
      offsets[curr_k] += equal_chs;
    }

    // With this operation we change the bit of the mask that
    //we just extended to zero
    mask &= (0xfffe << tz);
    }
  }
}
```

Listing 4.4: Hybrid Execution Optimization

4.4.3.2 Loop Unrolling of Remaining Diagonals

In general, the number of the diagonals to process can be arbitrarily large. Moreover, there is a good possibility that it will not be a multiple of sixteen, which is the number of elements that we vectorize for each outer iteration. That means there is a good possibility that we may create unnecessary overhead for the vectorization for the last diagonals that we need to examine. That happens because we will have to execute the intrinsics on the whole vector register even if we have only one diagonal left to extend, which is only one element in the register.

Loop peeling is the second optimization that was implemented. Listing 4.5 depicts the changes in the code. The idea is to sequentially extend the remaining diagonals that would not be multiple of sixteen. The remaining diagonals will be then multiple of sixteen. That way, we ensure that when we use the vectorization intrinsics, all the lanes will be available for useful work. In our implementation, we first compute the remaining iterations if we make a modulo of sixteen to the total number of diagonals. After computing and updating the new offsets for these diagonals, we add the value of the remaining diagonals to the *k_min* value. This will be the new initial value used as *k_min* for the vectorized for-loop.

```
void extend_wavefront_vectorized(...) {
  ...
  const int elems_per_register = 16;
  int num_of_diagonals = k_max-k_min+1;
  int loop_peeling_iters = num_of_diagonals%16;

  for(int i=k_min;i<k_min+loop_peeling_iters;i++){
    ...
    offsets[i] += equal_chs;
  }
  k_min+=loop_peeling_iters;
  for (k=k_min;k<=k_max;k+=elems_per_register) {
    // 1. ONE ITERATION OF VECTORIZED CODE
    // 2. SEQUENTIALLY EXTEND THE REMAINING DIAGONALS
  }
}
```

Listing 4.5: Loop Peeling Implementation

4.4.3.3 Other Code Improvements

Some other optimizations of the code result in the execution of fewer instructions. One of them is a change in how we assign the ks vector with the new diagonals in each iteration of the while loop. The change is depicted in Listing 4.6. Instead of updating all the elements in each iteration with new values using the *_mm512_set_epi32* intrinsic, we now assign only once before the for-loop, and in every iteration, we just add to each element a step value for the new successive indexes of diagonals which is equal to sixteen. We now use the intrinsic *_mm512_add_epi32*, which is very efficient (latency of one cycle). Also, we assign only once the *kmax_vector* before the for-loop starts.

Another small optimization is for the hybrid version to check the mask before continuing to the sequential extension of the remaining diagonals. If the mask is equal to zero, all the diagonals have been extended as much as possible during the vectorization step, so there is no need to continue.

```
void extend_wavefront_vectorized(...) {

  // 1. SEQUENTIALLY EXTEND THE REMAINING DIAGONALS

  __m512i ks = _mm512_set_epi32 (k-1,k-2,k-3,k-4,k-5,k-6,k-7,k-8,
                                 k-9,k-10,k-11,k-12,k-13,k-14,k-15,k-16);

  for (...) {
    ks =_mm512_add_epi32 (ks, vec_of_16s));

    // 2. ONE ITERATION OF VECTORIZED CODE

    if(mask == 0){
      continue;
    }

    // 3. SEQUENTIALLY EXTEND THE REMAINING DIAGONALS

  }
}
```

Listing 4.6: Other Code Optimizations

4.4.4 Performance Evaluation

In this section, we compare the different versions and optimizations used for the vectorization of the wavefront extend kernel. Table 4.1 contains the essential metrics of VTune's microarchitecture exploration analysis that we need to compare the various implementations of the code. In this analysis as well as the following ones we use the two simulated datasets D1 and D2. Their specifications can be found in Table 3.2.

We split the code into three main sections based on the final version of the code. First, we have the loop-peeling part, then there is the one iteration of the vectorized code, and finally, the scalar part where the extension of the remaining diagonals that needs a second iteration of extension takes place. We are more interested in three metrics acquired from VTune. First, the instructions retired metric represents how many instructions were completely executed. The second is called Clockticks, the basic unit of time recognized by each physical processor. The clockticks here are considered non-sleep. The third metric shown in the table is the clockticks per instructions retired (CPI) which is calculated by dividing the number of unhalted processor cycles (Clockticks) by the number of instructions retired. The smaller the CPI, the better.

Starting from the initial vectorized implementation, it is shown that the number of instructions decreased significantly with an average percentage decrease equal to 79.6%. We define percentage decrease as $Pd = \frac{\text{Starting Value-Final Value}}{|\text{Starting Value}|} \cdot 100$. Percentage decrease shows the loss of value from the original, expressed as a percentage regardless of units. Also, the clockticks were decreased giving us an average speedup equal to 1.3. In contrast, we observe an increase in the CPI. With the help of the rest of the optimizations, we decreased the total number of instructions retired up to an average percentage decrease equal to 89,94%. The number of clockticks is also reduced. We achieve an average speedup equal to 2.9 for the final vectorized wavefront-extend function compared to the scalar one.

Versions	**Sub-Sections**	**Instructions (B)**		**Clockticks (B)**		**CPI**	
	Error (%)	**D1**	**D2**	**D1**	**D2**	**D1**	**D2**
Scalar	Loop-Peeling	-	-	-	-	-	-
	SIMD	-	-	-	-	-	-
	Scalar	-	-	-	-	-	-
	Total	**167.9**	**336.5**	**23.5**	**31.5**	**0.140**	**0.094**
Initial Implementation	Loop-Peeling	-	-	-	-	-	-
	SIMD	34.4	68.1	22.6	20.2	0.659	0.297
	Scalar	-	-	-	-	-	-
	Total	**34.4**	**68.1**	**22.6**	**20.2**	**0.659**	**0.297**
Hybrid	Loop-Peeling	-	-	-	-	-	-
	SIMD	30.2	59.9	15.8	10.4	0.539	0.174
	Scalar	2.4	4.8	1.8	1.1	0.742	0.237
	Total	**32.7**	**65.1**	**17.6**	**11.6**	**0.539**	**0,178**
Loop-Peeling	Loop-Peeling	1.8	2.4	0.3	0.8	0.183	0.034
	SIMD	29.4	59.4	13.8	8.4	0.471	0.141
	Scalar	3.3	6.0	1.9	1.1	0.557	0.181
	Total	**34.6**	**68.3**	**16.1**	**9.6**	**0.466**	**0.141**
Other Optimizations	Loop-Peeling	1.8	2.6	0.3	0.1	0.216	0.056
	SIMD	14.2	28.6	13.0	7.1	0.915	0.247
	Scalar	1.0	1.7	1.1	0.5	1.026	0.268
	Total	**17.1**	**33.4**	**14.5**	**7.7**	**0.840**	**0.232**

Table 4.1: Performance metrics for the different stages of optimizations (Instructions and Clocktics are expressed in Billions)

Even if there is a decrease in the execution time, it is not as close to the ideal achievable speedup using AVX-512. Based on the size of the registers we used, the maximum achievable speedup is equal to sixteen. In the final version of the code (Table 4.1) we conclude that the most significant part of the execution time is spent on the vectorized part (89% of the time). So the scalar parts are not the bottleneck. In Appendix A, we can see the assembly code for the primary part that uses AVX-512 intrinsics. The main reason for the latency seems to be the gather function which has a considerable latency of twenty-five cycles. That seems to affect the execution the most. The problem here is that the next instruction depend on the gathers which make the big latency significant. We also have a move operation that takes eight cycles that we may need to eliminate. In the next section, we explore the use of AVX2, which is supported by more processors and can ideally provide us with more negligible latency for the respective intrinsics, such as the intrinsic for the gather, and probably the use of alternative/different intrinsics, which can provide us with less overall latency.

4.5 Vectorization using AVX2 Intrinsics

Advanced Vector Extensions 2 (AVX2) is a vectorization extension to the Intel x86 instruction set that can perform single instruction multiple data (SIMD) instructions over vectors of 256 bits. Figure 4.5 depicts the three main vector data types supported. First is *__m256*, which accounts for a 256-bit vector containing eight floats. Then we have *__m256d*, which describes a 256-bit vector containing four doubles. Finally, we have *__m256i* 256-bit vector containing integers. An integer vector type can contain any type of integer, from chars to shorts to unsigned long longs. An _m256i may contain 32 chars, 16 shorts, eight ints, or four longs. These integers can be signed or unsigned.

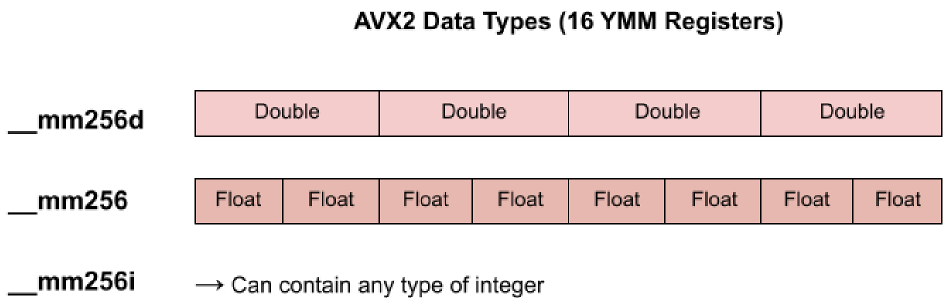

Figure 4.5: AVX2 Data Types

4.5.1 Implementation

To achieve vectorization using the AVX2 intrinsics, we followed the same steps as we did for AVX-512. We follow the same strategy as Figure 4.4 with the main difference being that we have half the capacity in the vector registers. That being said, in each iteration of the outer for-loop of the diagonals, we only process eight diagonals at a time. We extended this idea and implemented the same optimizations as we did with AVX-512.

Most of the intrinsics offered for AVX-512 in our implementation also have a counterpart version for AVX2. For example in AVX-512 we have the intrinsic $\_mm512\_add\_epi32$ while in AVX2 we have $\_mm256\_add\_epi32$. Both add packed 32-bit integers of two vector registers and assign the result to another vector register. In general, the intrinsics in AVX2 will have lower latency than the corresponding intrinsics in AVX-512. Also, the CPI of AVX2 intrinsics is equal to or lower than the corresponding ones in AVX-512.

Appendix B presents the assembly code for the primary part that uses AVX2 intrinsics. We can see that the latency of the gather functions is smaller but with not that much difference. We also see a decreased CPI for some of the instructions.

Table 4.2 depicts the performance results we acquired from VTune. We can see again that the reduction in both the retired instructions and the clockticks is noticeable. Specifically, we have an average percentage decrease of 83% for the number of instructions. The total clockticks are also reduced giving us an average speedup equal to 3.4 for the total execution. We see an improvement compared to AVX-512, but still, we are far from the ideal speedup (equal to eight when using AVX2).

Versions	**Sub-Sections**	**Instructions (B)**		**Clockticks (B)**		**CPI**	
	Error (%)	**D1**	**D2**	**D1**	**D2**	**D1**	**D2**
Scalar	Loop-Peeling	-	-	-	-	-	-
	SIMD	-	-	-	-	-	-
	Scalar	-	-	-	-	-	-
	Total	**167.9**	**336.5**	**23.5**	**31.5**	**0.140**	**0.094**
AVX2 Base Implementation	Loop-Peeling	1.1	1.3	0.1	0.7	0.100	0.051
	SIMD	26.5	48.1	8.3	6.1	0.316	0.126
	Scalar	1.0	7.0	0.7	1.1	0.587	0.170
	Total	**28.6**	**56.7**	**9.1**	**7.3**	**0.317**	**0.130**

Table 4.2: Performance evaluation of the first version (optimized) with AVX2 (Instructions and Clocktics are expressed in Billions, CPI)

4.5.2 Unrolling a Second Iteration of Vectorized Code

We know that the gather intrinsic is still the bottleneck. Figure 4.6 depicts the execution trace of the instructions for two iterations of the outer for-loop where the vectorized code uses the gathers. Because of the dependencies between each iteration, in the second iteration, the gather functions take a long to be dispatched and eventually retired completely. So the gather functions in the second iteration are retired on cycle ninety.

In this figure, the colors of the pipeline have the following meaning. Light blue means that the instruction is predecoded, and darker blue means that it is added to IDQ. Light red means the instruction is issued, and dark red means it is ready for dispatch. Orange means it is eventually dispatched, yellow means it is executed, and green means it is retired.

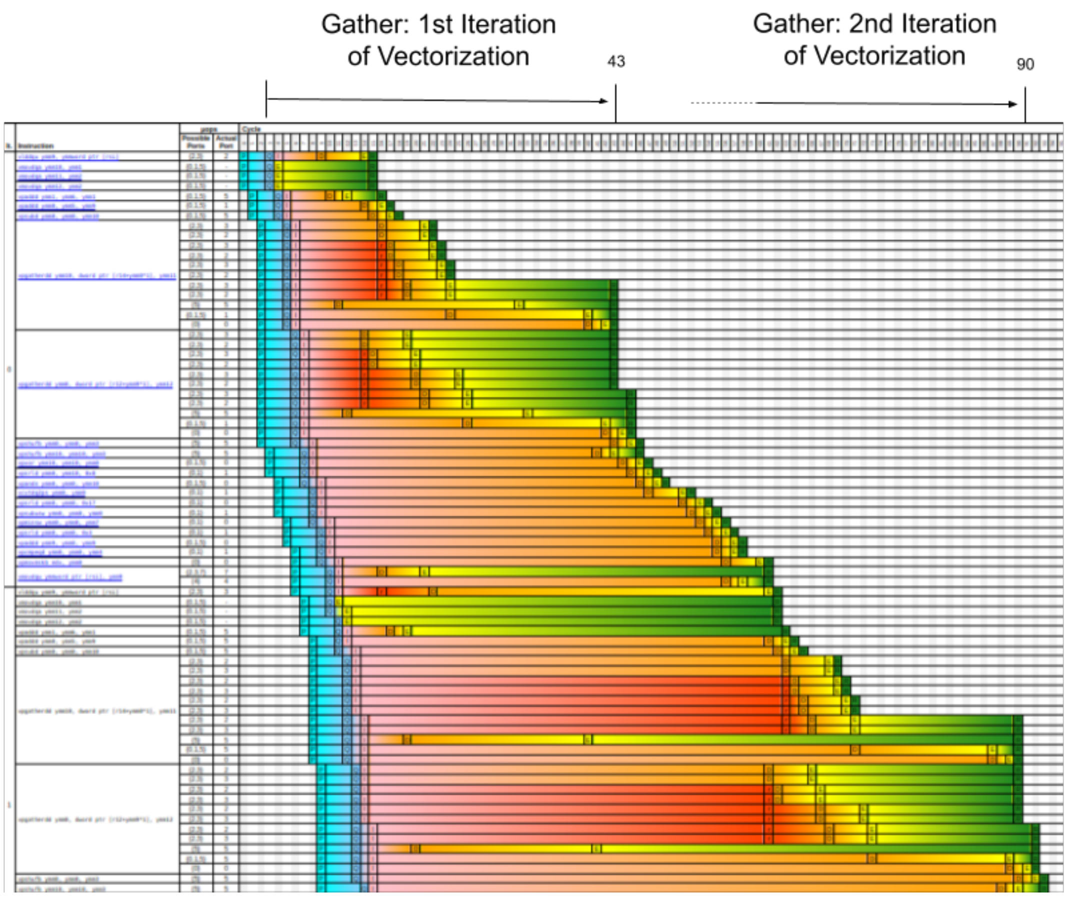

Figure 4.6: Execution trace of vectorized part with AVX2

One way to decrease this delay is to use loop-unrolling of the outer for-loop of the execution as shown in Listing 4.7. The loop is unrolled by a factor of 2. So, for every outer for-loop iteration of the diagonals, we extend sixteen diagonals (instead of eight) using a duplicate code for two blocks of eight elements. These blocks of instructions use different sets of variables. As we use duplicate code adapted for the two blocks of elements, we reduce the dependencies between these two blocks.

```
void extend_wavefront_vectorized(...) {
  // 1. SEQUENTIALLY EXTEND THE REMAINING DIAGONALS

  for (k=k_min;k<=k_max;k+=16) {
    // 2. ONE ITERATION OF VECTORIZED CODE
```

```
            // HERE WE HAVE DUPLICATE CODE
            // ACTING OVER TWO BLOCKS OF 8 ELEMENTS

      // 3. SEQUENTIALLY EXTEND THE REMAINING DIAGONALS

   }
}
```

Listing 4.7: AVX2 Loop Unrolling

The previous dependencies hold every other iteration of the initial outer loop instead of each iteration. That allows us to dispatch the two gathered instructions of the second iteration earlier, achieving better pipelining and, eventually, better throughput. We can see that effect in Figure 4.7 where we see the execution trace when using loop-unrolling for the outer for-loop. Now the gather functions of the second iteration (from the initial implementation) have been retired by cycle fifty-three. This is a significant improvement. This pipelining of the gather functions will happen every other iteration of the initial implementation.

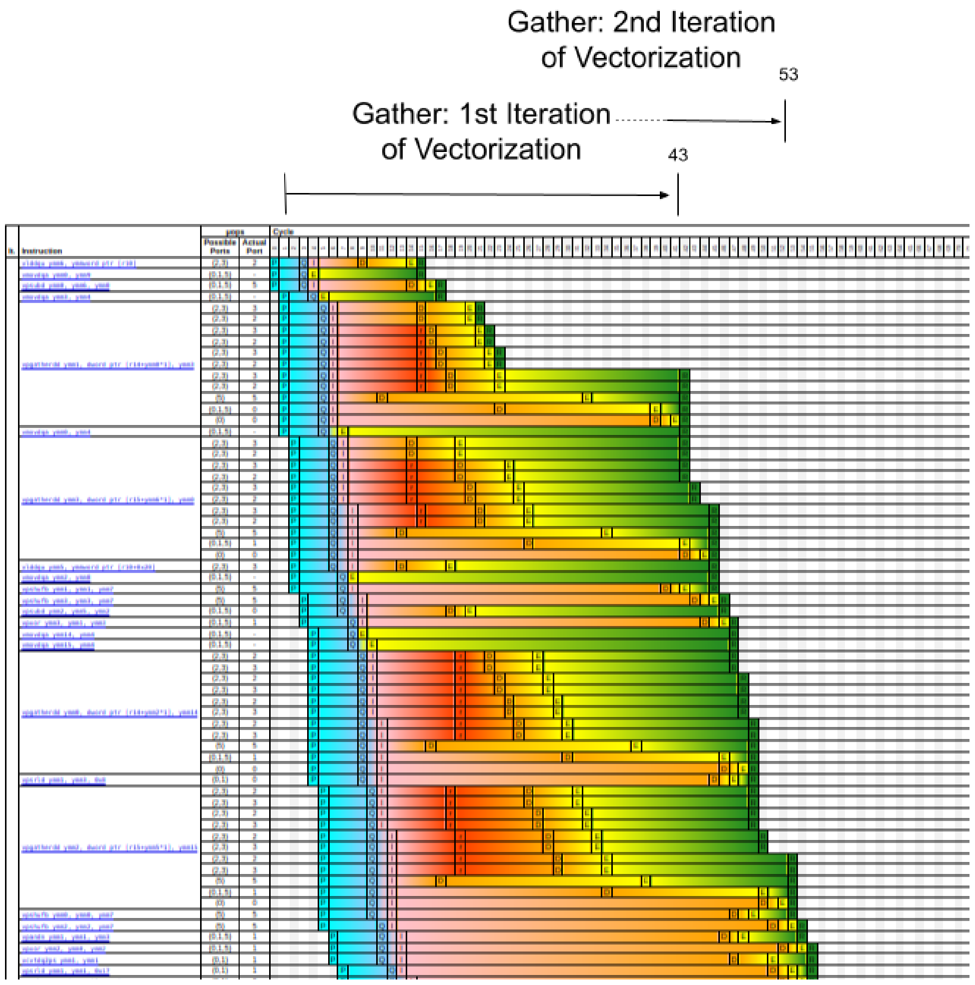

Figure 4.7: Execution trace of vectorized part with AVX2 using loop unrolling

4.5.3 Performance Evaluation

Table 4.3 depicts the performance results we acquired from VTune, comparing the two versions of the AVX2 implementation. The loop-unrolling technique has improved slightly the execution time of the vectorized part providing us with an average speedup of 1.2 for the vectored part between the loop-unrolling version and the basic version with AVX2. In general, comparing the final version of AVX2 with the scalar version, we have an average percentage decrease in the number of instructions equal to 83.5% and an average speedup for the clockticks equal to 3.6 for the overall execution of the extend-wavefronts kernel.

Versions	**Sub-Sections**	**Instructions (B)**		**Clockticks (B)**		**CPI**	
	Error (%)	**D1**	**D2**	**D1**	**D2**	**D1**	**D2**
Scalar	Loop-Peeling	-	-	-	-	-	-
	SIMD	-	-	-	-	-	-
	Scalar	-	-	-	-	-	-
	Total	**167.9**	**336.5**	**23.5**	**31.5**	**0.140**	**0.094**
AVX2 Base Implementation	Loop-Peeling	1.1	1.3	0.1	0.7	0.100	0.051
	SIMD	26.5	48.1	8.3	6.1	0.316	0.126
	Scalar	1.0	7	0.7	1.1	0.587	0.170
	Total	**28.6**	**56.7**	**9.1**	**7.3**	**0.317**	**0.130**
Loop Unrolling	Loop-Peeling	**1.9**	2.5	0.2	0.2	0.100	0.057
	SIMD	23.7	46.3	6.4	6.1	0.272	0.132
	Scalar	2.1	5.6	0.9	1.2	0.469	0.203
	Total	**27.8**	**54.7**	**7.5**	**7.5**	**0.274**	**0.153**

Table 4.3: Performance evaluation of the loop-unrolling version of AVX2 (Instructions and Clocktics are expressed in Billions)

4.6 Comparison of Vectorized WFA Implementations

This section compares the basic metrics between our implementations in AVX2 and AVX-512. In Table 4.4 we have gathered this information. Both AVX2 and AVX-512 implementations are faster than the scalar implementation, with speedups of 3.6 and 2.9, respectively. We have implemented the same optimizations in the AVX-512 and AVX2 base implementation, and we see that the AVX2 implementation is considerably faster.

This can be explained using the appendices A and B. Here we see that the instructions of AVX2 have smaller latency and CPI in general. Sometimes the CPI of AVX2 instructions can be even half the one we have in AVX-512. For example, the gather instruction has a CPI equal to eight in AVX-512 and a CPI equal to 4 in AVX2. This has a significant effect on the total execution time. We see that the average CPI of the wavefront extend when using AVX-512 is equal to 0.54.

This is higher compared to the average CPI for the AVX2 base implementation, which equals to 0.23. That means that even if the AVX-512 implementation has fewer total instructions retired, the difference in CPI will give the advantage to the AVX2 version. The AVX2 base implementation is already faster than the corresponding AVX-512 implementation, using the same optimizations. On top of that, we tried the loop-unrolling technique to reduce the dependencies between the iterations of the outer for-loop of the diagonals.

Versions	**Sub-Sections**	**Instructions (B)**		**Clockticks (B)**		**CPI**	
	Error (%)	**D1**	**D2**	**D1**	**D2**	**D1**	**D2**
Scalar	Loop-Peeling	-	-	-	-	-	-
	SIMD	-	-	-	-	-	-
	Scalar	-	-	-	-	-	-
	Total	**167.9**	**336.5**	**23.5**	**31.5**	**0.140**	**0.094**
AVX-512	Loop-Peeling	1.8	2.6	0.3	0.1	0.216	0.056
	SIMD	14.2	28.6	13.0	7.1	0.915	0.247
	Scalar	1.0	1.7	1.1	0.5	1.026	0.268
	Total	**17.1**	**33.4**	**14.5**	**7.7**	**0.840**	**0.232**
AVX2	Loop-Peeling	1.9	2.5	0.2	0.2	0.100	0.057
	SIMD	23.7	46.3	6.4	6.1	0.272	0.132
	Scalar	2.1	5.6	0.9	1.2	0.469	0.203
	Total	**27.8**	**54.7**	**7.5**	**7.5**	**0.274**	**0.153**

Table 4.4: Comparison of AVX implementations (Instructions and Clocktics are expressed in Billions)

4.7 Comparison of Sequence Alignment Implementations

We have to calculate the potential speedups we could reach for the genome-mapping tools (Minimap2, Bwa-Mem2), if we substitute their alignment algorithm (KSW2) with the vectorized WFA implementation. Table 4.5 presents a performance evaluation of the WFA versions compared to the KSW2 algorithm. We use real datasets (Illumina, PacBio, Nanopore) and four simulated ones (D1, D2, D3, D4). KSW2 is the current version of KSW used in the genome tools. WFA (no SIMD) represents the version of WFA where we do not use any form vectorization. WFA (partial-SIMD) only uses auto-vectorization in the compute part of the algorithm. WFA+AVX2 uses auto-vectorization in the algorithm's compute part and AVX2 vectorization intrinsics for the extend part. The WFA+AVX2 also uses auto-vectorization in the compute part, using this time AVX-512 vectorization intrinsics. For all the datasets, the AVX vectorized versions of WFA deliver the best results.

	Time(s)						
	Illumina	**PacBio**	**Nanopore**	**D1**	**D2**	**D3**	**D4**
KSW2	1194.6	315457.4	83425.7	1888.8	1890.6	53.8	56.3
WFA (no SIMD)	34.3	750.0	28726.2	140.4	906.0	8.6	32.5
WFA (partial-SIMD)	32.9	509.4	17626.8	88.8	567.0	7.9	25.1
WFA+AVX2	32.6	391.8	**12021.6**	**61.2**	**390.0**	7.9	**23.1**
WFA+AVX512	**28.8**	**381.6**	12505.8	**61.2**	390.6	**7.8**	**23.1**

Table 4.5: Execution Time of KSW and WFA implementations

	Speedup With Respect To KSW2						
	Illumina	**PacBio**	**Nanopore**	**D1**	**D2**	**D3**	**D4**
WFA (no SIMD)	34.8	420.6	2.9	13.5	2.1	6.2	1.7
WFA (partial-SIMD)	36.3	619.3	4.7	21.3	3.3	6.8	2.2
WFA+AVX2	36.7	805.1	6.9	30.9	4.8	6.8	2.4
WFA+AVX512	41.5	826.7	6.7	30.9	4.8	6.9	2.4

Table 4.6: Speedups of WFA versions with respect to KSW2

Table 4.6 presents the speedups of the different WFA versions with respect to the KSW2. The speedups indicate a significant improvement in execution time. The lower speedup is 2.4 for the D4 dataset of short reads. The most significant speedup is equal to 826.7, and we achieve it for the PacBio dataset, which consists of long sequences.

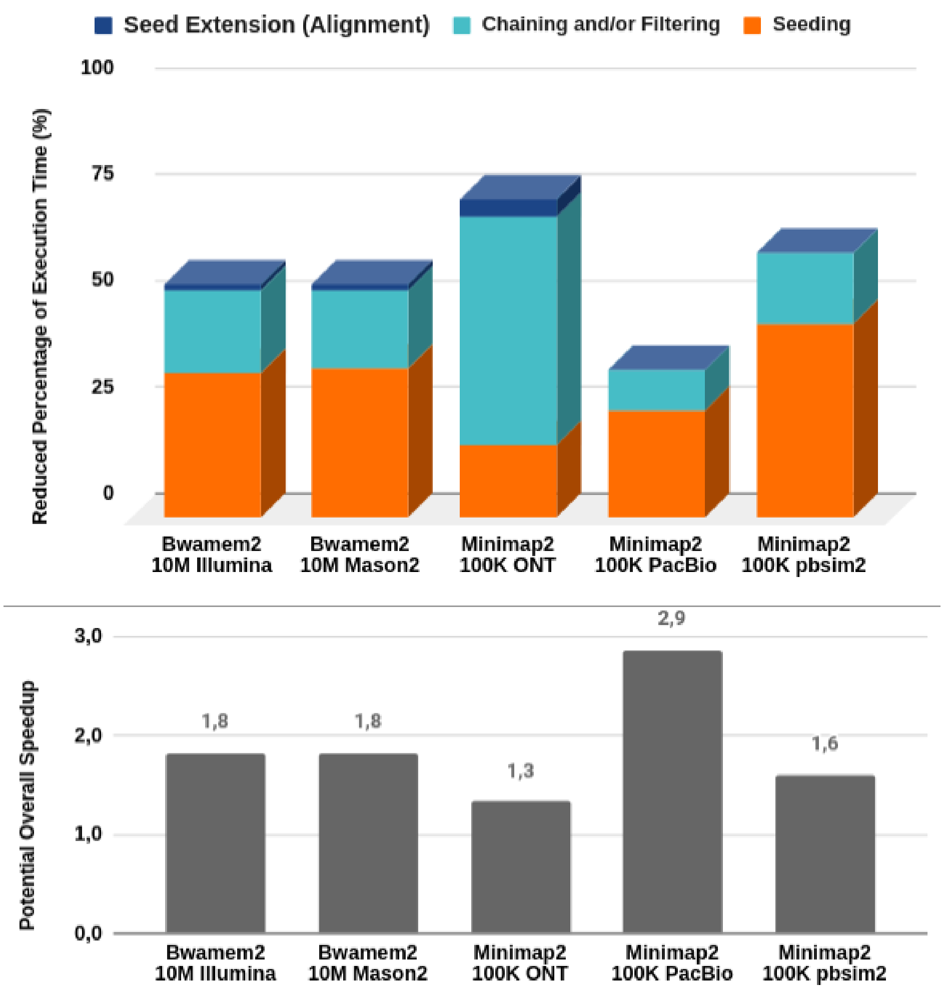

Figure 4.8: Theoretical percentage decrease based on the initial bottlenecks characterization and potential overall speedup of genome-mapping tools using the vectorized WFA algorithm

Figure 4.8 depicts the potential speedups for the genome mapping tools using our previous hotspots analysis. The percentage of the alignment in the overall execution decreased dramatically, specifically for the PacBio dataset, where we have a significant speedup, using vectorized WFA. We can see that the tools have potential speedups from 1.3 to 2.9 if we substitute the KSW2 algorithm with the vectorized WFA.

Chapter 5

Conclusions

In this thesis, we presented a thorough performance analysis of widely-used genome-mapping tools (Minimap2, Bwa-Mem2, and Bowtie2). This work offered insights into the mapping tools' scalability, performance bottlenecks, and microarchitecture exploitation. In turn, this analysis motivated the necessity of accelerating the alignment step of the genome-mapping tools. In particular, the sequence alignment kernel of these tools is based on traditional DP-based algorithms, like Smith-Waterman-Gotoh (i.e., KSW2), which require quadratic time and memory on the length of the sequences. As a result, the computational requirements of classical DP-based approaches quickly becomes a critical performance bottleneck, and most of these methods fail to scale with longer read lengths [47]. Moreover, intrinsic dependencies on the DP recurrences limit the effectiveness of vectorization approaches.

Consequently, the aim of this work is to motivate more efficient algorithms that allow accelerating genome-mapping tools. In this thesis, we proposed the utilization of the WFA algorithm to replace classical DP-based alignment algorithms. Moreover, as we showed before, the WFA algorithm can be effectively vectorized using SIMD instructions. In particular, both stages of the WFA algorithm (i.e., extend and compute) are suitable for vectorization. The first stage (WFA compute) can be easily vectorized using the auto-vectorization features of modern compilers. However, the second stage (WFA extend) requires a fine-tuned tailored vectorized implementation. This way, we proposed two SIMD implementations using Intel's AVX2 and AVX512 instructions.

By utilizing the vectorized WFA algorithm we can accelerate the genome-mapping tools (Minimap2 and Bwa-Mem2) that use the widely-used DP-based algorithm KSW2, for their alignment step. Specifically, our evaluation indicated that our vectorized AVX implementation achieves speedups from 2.4× up to 826.7× compared to KSW2. In return, this can yield significant potential speedups for the genome tools mentioned above, from 1.3 up to 2.9.

5.0.1 Publications

In the context of the work developed for this thesis, the author has contributed to the elaboration of a scientific paper and a poster. Moreover, he is working on the preparation of a manuscript derived from this work's thesis.

- **Quim Aguado-Puig**, Santiago Marco-Sola, Juan Carlos Moure, Christos Matzoros, David Castells-Rufas, Antonio Espinosa, and Miquel Moreto. "WFA-GPU: Gap-affine pairwise alignment using GPUs." bioRxiv (2022) [2]
- WFA-GPU: Accelerated gap-affine pairwise alignment, 21st European Conference on Computational Biology, 2022 (Poster).

5.1 Future work

In the previous section, we discussed the potential speedups we could attain by adequately integrating the vectorized WFA implementation into state-of-the-art genome-mapping tools. Currently, we are working on the integration of these WFA accelerated implementation into Minimap2 and Bowtie2. Further work involves the analysis and integration into other bioinformatics tools that perform extensive usage of pairwise alignment primitives. In the future, we would be interested in examining the possibility of further accelerating WFA on multicores (using intraparallelization techniques). Furthermore, we could extend our work into the optimization of other critical kernels (like seeding, chaining, and filtering)

5.2 Financial and Technical Support

This thesis was supported by the European Unions's Horizon 2020 Framework Programme under the DeepHealth project [825111], by the European Union Regional Development Fund within the framework of the ERDF Operational Program of Catalonia 2014-2020 with a grant of 50% of total cost eligible under the DRAC project [001-P-001723] and Lenovo-BSC Contract-Framework Contract (2020). It was also supported by the Spanish Ministerio de Ciencia e Innovacion MCIN AEI/10.13039/501100011033 under contracts PID2020-113614RB-C21 and TIN2015-65316-P and by the Generalitat de Catalunya GenCat-DIUiE(GRR) (contracts 2017-SGR-313, 2017-SGR-1328, and 2017-SGR-1414).

Chapter 6

Appendices

Appendix A: Latency of AVX-512 instructions for the primary part of vectorization

uOps	Latency	CPI	Instructions
1	1	0.33	vpaddd %zmm5, %zmm9, %zmm1
1	4	1.00	vpcmpled %zmm12, %zmm1, %k1
1	1	0.33	vmovdqa64 %zmm1, %zmm5
1	1	0.33	vmovdqa32 %zmm6, %zmm2
2	8	0.50	vmovdqu32 (%r12), %zmm0 {%k1} {z}
1	1	1.00	kmovw %k1, %k2
1	1	0.33	vpsubd %zmm1, %zmm0, %zmm4 {%k1} {z}
1	1	0.33	vmovdqa32 %zmm6, %zmm1
5	**25**	**8.00**	**vpgatherdd (%r14,%zmm4), %zmm1 {%k2}**
1	1	1.00	kmovw %k1, %k3
5	**25**	**8.00**	**vpgatherdd (%r15,%zmm0), %zmm2 {%k3}**
1	1	1.00	vpshufb %zmm7, %zmm1, %zmm1
1	1	1.00	vpshufb %zmm7, %zmm2, %zmm2
1	1	0.50	vpxord %zmm1, %zmm2, %zmm8 {%k1} {z}
1	5	0.50	vplzcntd %zmm8, %zmm2 {%k1} {z}
1	1	0.33	vpaddd %zmm3, %zmm4, %zmm8 {%k1} {z}
1	1	1.00	vpsrld $3, %zmm2, %zmm1 {%k1} {z}
1	1	0.33	vmovdqa32 %zmm0, %zmm2
1	1	0.33	vpaddd %zmm1, %zmm0, %zmm2 {%k1}
1	1	0.33	vpaddd %zmm3, %zmm0, %zmm4 {%k1} {z}
1	4	1.00	vpcmpled %zmm10, %zmm8, %k1 {%k1}
2	1	1.00	vmovdqu64 %zmm2, (%r12)
1	4	1.00	vpcmpled %zmm11, %zmm4, %k1 {%k1}
1	4	1.00	vpcmpeqd %zmm3, %zmm1, %k0 {%k1}
1	3	1.00	kmovw %k0, %eax

Appendix B: Latency of AVX2 instructions for the primary part of vectorization

#uOps	Latency	CPI	Instructions
1	7	0.50	vlddqu (%r12), %ymm2
1	1	0.33	vpaddd %ymm4, %ymm9, %ymm0
1	1	0.33	vpsubd %ymm0, %ymm2, %ymm1
1	1	0.33	vmovdqa %ymm3, %ymm10
1	1	0.33	vmovdqa %ymm3, %ymm11
1	1	0.33	vmovdqa %ymm0, %ymm4
5	**22**	**4.00**	vpgatherdd %ymm10, (%r14,%ymm1), %ymm0
5	**22**	**4.00**	vpgatherdd %ymm11, (%r15,%ymm2), %ymm1
1	1	1.00	vpshufb %ymm5, %ymm0, %ymm0
1	1	1.00	vpshufb %ymm5, %ymm1, %ymm1
1	1	0.33	vpxor %ymm1, %ymm0, %ymm1
1	1	0.50	vpsrld $8, %ymm1, %ymm0
1	1	0.33	vpandn %ymm1, %ymm0, %ymm0
1	4	0.50	vcvtdq2ps %ymm0, %ymm0
1	4	0.50	vpsrld $23, %ymm0, %ymm0
1	4	0.50	vpsubusw %ymm0, %ymm8, %ymm0
1	4	0.50	vpminsw %ymm7, %ymm0, %ymm0
1	4	0.50	vpsrld $3, %ymm0, %ymm0
1	4	1.00	vpcmpeqd %ymm6, %ymm0, %k1
1	1	0.33	vpaddd %ymm2, %ymm0, %ymm2
2	1	1.00	vmovdqu %ymm2, (%r12)
1	1	0.33	vmovdqa32 %ymm3, %ymm0 {%k1} {z}
1	2	1.00	vpmovmskb %ymm0, %edx

References

[1] Aguado-Puig, Q., Marco-Sola, S., Moure, J. C., Castells-Rufas, D., Alvarez, L., Espinosa, A., and Moreto, M. (2022a). Accelerating edit-distance sequence alignment on gpu using the wavefront algorithm. *IEEE access*, **10**, 63782–63796. 15

[2] Aguado-Puig, Q., Marco-Sola, S., Moure, J. C., Matzoros, C., Castells-Rufas, D., Espinosa, A., and Moreto, M. (2022b). Wfa-gpu: Gap-affine pairwise alignment using gpus. *bioRxiv*. 15, 72

[3] Ahmed, N., Bertels, K., and Al-Ars, Z. (2016). A comparison of seed-and-extend techniques in modern dna read alignment algorithms. In *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 1421–1428. 14

[4] Alser, M., Bingol, Z., Cali, D., Kim, J., Ghose, S., Alkan, C., and Mutlu, O. (2020). Accelerating genome analysis: A primer on an ongoing journey. *IEEE Micro*, **40**(05), 65–75. 2

[5] Alser, M., Rotman, J., Deshpande, D., Taraszka, K., Shi, H., Baykal, P. I., Yang, H. T., Xue, V., Knyazev, S., Singer, B. D., Balliu, B., Koslicki, D., Skums, P., Zelikovsky, A., Alkan, C., Mutlu, O., and Mangul, S. (2021). Technology dictates algorithms: recent developments in read alignment. *Genome Biology*, **22**(1), 249. 12, 14

[6] Alser, M., Lindegger, J., Firtina, C., Almadhoun, N., Mao, H., Singh, G., Gomez-Luna, J., and Mutlu, O. (2022). From molecules to genomic variations: Accelerating genome analysis via intelligent algorithms and architec-

tures. *Computational and Structural Biotechnology Journal*, **20**, 4579–4599. x, 1, 2, 8, 9, 10, 15

[7] Behjati, S. and Tarpey, P. S. (2013). What is next generation sequencing? *Archives of Disease in Childhood - Education and Practice*, **98**(6), 236–238. 7

[8] Bohannan, Z. S. and Mitrofanova, A. (2019). Calling variants in the clinic: Informed variant calling decisions based on biological, clinical, and laboratory variables. *Computational and Structural Biotechnology Journal*, **17**, 561–569. 11

[9] Britannica, T. Editors of Encyclopaedia (2022). Genetic code. https://www.britannica.com/science/genetic-code. 5

[10] BSC (2021). Marenostrum 4 node specs. https://www.bsc.es/support/MareNostrum4-ug.pdf. 21

[11] BSC (2022). MN4 Technical Information. https://www.bsc.es/marenostrum/marenostrum/technical-information. viii, 21

[12] Castells-Rufas, D., Marco-Sola, S., Aguado-Puig, Q., Espinosa-Morales, A., Moure, J. C., Alvarez, L., and Moretó, M. (2021). Opencl-based fpga accelerator for semi-global approximate string matching using diagonal bit-vectors. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 174–178. IEEE. 14

[13] Castells-Rufas, D., Marco-Sola, S., Moure, J. C., Aguado, Q., and Espinosa, A. (2022). Fpga acceleration of pre-alignment filters for short read mapping with hls. *IEEE Access*, **10**, 22079–22100. 14

[14] CDC (2022). Genomic sequencing. https://www.cdc.gov/coronavirus/2019-ncov/variants/genomic-surveillance.html. 1, 7

[15] Chacón, A., Marco-Sola, S., Espinosa, A., Ribeca, P., and Moure, J. C. (2014). Thread-cooperative, bit-parallel computation of levenshtein distance on gpu. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 103–112. 14

[16] Churko, J. M., Mantalas, G. L., Snyder, M. P., and Wu, J. C. (2013). Overview of high throughput sequencing technologies to elucidate molecular pathways in cardiovascular diseases. *Circ Res*, **112**(12), 1613–1623. 2

[17] Cingolani, P., Platts, A., Wang, L. L., Coon, M., Nguyen, T., Wang, L., Land, S. J., Lu, X., and Ruden, D. M. (2012). A program for annotating and predicting the effects of single nucleotide polymorphisms, SnpEff: SNPs in the genome of drosophila melanogaster strain w1118; iso-2; iso-3. *Fly (Austin)*, **6**(2), 80–92. 10

[18] CSU (2022). Practical Computing and Bioinformatics for Conservation and Evolutionary Genomics. https://eriqande.github.io/eca-bioinf-handbook/. Accessed: 2022-09-08. 2

[19] Daily, J. (2016). Parasail: Simd c library for global, semi-global, and local pairwise sequence alignments. *BMC bioinformatics*, **17**(1), 1–11. 15

[20] Davis, U. (2022). Illumina High Throughput Sequencing. https://dnatech.genomecenter.ucdavis.edu/illumina-high-throughput-sequencing/. Accessed: 2022-09-06. 2

[21] Davis-Turak, J., Courtney, S. M., Hazard, E. S., Glen, Jr, W. B., da Silveira, W. A., Wesselman, T., Harbin, L. P., Wolf, B. J., Chung, D., and Hardiman, G. (2017). Genomics pipelines and data integration: challenges and opportunities in the research setting. *Expert Rev Mol Diagn*, **17**(3), 225–237. 9

[22] Eric Green (2021). Cost of Sequencing. https://www.humanprogress.org/the-fastest-learning-curve-in-history/. 8

[23] Fakruddin, M. and Chowdhury, A. (2012). Pyrosequencing-an alternative to traditional sanger sequencing. *American Journal of Biochemistry and Biotechnology*, **8**(1), 14–20. 7

[24] Farrar, M. (2006). Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, **23**(2), 156–161. 16, 19, 38

[25] Flicek, P. and Birney, E. (2009). Sense from sequence reads: methods for alignment and assembly. *Nat Methods*, **6**(11 Suppl), S6–S12. 7

[26] Frellsen, J., Menzel, P., and Krogh, A. (2014). 6.03 - algorithms for mapping high-throughput dna sequences**jes frellsen and peter menzel contributed equally. In A. Brahme, editor, *Comprehensive Biomedical Physics*, pages 41–50. Elsevier, Oxford. 2

[27] GATK (2022). HaplotypeCaller. `https://gatk.broadinstitute.org/hc/en-us/articles/360037225632-HaplotypeCaller`. 11

[28] Gotoh, O. (1982). An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, **162**(3), 705–708. 16, 37

[29] Hagemann, I. S. (2015). Chapter 1 - overview of technical aspects and chemistries of next-generation sequencing. In S. Kulkarni and J. Pfeifer, editors, *Clinical Genomics*, pages 3–19. Academic Press, Boston. 7

[30] Haghi, A., Marco-Sola, S., Alvarez, L., Diamantopoulos, D., Hagleitner, C., and Moreto, M. (2021). An fpga accelerator of the wavefront algorithm for genomics pairwise alignment. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 151–159. IEEE. 15

[31] Holtgrewe, M. (2010). Mason : A read simulator for second generation sequencing data. *Technical Report FU Berlin*. 22

[32] Illumina.com (2022). Illumina sequencing. `https://www.illumina.com/science/technology/next-generation-sequencing.html`. Accessed: 2022-09-04. 2, 8

[33] Institute, N. H. G. R. (2013). First human genome draft. `https://bit.ly/3T8MU2D`. 7

[34] Intel (2022a). AVX-512 Overview. `https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html`. 3

[35] Intel (2022b). AVX2 Overview. `intel.ly/3V16LCj`. 3

[36] Intel (2022). Intel® VTune™ Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. 23

[37] Karp, R. M. and Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, **31**(2), 249–260. 16

[38] Kim, J. S., Senol Cali, D., Xin, H., Lee, D., Ghose, S., Alser, M., Hassan, H., Ergin, O., Alkan, C., and Mutlu, O. (2018). Grim-filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies. *BMC Genomics*, **19**(2), 89. 14

[39] Koboldt, D. C. (2020). Best practices for variant calling in clinical sequencing. *Genome Medicine*, **12**(1), 91. 10

[40] Lander, E. S., Linton, L. M., Birren, B., *et al.* (2001). Initial sequencing and analysis of the human genome. *Nature*, **409**(6822), 860–921. 1

[41] Langmead, B. and Salzberg, S. L. (2012). Fast gapped-read alignment with bowtie 2. *Nature Methods*, **9**(4), 357–359. 3, 19, 20, 22

[42] Li, H. (2013). Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. *arXiv preprint arXiv:1303.3997*. 18

[43] Li, H. (2018). Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, **34**(18), 3094–3100. 3, 10, 16, 20, 22

[44] M. Holtgrewe (2010). Mason2 Github. https://github.com/seqan/seqan/tree/master/apps/mason2. 22

[45] Marco-Sola, S., Sammeth, M., Guigó, R., and Ribeca, P. (2012). The gem mapper: fast, accurate and versatile alignment by filtration. *Nature Methods*, **9**(12), 1185–1188. 10

[46] Marco-Sola, S., Moure, J. C., Moreto, M., and Espinosa, A. (2020). Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics*, **37**(4), 456–463. 3

[47] Marco-Sola, S., Moure, J. C., Moreto, M., and Espinosa, A. (2021). Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics*, **37**(4), 456–463. 15, 38, 71

[48] Marco-Sola, S., Eizenga, J. M., Guarracino, A., Paten, B., Garrison, E., and Moreto, M. (2022). Optimal gap-affine alignment in o(s) space. *bioRxiv*. 38

[49] Nagwa (2022). DNA/Nucleotides Figure. https://www.nagwa.com/en/explainers/328197350602/. viii, 6

[50] nature.com (2022). "What Is a Cell?". https://www.nature.com/scitable/topicpage/what-is-a-cell-14023083/. 5

[51] NCBI (2019). FASTQ file format. https://www.ncbi.nlm.nih.gov/sra/docs/submitformats. 10

[52] Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, **48**(3), 443–453. 16, 37

[53] NHGRI (2022a). Chromosome Def. https://www.genome.gov/genetics-glossary/Chromosome. 6

[54] NHGRI (2022b). DNA Def. https://www.genome.gov/genetics-glossary/Deoxyribonucleic-Acid. 6

[55] NHGRI (2022c). Gene Def. https://www.genome.gov/genetics-glossary/Gene. 6

[56] Ono, Y., Asai, K., and Hamada, M. (2020). PBSIM2: a simulator for long-read sequencers with a novel generative model of quality scores. *Bioinformatics*, **37**(5), 589–595. 22

[57] Ono, Y., Asai, K., and Hamada, M. (2022). pbsim2 Github. https://learn.gencore.bio.nyu.edu/variant-calling/variant-discovery/. 22

[58] Oxford Nanopore Technologies (2022). ONT Sequencing. https://nanoporetech.com/. 8

[59] Pacific Biosciences (2022). PacBio Sequencing. https://www.pacb.com/. 8

[60] Pirooznia, M., Kramer, M., Parla, J., Goes, F. S., Potash, J. B., McCombie, W. R., and Zandi, P. P. (2014). Validation and assessment of variant calling pipelines for next-generation sequencing. *Human Genomics*, **8**(1), 14. 9, 10, 11

[61] Ranganathan, S., Gribskov, M., Nakai, K., and Schönbach, C., editors (2019). *Encyclopedia of Bioinformatics and Computational Biology - Volume 2*. Elsevier. 1

[62] Reuter, J. A., Spacek, D. V., and Snyder, M. P. (2015). High-throughput sequencing technologies. *Mol Cell*, **58**(4), 586–597. 2

[63] Roy, S., Coldren, C., Karunamurthy, A., Kip, N. S., Klee, E. W., Lincoln, S. E., Leon, A., Pullambhatla, M., Temple-Smolkin, R. L., Voelkerding, K. V., Wang, C., and Carter, A. B. (2018). Standards and guidelines for validating next-generation sequencing bioinformatics pipelines: A joint recommendation of the association for molecular pathology and the college of american pathologists. *The Journal of Molecular Diagnostics*, **20**(1), 4–27. 1, 2

[64] SADASIVAN, H., Maric, M., Dawson, E., Iyer, V., Israeli, J., and Narayanasamy, S. (2022). Accelerating minimap2 for accurate long read alignment on gpus. 31, 35

[65] Sanger, F., Nicklen, S., and Coulson, A. R. (1977). DNA sequencing with chain-terminating inhibitors. *Proc Natl Acad Sci U S A*, **74**(12), 5463–5467. 7

[66] Schatz, M. (2010). High performance computing for dna sequence alignment and assembly. 3

[67] Schloss, J. A. (2008). How to get genomes at one ten-thousandth the cost. *Nature Biotechnology*, **26**(10), 1113–1115. 2

[68] Shibuya, T. (2006). Geometric suffix tree: A new index structure for protein 3-d structures. In M. Lewenstein and G. Valiente, editors, *Combinatorial Pattern Matching*, pages 84–93, Berlin, Heidelberg. Springer Berlin Heidelberg. 13

[69] Smith, T. and Waterman, M. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, **147**(1), 195–197. 16, 17

[70] Song, B., Marco-Sola, S., Moreto, M., Johnson, L., Buckler, E. S., and Stitzer, M. C. (2022). Anchorwave: Sensitive alignment of genomes with high sequence diversity, extensive structural polymorphism, and whole-genome duplication. *Proceedings of the National Academy of Sciences*, **119**(1), e2113075119. 10

[71] Suzuki, H. and Kasahara, M. (2018a). Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC Bioinformatics*, **19**(Suppl 1), 45. 17

[72] Suzuki, H. and Kasahara, M. (2018b). Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC bioinformatics*, **19**(1), 33–47. 38

[73] Tikiri, C. (2013). Fast and accurate mapping of next generation sequencing data. 5

[74] Ukkonen, E. (2012). How to reconstruct a genome. In B. Rovan, V. Sassone, and P. Widmayer, editors, *Mathematical Foundations of Computer Science 2012*, pages 48–48, Berlin, Heidelberg. Springer Berlin Heidelberg. 3

[75] Univeristy, A. S. (2022). "Are viruses alive?". `https://askabiologist.asu.edu/questions/are-viruses-alive`. 5

[76] Vasimuddin, M., Misra, S., Li, H., and Aluru, S. (2019). Efficient architecture-aware acceleration of bwa-mem for multicore systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 314–324. 3, 10, 18, 20, 22

[77] Xin, H., Nahar, S., Zhu, R., Emmons, J., Pekhimenko, G., Kingsford, C., Alkan, C., and Mutlu, O. (2015). Optimal seed solver: optimizing seed selection in read mapping. *Bioinformatics*, **32**(11), 1632–1642. 13