ACCELERATING THE UNDERSTANDING OF LIFE'S CODE THROUGH BETTER ALGORITHMS AND HARDWARE DESIGN

A DISSERTATION SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

By
Mohammed H. K. Alser
June 2018

ACCELERATING THE UNDERSTANDING OF LIFE'S CODE THROUGH BETTER ALGORITHMS AND HARDWARE DESIGN

By Mohammed H. K. Alser

June 2018

We certify that we have read this dissertation and that in our opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Can Alkan (Advisor)

Onur Mutlu (Co-Advisor)

Ergin Atalar

Mehmet Koyutürk

Nurcan Tunçbağ

Oğuz Ergin

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

ACCELERATING THE UNDERSTANDING OF LIFE'S CODE THROUGH BETTER ALGORITHMS AND HARDWARE DESIGN

Mohammed H. K. Alser
Ph.D. in Computer Engineering
Advisor: Can Alkan
Co-Advisor: Onur Mutlu
June 2018

Our understanding of human genomes today is affected by the ability of modern computing technology to quickly and accurately determine an individual's entire genome. Over the past decade, high throughput sequencing (HTS) technologies have opened the door to remarkable biomedical discoveries through its ability to generate hundreds of millions to billions of DNA segments per run along with a substantial reduction in time and cost. However, this flood of sequencing data continues to overwhelm the processing capacity of existing algorithms and hardware. To analyze a patient's genome, each of these segments - called *reads* - must be mapped to a reference genome based on the similarity between a read and "candidate" locations in that reference genome. The similarity measurement, called alignment, formulated as an approximate string matching problem, is the computational bottleneck because: (1) it is implemented using quadratic-time dynamic programming algorithms, and (2) the majority of candidate locations in the reference genome do not align with a given read due to high dissimilarity. Calculating the alignment of such incorrect candidate locations consumes an overwhelming majority of a modern read mapper's execution time. Therefore, it is crucial to develop a fast and effective filter that can detect incorrect candidate locations and eliminate them before invoking computationally costly alignment algorithms.

In this thesis, we introduce four new algorithms that function as a pre-alignment step and aim to filter out most incorrect candidate locations. We call our algorithms GateKeeper, Slider, MAGNET, and SneakySnake. The first key idea of our proposed pre-alignment filters is to provide high filtering accuracy by correctly detecting all similar segments shared between two sequences.

The second key idea is to exploit the massively parallel architecture of modern FPGAs for accelerating our four proposed filtering algorithms. We also develop an efficient CPU implementation of the SneakySnake algorithm for commodity desktops and servers, which are largely available to bioinformaticians without the hassle of handling hardware complexity. We evaluate the benefits and downsides of our pre-alignment filtering approach in detail using 12 real datasets across different read length and edit distance thresholds. In our evaluation, we demonstrate that our hardware pre-alignment filters show two to three orders of magnitude speedup over their equivalent CPU implementations. We also demonstrate that integrating our hardware pre-alignment filters with the state-of-the-art read aligners reduces the aligner's execution time by up to 21.5x. Finally, we show that efficient CPU implementation of pre-alignment filtering still provides significant benefits. We show that SneakySnake on average reduces the execution time of the best performing CPU-based read aligners Edlib and Parasail, by up to 43x and 57.9x, respectively. The key conclusion of this thesis is that developing a fast and efficient filtering heuristic, and developing a better understanding of its accuracy together leads to significant reduction in read alignment's execution time, without sacrificing any of the aligner' capabilities. We hope and believe that our new architectures and algorithms catalyze their adoption in existing and future genome analysis pipelines.

Keywords: Read Mapping, Approximate String Matching, Read Alignment, Levenshtein Distance, String Algorithms, Edit Distance, fast pre-alignment filter, field-programmable gate arrays (FPGA).

ÖZET

YAŞAMIN KODUNU ANLAMAYI DAHA İYİ ALGORİTMALAR VE DONANIM TASARIMLARIYLA HIZLANDIRMAK

Mohammed H. K. Alser
Bilgisayar Mühendisliği, Doktora
Tez Danışmanı: Can Alkan
İkinci Tez Danışmanı: Onur Mutlu
Haziran 2018

Günümüzde insan genomları konusundaki anlayışımız, modern bilişim teknolojisinin bir bireyin tüm genomunu hızlı ve doğru bir şekilde belirleyebilme yeteneğinden etkilenmektedir. Geçtiğimiz on yıl boyunca, yüksek verimli dizileme (HTS) teknolojileri, zaman ve maliyette önemli bir azalma ile birlikte, tek bir çalışmada yüz milyonlardan milyarlarcaya kadar DNA parçası üretme kabiliyeti sayesinde dikkat çekici biyomedikal keşiflere kapı açmıştır. Ancak, bu dizileme verisi bolluğu mevcut algoritmaların ve donanımların işlem kapasitelerinin sınırlarını zorlamaya devam etmektedir. Bir hastanın genomunu analiz etmek için, "okuma" adı verilen bu parçaların her biri referans genomundaki aday bölgelerle olan benzerliklerine bakılarak, referans genomu üzerine yerleştirilir. Yaklaşık karakter dizgisi eşleştirme problemi şeklinde formüle edilen ve hizalama olarak adlandırılan benzerlik hesaplaması, işlemsel bir darboğazdır çünkü: (1) ikinci dereceden devingen programlama algoritmaları kullanılarak hesaplanır ve (2) referans genomundaki aday bölgelerin büyük bir bölümü ile verilen okuma parçası birbirlerinden yüksek düzeyde farklılık gösterdiklerinden dolayı hizalanamaz. Bu şekilde yanlış belirlenen aday bölgelerin hizalanabilirliğin hesaplanması, günümüzdeki okuma haritalandırıcı algoritmaların çalışma sürelerinin büyük bölümünü oluşturmaktadır. Bu nedenle, hesaplama olarak maliyetli bu hizalama algoritmalarını çalıştırmadan önce, doğru olmayan aday bölgeleri tespit edebilen ve bu bölgeleri aday bölge olmaktan çıkaran, hızlı ve etkili bir filtre geliştirmek çok önemlidir.

Bu tezde, ön hizalama aşaması olarak işlev gören ve yanlış aday konumlarının çoğunu filtrelemeyi hedefleyen dört yeni algoritma sunuyoruz. Algoritmalarımızı GateKeeper, SLIDER, MAGNET ve SneakySnake olarak adlandırıyoruz.

Önerilen ön hizalama filtrelerinin ilk temel fikri, iki dizi arasında paylaşılan tüm benzer segmentleri doğru bir şekilde tespit ederek yüksek filtreleme doğruluğu sağlamaktır. İkinci temel fikir, önerilen dört filtreleme algoritmamızın hızlandırılması için modern FPGA'ların çok büyük ölçekte paralel mimarisini kullanmaktır. SneakySnake'i esas olarak biyoinformatisyenlerin mevcut olan, donanım karmaşıklığı ile uğraşmak zorunda olmadıkları emtia masaüstü ve sunucularında kullanabilmeleri için geliştirdik. Ön okuma filtreleme yaklaşımımızın avantaj ve dezavantajlarını 12 gerçek veri setini, farklı okuma uzunlukları ve mesafe eşikleri kullanarak ayrıntılı olarak değerlendirdik. Değerlendirmemizde, donanım ön hizalama filtrelerimizin eşdeğer CPU uygulamalarına göre iki ila üç derece hızlı olduklarını gösteriyoruz. Donanım ön hizalama filtrelerimizi son teknoloji okuma hizalayıcılarıyla entegre etmenin hizalayıcının çalışma süresini düzenleme mesafesi eşiğine bağlı olarak 21.5x. Son olarak, ön hizalama filtrelerinin etkin CPU uygulamasının hala önemli faydalar sağladığını gösteriyoruz. SneakySnake'in en iyi performansa sahip CPU tabanlı okuma ayarlayıcıları Edlib ve Parasail'in yürütme sürelerini sırasıyla 43x ve 57,9x'e kadar azalttığını gösteriyoruz. Bu tezin ana sonucu, hızlı ve verimli bir filtreleme mekanizması geliştirilmesi ve bu mekanizmanın doğruluğunun daha iyi anlaşılması, hizalayıcıların yeteneklerinden hiçbir şey ödün vermeden, okuma hizalamasının çalışma süresinde önemli bir azalmaya yol açmaktadır. Yeni mimarilerimizin ve algoritmalarımızın, mevcut ve gelecekteki genom analiz planlarında benimsenmelerini katalize ettiğimizi umuyor ve buna inanıyoruz.

Anahtar sözcükler: Haritalamayı Oku, Yaklaşık Dize Eşleştirme, Hizalamayı Oku, Levenshtein Mesafesi, String Algoritmaları, Mesafeyi Düzenle, hızlı ön hizalama filtresi, Alan programlanabilir kapı dizileri (FPGA).

Acknowledgement

The last nearly four years at Bilkent University have been most exciting and fruitful time of my life, not only in the academic arena, but also on a personal level. For that, I would like to seize this opportunity to acknowledge all people who have supported and helped me to become who I am today. First and foremost, the greatest of all my appreciation goes to the almighty Allah for granting me countless blessings and helping me stay strong and focused to accomplish this work.

I would like to extend my sincere thanks to my advisor Prof. Can Alkan, who introduced me to the realm of bioinformatics. I feel very privileged to be part of his lab and I am proud that I am going to be his first PhD graduate. Can generously provided the resources and the opportunities that enabled me to publish and collaborate with researchers from other institutions. I appreciate all his support and guidance at key moments in my studies while allowing me to work independently with my ideas. I truly enjoyed the memorable time we spent together in Ankara, Izmir, Istanbul, Vienna, and Los Angeles. Thanks to him, I get addicted to the dark chocolate with almonds and sea salt.

I am also very thankful to my co-advisor Prof. Onur Mutlu for pushing my academic writing to a new level. It is an honor that he also gave me the opportunity to join his lab at ETH Zürich as a postdoctoral researcher. His dedication to research is contagious and inspirational.

I am grateful to the members of my thesis committee: Prof. Ergin Atalar, Prof. Mehmet Koyutürk, Prof. Nurcan Tunçbağ, Prof. Oğuz Ergin, and Prof. Ozcan Oztürk for their valuable comments and discussions.

I would also like to thank Prof. Eleazar Eskin and Prof. Jason Cong for giving me the opportunity to join UCLA during the summer of 2017 as a staff research associate. I would like to extend my thanks to Serghei Mangul, David Koslicki, Farhad Hormozdiari, and Nathan LaPierre who provided the guidance to make my visit fruitful and enjoyable. I am also thankful to Prof. Akash Kumar for giving me the chance to join the Chair for Processor Design during the winter of 2017-2018 as a visiting researcher at TU Dresden.

I would like to acknowledge Dr. Nordin Zakaria and Dr. Izzatdin B A Aziz for providing me the opportunity to continue carrying out PETRONAS projects while i am in Turkey. I am grateful to Prof. Nor Hisham Hamid for believing in me, hiring me as a consultant for one of his government-sponsored projects, and inviting me to Malaysia. It was my great fortune to work with them.

I want to thank all my co-authors and colleagues: Eleazar Eskin, Erman Ayday, Hongyi Xin, Hasan Hassan, Serghei Mangul, Oğuz Ergin, Akash Kumar, Nour Almadhoun, Jeremie Kim, and Azita Nouri.

I would like to acknowledge all past and current members of our research group for being both great friends and colleagues to me. Special thanks to Fatma Kahveci for being a great labmate who taught me her native language and tolerated me for many years. Thanks to Gülfem Demir for all her valuable support and encouragement. Thanks to Shatlyk Ashyralyyev for his great help when I prepared for my qualifying examination and for many valuable discussions on research and life. Thanks to Handan Kulan for her friendly nature and priceless support. Thanks to Can Firtina and Damla Şenol Cali (CMU) for all the help in translating the abstract of this thesis to Turkish. I am grateful to other members for their companionship: Marzieh Eslami Rasekh, Azita Nouri, Elif Dal, Zülal Bingöl, Ezgi Ebren, Arda Söylev, Halil İbrahim Özercan, Fatih Karaoğlanoğlu, Tuğba Doğan, and Balanur İçen.

I would like to express my deepest gratitude to Ahmed Nemer Almadhoun, Khaldoun Elbatsh (and his wonderful family), Hamzeh Ahangari, Soe Thane, Mohamed Meselhy Eltoukhy (and his family), Abdel-Haleem Abdel-Aty, Heba Kadry, and Ibrahima Faye for all the good times and endless generous support in the ups and downs of life. I feel very grateful to all my inspiring friends who made Bilkent a happy home for me, especially Tamer Alloh, Basil Heriz, Kadir Akbudak, Amirah Ahmed, Zahra Ghanem, Muaz Draz, Nabeel Abu Baker, Maha Sharei, Salman Dar, Elif Doğan Dar, Ahmed Ouf, Shady Zahed, Mohammed Tareq, Abdelrahman Teskyeh, Obada and many others.

My graduate study is an enjoyable journey with my lovely wife, Nour Almadhoun, and my sons, Hasan and Adam. Their love and never ending support always give me the internal strength to move on whenever I feel like giving up or whenever things become too hard.

Pursuing PhD together in the same department is our most precious memory in lifetime. We could not have been completed any of our PhD work without continuously supporting and encouraging each other.

My family has been a pillar of support, encouragement and comfort all throughout my journey. Thanks to my parents, Hasan Alser and Itaf Abdelhadi, who raised me with a love of science and supported me in all my pursuits. Thanks to my brothers Ayman (and his family), Hani, Sohaib, Osaid, Moath, and Ayham and my lovely sister Deema for always believing in me and being by my side. Thanks to my parents-in-law, Mohammed Almadhoun and Nariman Almadhoun, my brotherss-in-law, Ahmed Nemer, Ezz Eldeen, Moatasem, Mahmoud, sisters-in-law, Narmeen, Rania, and Eman, and their families for all their understanding, great support, and love.

Finally, I gratefully acknowledge the funding sources that made my PhD work possible. I was honored to be a TÜBİTAK Fellow for 4 years offered by the Scientific and Technological Research Council of Turkey under 2215 program. In 2017, I was also honored to receive the HiPEAC Collaboration Grant. I am grateful to Bilkent University for providing generous financial support and funding my academic visits. This thesis was supported by NIH Grant (No. HG006004) to Professors Onur Mutlu and Can Alkan and a Marie Curie Career Integration Grant (No. PCIG-2011-303772) to Can Alkan under the Seventh Framework Programme.

Mohammed H. K. Alser

June 2018, Ankara, Turkey

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Genome is the *code of life* that includes set of instructions for making everything from humans to elephants, bananas, and yeast. Analyzing the life's code helps, for example, to determine differences in genomes from human to human that are passed from one generation to the next and may cause diseases or different traits [10, 11, 12, 13, 14, 15]. One benefit of knowing the genetic variations is better understanding and diagnosis diseases such as cancer and autism [16, 17, 18, 19] and the development of efficient drugs [20, 21, 22, 23, 24]. The first step in genome analysis is to reveal the entire content of the subject genome – a process known as DNA sequencing [25]. Until today, it remains challenging to sequence the entire DNA molecule as a whole. As a workaround, high throughput DNA sequencing (HTS) technologies are used to sequence random fragments of copies of the original molecule. These fragments are called short reads and are 75-300 basepairs (bp) long. The resulting reads lack information about their order and origin (i.e., which part of the subject genome they are originated from). Hence the main challenge in genome analysis is to construct the donor's complete genome with respect to a reference genome.

During a process, called read mapping, each read is mapped onto one or more possible locations in the reference genome based on the similarity between the read and the reference sequence segment at that location (like solving a jigsaw puzzle). The similarity measurement is referred to as optimal read alignment (i.e., verification) and could be calculated using the Smith-Waterman local alignment algorithm [26]. However, this approach is infeasible as it requires *O(mn)* running time, where m is the read length (few hundreds of bp) and n is the reference length ($\sim$ 3.2 billion bp for human genome), for each read in the data set (hundreds of millions to billions).

To accelerate the read mapping process and reduce the search space, state-of-the-art mappers employ a strategy called *seed-and-extend*. In this strategy, a mapper applies heuristics to first find candidate map locations (seed locations) of subsequences of the reads using hash tables (BitMapper [27], mrFAST with FastHASH [28], mrsFAST [29]) or BWT-FM indices (BWA-MEM [30], Bowtie 2 [31], SOAP3-dp [32]). It then aligns the read in full *only* to those seed locations. Although the strategies for finding seed locations vary among different read mapping algorithms, seed location identification is typically followed by alignment step. The general goal of this step is to compare the read to the reference segment at the seed location to check if the read aligns to that location in the genome with fewer differences (called *edits*) than a threshold [33].

1.1 Research Problem

The alignment step is the performance bottleneck of today's read mapper taking over 70% to 90% of the total running time [34, 27, 28]. We pinpoint three specific problems that cause, affect, or exacerbate the long alignment's execution time. (1) We find that across our data set (see **Chapter 9**), an overwhelming majority (more than 90% as we present in Figure 1.1) of the seed locations, that are generated by a state-of-the-art read mapper, mrFAST with FastHASH [28], exhibit more edits than the allowed threshold.

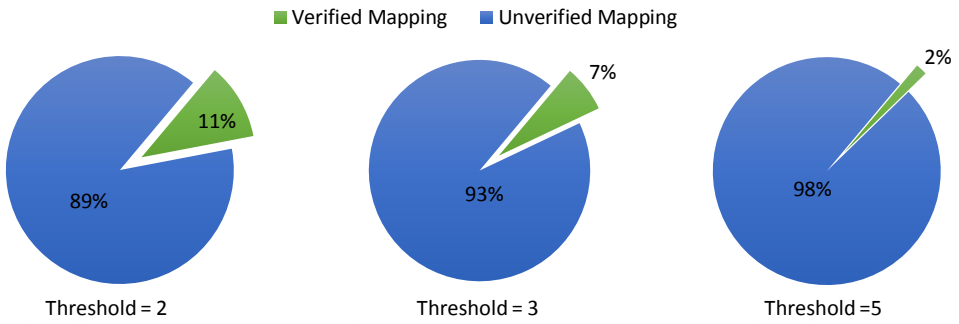

Figure 1.1: Rate of verified and unverified read-reference pairs that are generated by mrFAST mapper and are fed into its read alignment algorithm. We set the threshold to 2, 3, and 5 edits.

These particular seed locations impose a large computational burden as they waste 90% of the alignment's execution time in verifying these incorrect mappings. This observation is also in line with similar results for other read mappers [28, 34, 27, 35].

(2) Typical read alignment algorithm also needs to tolerate sequencing *errors* [36] as well as genetic variations [37]. Thus, read alignment step is implemented using dynamic programming algorithms such as Levenshtein distance [38], Smith-Waterman [26], Needleman-Wunsch [39] and their improved implementations. These algorithms are inefficient as they run in a quadratic-time complexity in the read length, m, (i.e., $O(m^2)$).

(3) This computational burden is further aggravated by the unprecedented flood of sequencing data which continues to overwhelm the processing capacity of existing algorithms and compute infrastructures [40]. While today's HTS machines (e.g., Illumina HiSeq4000) can generate more than 300 million bases per minute, state-of-the-art read mapper can only map 1% of these bases per minute [41]. The situation gets even worse when one tries to understand complex diseases such as autism and cancer, which require sequencing hundreds of thousands of genomes [17, 18, 42, 43, 44, 45, 46]. The long execution time of modern-day read alignment can severely hinder such studies.

There is also an urgent need for rapidly incorporating clinical sequencing into clinical practice for diagnosis of genetic disorders in critically ill infants [47, 48, 49, 50]. While early diagnosis in such infants shortens the clinical course and enables optimal outcomes [51, 52, 53], it is still challenging to deliver efficient clinical sequencing for tens to hundreds of thousands of hospitalized infants each year [54].

Tackling these challenges and bridging the widening gap between the execution time of read alignment and the increasing amount of sequencing data necessitate the development of fundamentally new, fast, and efficient read mapping algorithms. In the next sections, we provide the motivation behind our proposed work to considerably boost the performance of read alignment. We also provide further background information and literature study in Chapter 2.

1.2 Motivation

We present in Figure 1.2 the flow chart of a typical seed-and-extend based mapper during the mapping stage. The mapper follows five main steps to map a read set to the reference genome sequence. (1) In step 1, typically a mapper first constructs fast indexing data structure (e.g., large hash table) for short segments (called seeds or k-mers or q-maps) of the reference sequence. (2) In step 2, the mapper extracts short subsequences from a read and uses them to query the hash table. (3) In step 3, The hash table returns all the occurrence hits of each seed in the reference genome. Modern mappers employ seed filtering mechanism to reduce the false seed locations that leads to incorrect mappings. (4) In step 4, for each possible location in the list, the mapper retrieves the corresponding reference segment from the reference genome based on the seed location. The mapper can then examine the alignment of the entire read with the reference segment using fast filtering heuristics that reduce the need for the dynamic programming algorithms. It rejects the mappings if the read and the reference are obviously dissimilar.

Otherwise, the mapper proceeds to the next step. (5) In step 5, the mapper calculates the optimal alignment between the read sequence and the reference sequence using a computationally costly sequence alignment (i.e., verification) algorithm to determine the similarity between the read sequence and the reference sequence. Many attempts were made to tackle the computationally very expensive alignment problem. Most existing works tend to follow one of three key directions: (1) accelerating the dynamic programming algorithms [55, 56, 9, 2, 57, 58, 5, 32, 59, 60], (2) developing seed filters that aim to reduce the false seed locations [27, 28, 35, 61, 62, 63, 64], and (3) developing pre-alignment filtering heuristics [7].

The first direction takes advantage of parallelism capabilities of high performance computing platforms such as central processing units (CPUs) [9, 57], graphics processing units (GPUs) [58, 32, 59], and field-programmable gate arrays (FPGAs) [55, 56, 65, 66, 2, 5, 60]. Among these computing platforms, FPGA accelerators seem to yield the highest performance gain [56, 6]. However, many of these efforts either *simplify* the scoring function, or only take into account accelerating the computation of the dynamic programming matrix *without* providing the optimal alignment (i.e., backtracking) as in [65, 66]. Different scoring functions are typically needed to better quantify the similarity between the read and the reference sequence segment [67]. The backtracking step required for optimal alignment computation involves unpredictable and irregular memory access patterns, which poses difficult challenge for efficient hardware implementation.

The second direction to accelerate read alignment is to use filtering heuristics to reduce the size of the seed location list. This is the basic principle of nearly all mappers that employ seed-and-extend approach. Seed filter applies heuristics to reduce the output location list. The location list stores all the occurrence locations of each seed in the reference genome. The returned location list can be tremendously large as a mapper searches for an exact matches of short segment (typically 10 bp -13 bp for hash-based mappers) between two very long homologous genomes [68]. Filters in this category suffer from low filtering accuracy as they can only look for exact matches with the help of hash table.

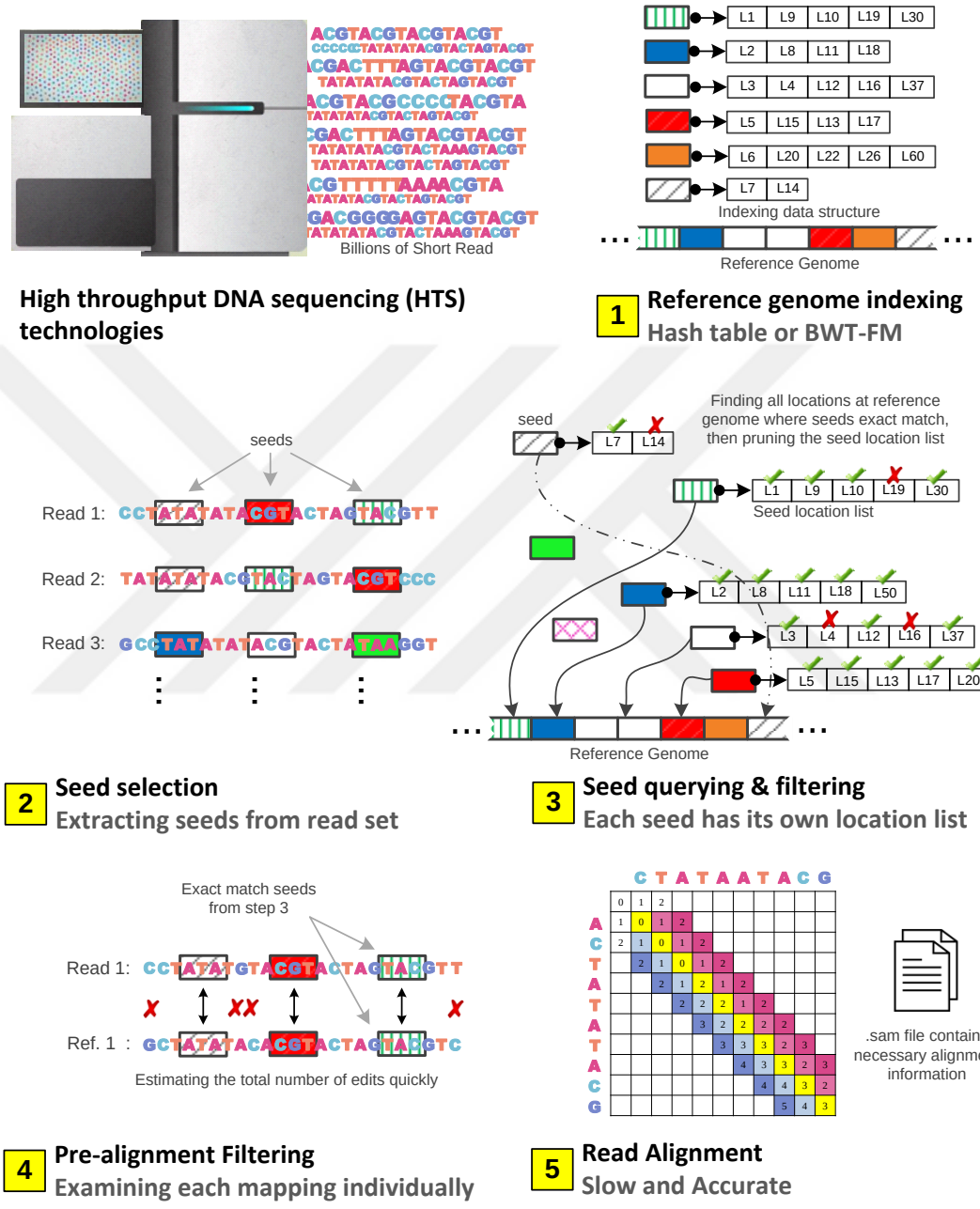

Figure 1.2: The flow chart of seed-and-extend based mappers that includes five main steps. (1) Typical mapper starts with indexing the reference genome using a scheme based on hash table or BWT-FM. (2) It extracts number of seeds from each read that are produced using sequencing machine. (3) It obtains a list of possible locations within the reference genome that could result in a match with the extracted seeds. (4) It applies fast heuristics to examine the alignment of the entire read with its corresponding reference segment of the same length. (5) It uses expensive and accurate alignment algorithm to determine the similarity between the read and the reference sequences.

Thus, they query a few number of seeds per read (e.g., in Bowtie 2 [31], it is 3 fixed length seeds at fixed locations) to maintain edit tolerance. mrFAST [28] uses another approach to increase the seed filtering accuracy by querying the seed and its shifted copies. This idea is based on the observation that indels cause the trailing characters to be shifted to one direction. If one of the shifted copies of the seed, generated from the read sequence, or the seed itself matches the corresponding seed from the reference, then this seed has zero edits. Otherwise, this approach calculates the number of edits in this seed as a single edit (that can be a single indel or a single substitution). Therefore, this approach fails to detect the correct number of edits for these case, for example, more than one substitutions in the same seed, substitutions and indel in the same seed, or more than one indels in the last seed). Seed filtering successes to eliminate some of the incorrect locations but it is still unable to eliminate sufficiently enough large portion of the false seed locations, as we present in Figure 1.1.

The third direction to accelerate read alignment is to minimize the number of incorrect mappings on which alignment is performed by incorporating filtering heuristics. This is the last line of defense before invoking computationally expensive read alignment. Such filters come into play before read alignment (i.e., hence called pre-alignment filter), discarding incorrect mappings that alignment would deem a poor match. Though the seed filtering and the pre-alignment filtering have the same goal, they are fundamentally different problems. In pre-alignment filtering approach, a filter needs to examine the entire mapping. They calculate a best guess estimate for the alignment score between a read sequence and a reference segment. If the lower bound exceeds a certain number of edits, indicating that the read and the reference segment do not align, the mapping is eliminated such that no alignment is performed. Unfortunately, the best performing existing pre-alignment filter, such as shifted Hamming distance (SHD), is slow and its mechanism introduces inaccuracy in its filtering unnecessarily as we show in our study in Chapter 3 and in our experimental evaluation, Chapter 9.

Pre-alignment filter enables the acceleration of read alignment and meanwhile offers the ability to make the best use of existing read alignment algorithms.

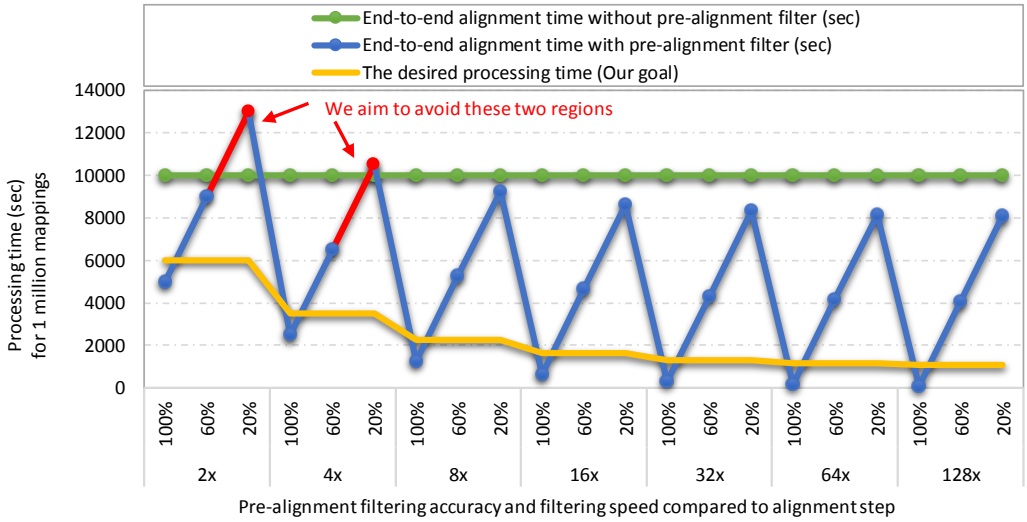

Figure 1.3: End-to-end execution time (in seconds) for read alignment, with (blue plot) and without (green plot) pre-alignment filter. Our goal is to significantly reduce the alignment time spent on verifying incorrect mappings (highlighted in yellow). We sweep the percentage of rejected mappings and the filtering speed compared to alignment algorithm in the x-axis.

These benefits come *without* sacrificing any capabilities of these algorithms, as pre-alignment filter does *not* modify or replace the alignment step. This motivates us to focus our improvement and acceleration efforts on pre-alignment filtering. We analyze in Figure 1.3 the effect of adding pre-alignment filtering step before calculating the optimal alignment and after generating the seed locations. We make two key observations. (1) The reduction in the end-to-end processing time of the alignment step largely depends on the accuracy and the speed of the pre-alignment filter. (2) Pre-alignment filtering can provide unsatisfactory performance (as highlighted in red) if it can not reject more than about 30% of the potential mappings while it's only 2x-4x faster than read alignment step.

We conclude that it is important to understand well what makes pre-alignment filter inefficient, such that we can devise new filtering technique that is much faster than read alignment and yet maintains high filtering accuracy.

1.3 Thesis Statement

Our goal in this thesis is to significantly reduce the time spent on calculating the optimal alignment in genome analysis from hours to mere seconds, given limited computational resources (i.e., personal computer or small hardware). This would make it feasible to analyze DNA routinely in the clinic for personalized health applications. Towards this end, we analyze the mappings that are provided to read alignment algorithm, and explore the causes of filtering inaccuracy. Our thesis statement is:

read alignment can be substantially accelerated using computationally inexpensive and accurate pre-alignment filtering algorithms designed for specialized hardware.

Accurate filter designed on a specialized hardware platform can drastically expedite read alignment by reducing the number of locations that must be verified via dynamic programming. To this end, we (1) develop four hardware-acceleration-friendly filtering algorithms and highly-parallel hardware accelerator designs which greatly reduce the need for alignment verification in DNA read mapping, (2) introduce fast and accurate pre-alignment filter for general purpose processors, and (3) develop a better understanding of filtering inaccuracy and explore speed/accuracy trade-offs.

1.4 Contributions

The overarching contribution of this thesis is the new algorithms and architectures that reduce read alignment's execution time in read mapping. More specifically, this thesis makes the following main contributions:

1. We provide a detailed investigation and analysis of four potential causes of filtering inaccuracy in the state-of-the-art alignment filter, SHD [7].

We also provide our recommendations on eliminating these causes and improving the overall filtering accuracy.

2. **GateKeeper.** We introduce the first hardware pre-alignment filtering, GateKeeper, which substantially reduces the need for alignment verification in DNA read mapping. GateKeeper is highly parallel and heavily relies on bitwise operations such as look-up table, shift, XOR, and AND. GateKeeper can examine up to 16 mappings in parallel, on a single FPGA chip with a logic utilization of less than 1% for a single filtering unit. It provides two orders of magnitude speedup over the state-of-the-art pre-alignment filter, SHD. It also provides up to 13.9x speedup to the state-of-the-art aligners. GateKeeper is published in Bioinformatics [69] and also available in arXiv [70].

3. **SLIDER.** We introduce SLIDER, a highly accurate and parallel pre-alignment filter which uses a sliding window approach to quickly identify dissimilar sequences without the need for computationally expensive alignment algorithms. SLIDER can examine up to 16 mappings in parallel, on a single FPGA chip with a logic utilization of up to 2% for a single filtering unit. It provides, on average, 1.2x to 1.4x more speedup than what GateKeeper provides to the state-of-the-art read aligners due to its high accuracy. SLIDER is 2.9x to 155x more accurate than GateKeeper. This work is yet to be published.

4. **MAGNET.** We introduce MAGNET, a highly accurate pre-alignment filter which employs greedy divide-and-conquer approach for identifying all non-overlapping long matches between two sequences. MAGNET can examine 2 or 8 mappings in parallel depending on the edit distance threshold, on a single FPGA chip with a logic utilization of up to 37.8% for a single filtering unit. MAGNET is, on average, two to four orders of magnitude more accurate than both SLIDER and GateKeeper. This comes at the expense of its filtering speed as it becomes up to 8x slower than SLIDER and GateKeeper. It still provides up to 16.6x speedup to the state-of-the-art read aligners. MAGNET is published in IPSI [71], available in arXiv [72], and presented in AACBB2018 [73].

5. **SneakySnake.** We introduce SneakySnake, the fastest and the most accurate pre-alignment filter. SneakySnake reduces the problem of finding the optimal alignment to finding a snake's optimal path (with the least number of obstacles) in linear time complexity in read length. We provide a cost-effective CPU implementation for our SneakySnake algorithm that accelerates the state-of-the-art read aligners, Edlib [8] and Parasail [9], by up to 43x and 57.9x, respectively, without the need for hardware accelerators. We also provide a scalable hardware architecture and hardware design optimization for the SneakySnake algorithm in order to further boost its speed. The hardware implementation of SneakySnake accelerates the existing state-of-the-art aligners by up to 21.5x when it is combined with the aligner. SneakySnake is up to one order, four orders, and five orders of magnitude more accurate compared to MAGNET, SLIDER, and GateKeeper, while preserving all correct mappings. SneakySnake also reduces the memory footprint of Edlib aligner by 50%. This work is yet to be published.

6. We provide a comprehensive analysis of the asymptotic run time and space complexity of our four pre-alignment filtering algorithms. We perform a detailed experimental evaluation of our proposed algorithms using 12 real datasets across three different read lengths (100 bp, 150 bp, and 250 bp) an edit distance threshold of 0% to 10% of the read length. We explore different implementations for the edit distance problem in order to compare the performance of SneakySnake that calculate approximate edit distance with that of the efficient implementation of exact edit distance. This also helps us to develop a deep understanding of the trade-off between the accuracy and speed of pre-alignment filtering.

Overall, we show in this thesis that developing a hardware-based alignment filtering algorithm and architecture together is both feasible and effective by building our hardware accelerator on a modern FPGA system. We also demonstrate that our pre-alignment filters are more effective in boosting the overall performance of the alignment step than only accelerating the dynamic programming algorithms by one to two orders of magnitude.

This thesis provides a foundation in developing fast and accurate pre-alignment filters for accelerating existing and future read mappers.

1.5 Outline

This thesis is organized into 11 chapters. Chapter 2 describes the necessary background on read mappers and related prior works on accelerating their computations. Chapter 3 explores the potential causes of filtering inaccuracy and provides recommendations on tackling them. Chapter 4 presents the architecture and implementation details of our hardware accelerator that we use for boosting the speed of our proposed pre-alignment filters. Chapter 5 presents GateKeeper algorithm and architecture. Chapter 6 presents SLIDER algorithm and its hardware architecture. Chapter 7 presents MAGNET algorithm and its hardware architecture. Chapter 8 presents SneakySnake algorithm. Chapter 9 presents the detailed experimental evaluation for all our proposed pre-alignment filters along with comprehensive comparison with the state-of-the-art existing works. Chapter 10 presents conclusions and future research directions that are enabled by this thesis. Finally, Appendix A extends Chapter 9 with more detailed information and additional experimental data/results.

Chapter 2

Background

In this chapter, we provide the necessary background on two key read mapping methods. We highlight the strengths and weaknesses of each method. We then provide an extensive literature review on the prior, existing, and recent approaches for accelerating the operations of read mappers. We devote the provided background materials only to the reduction of read mapper's execution time.

2.1 Overview of Read Mapper

With the presence of a reference genome, read mappers maintain a large index database ($\sim$ 3 GB to 20 GB for human genome) for the reference genome. This facilitates querying the whole reference sequence quickly and efficiently. Read mappers can use one of the following indexing techniques: suffix trees [74], suffix arrays [75], Burrows-Wheeler transformation [76] followed by Ferragina-Manzini index [77] (BWT-FM), and hash tables [28, 78, 62]. The choice of the index affects the query size, querying speed, and memory footprint of the read mapper, and even access patterns.

Unlike hash tables, suffix-array or suffix tree can answer queries of variable length sequences. Based on the indexing technique used, short read mappers typically fall into one of two main categories [40]: (1) Burrows-Wheeler Transformation [76] and Ferragina-Manzini index [77] (BWT-FM)-based methods and (2) Seed-and-extend based methods. Both types have different strengths and weaknesses. The first approach (implemented by BWA [79], BWT-SW [80], Bowtie [81], SOAP2 [82], and SOAP3 [83]) uses aggressive algorithms to optimize the candidate location pools to find closest matches, and therefore may not find many potentially-correct mappings [84]. Their performance degrades as either the sequencing error rate increases or the genetic differences between the subject and the reference genome are more likely to occur [85, 79]. To allow mismatches, BWT-FM mapper exhaustively traverses the data structure and match the seed to each possible path. Thus, Bowtie [81], for example, performs a depth-first search (DFS) algorithm on the prefix trie and stops when the first hit (within a threshold of less than 4) is found. Next, we explain SOAP2, SOAP3, and Bowtie as examples of this category.

SOAP2 [82] improves the execution time and the memory utilization of SOAP [86] by replacing its hash index technique with the BWT index. SOAP2 divides the read into non-overlapping (i.e., consecutive) seeds based on the number of allowed edits (default five). To tolerate two edits, SOAP2 splits a read into three consecutive seeds to search for at least one exact match seed that allows for up to two mismatches. SOAP3 [83] is the first read mapper that leverage graphics processing unit (GPU) to facilitate parallel calculations, as the authors claim in [83]. It speeds up the mapping process of SOAP2 [82] using a reference sequence that is indexed by the combination of the BWT index and the hash table. The purpose of this combination is to address the issue of random memory access while searching the BWT index, which is challenging for a GPU implementation. Both SOAP2 [82] and SOAP3 [83] can support alignment with an edit distance threshold of up to four bp.

Bowtie [81] follows the same concept of SOAP2. However, it also provides a backtracking step that favors high-quality alignments. It also uses a 'double BWT indexing' approach to avoid excessive backtracking by indexing the reference genome and its reversed version. Bowtie fails to align reads to a reference for an edit distance threshold of more than three bp.

The second category uses a hash table to index short seeds presented in either the read set (as in SHRiMP [78], Maq [87], RMAP [88], and ZOOM [89]) or the reference (as in most of the other modern mappers in this category). The idea of the hash table indexing can be tracked back to BLAST [90]. Examples of this category include BFAST [91], BitMapper [27], mrFAST with FastHASH [28], mrsFAST [29], SHRiMP [78], SHRiMP2 [92], RazerS [64], Maq [87], Hobbes [62], drFAST [93], MOSAIK [94], SOAP [86], Saruman [95] (GPU), ZOOM [89], and RMAP [88]. Hash-based mappers build a very comprehensive but overly large candidate location pool and rely on seed filters and local alignment techniques to remove incorrect mappings from consideration in the verification step. Mappers in this category are able to find all correct mappings of a read, but waste computational resources for identifying and rejecting incorrect mappings. As a result, they are slower than BWT-FM-based mappers. Next, we explain mrFAST mapper as an example of this category.

mrFAST (> version 2.5) [28] first builds a hash table to index fixed-length seeds (typically 10-13 bp) from the reference genome . It then applies a seed location filtering mechanism, called Adjacency Filter, on the hash table to reduce the false seed locations. It divides each query read into smaller fixed-length seeds to query the hash table for their associated seed locations. Given an edit distance threshold, Adjacency Filter requires N-E seeds to exactly match adjacent locations, where N is the number of the seeds and E is the edit distance threshold. Finally, mrFAST tries to extend the read at each of the seed locations by aligning the read to the reference fragment at the seed location via Levenshtein edit distance [38] with Ukkonen's banded algorithm [96]. One drawback of this seed filtering is that the presence of one or more substitutions in any seed is counted by the Adjacency Filter as a single mismatch. The effectiveness of the Adjacency Filter for substitutions and indels diminishes when E becomes larger than 3 edits.

A recent work in [97] shows that by removing redundancies in the reference genome and also across the reads, seed-and-extend mappers can be faster than BWT-FM-based mappers. This space-efficient approach uses a similar idea presented in [89]. A hybrid method that incorporates the advantages of each approach can be also utilized, such as BWA-MEM [30].

2.2 Acceleration of Read Mappers

A majority of read mappers are developed for machines equipped with the general-purpose central processing units (CPUs). As long as the gap between the CPU computing speed and the very large amount of sequencing data widens, CPU-based mappers become less favorable due to their limitations in accessing data [55, 56, 9, 2, 57, 58, 5, 32, 59, 60]. To tackle this challenge, many attempts were made to accelerate the operations of read mapping. We survey in Figure 2.1 the existing read mappers implemented in various acceleration platforms. FPGA-based read mappers often demonstrate one to two orders of magnitude speedups against their GPU-based counterparts [56, 6]. Most of the existing works used hardware platforms to only accelerate the dynamic programming algorithms (e.g., Smith-Waterman algorithm [26]), as these algorithms contributed significantly to the overall running time of read mappers. Most existing works can be divided into three main approaches: (1) Developing seed filtering mechanism to reduce the seed location list, (2) Accelerating the computationally expensive alignment algorithms using algorithmic development or hardware accelerators, and (3) Developing pre-alignment filtering heuristics to reduce the number of incorrect mappings before being examined by read alignment. We describe next each of these three acceleration efforts in detail.

2.2.1 Seed Filtering

The first approach to accelerate today's read mapper is to filter the seed location list before performing read alignment.

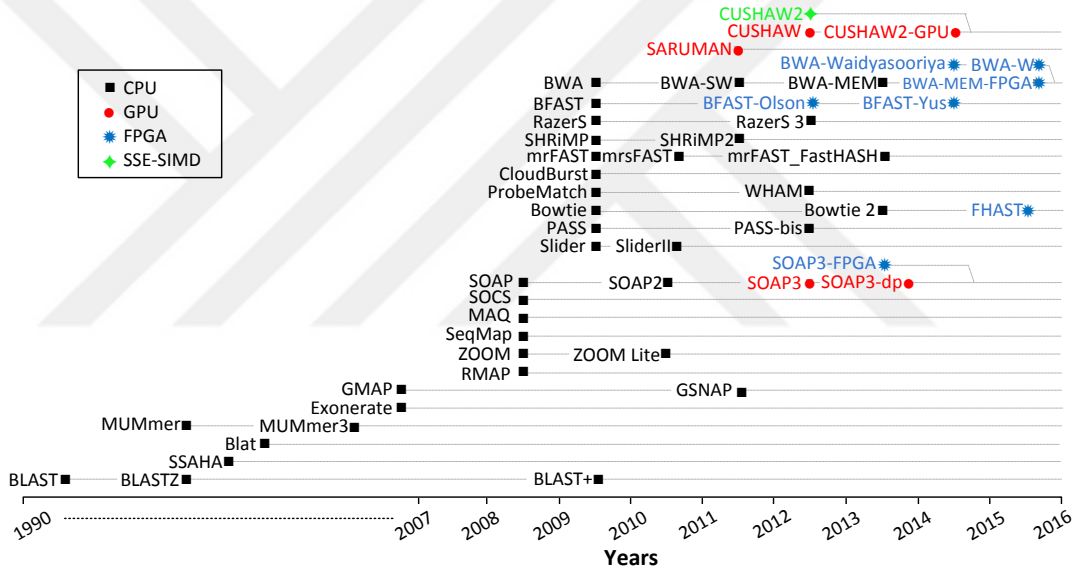

Figure 2.1: Timeline of read mappers. CPU-based mappers are plotted in black, GPU accelerated mappers in red, FPGA accelerated mappers in blue and SSE-based mappers in green. Grey dotted lines connect related mappers (extensions or new versions). The names in the timeline are exactly as they appear in publications, except: SOAP3-FPGA [1], BWA-MEM-FPGA [2], BFAST-Olson [3], BFAST-Yus [4], BWA-Waidyasooriya [5], and BWA-W [6].

This is the basic principle of nearly all seed-and-extend mappers. Seed filtering is based on the observation that if two sequences are potentially similar, then they share a certain number of seeds. Seeds (sometimes called q-grams or k-mers) are short subsequences that are used as indices into the reference genome to reduce the search space and speed up the mapping process. Modern mappers extract short subsequences from each read and use them as a key to query the previously built large reference index database. The database returns the location lists for each seed. The location list stores all the occurrence locations of each seed in the reference genome. The mapper then examines the optimal alignment between the read and the reference segment at each of these seed locations. The performance and accuracy of seed-and-extend mappers depend on how the seeds are selected in the first stage. Mappers should select a large number of non-overlapping seeds while keeping each seed as infrequent as possible for full sensitivity [68, 98, 28]. There is also a significant advantage to selecting seeds with unequal lengths, as possible seeds of equal lengths can have drastically different levels of frequencies. Finding the optimal set of seeds from read sequences is challenging and complex, primarily because the associated search space is large and it grows exponentially as the number of seeds increases. There are other variants of seed filtering based on the pigeonhole principle [27, 61], non-overlapping seeds [28], gapped seeds [99, 63], variable-length seeds [68], random permutation of subsequences [100], or full permutation of all possible subsequences [34].

2.2.2 Accelerating Read Alignment

The second approach to boost the performance of read mappers is to accelerate read alignment step. One of the most fundamental computational steps in most bioinformatics analyses is the detection of the differences/similarities between two genomic sequences. Edit distance and pairwise alignment are two approaches to achieve this step, formulated as approximate string matching [33]. Edit distance approach is a measure of how much the sequences differ. It calculates the minimum number of edits needed to convert one sequence into the other.

The higher the distance, the more different the sequences from one another. Commonly allowed edit operations include deletion, insertion, and substitution of characters in one or both sequences. Pairwise alignment is a way to identify regions of high similarity between sequences. Each application employs a different edit model (called scoring function), which is then used to generate an alignment score. The latter is a measure of how much the sequences are alike. Any two sequences have a single edit distance value but they can have several different alignments (i.e., ordered lists of possible edit operations and matches) with different alignment scores. Thus, alignment algorithms usually involve a backtracking step for providing the optimal alignment (i.e., the best arrangement of the possible edit operations and matches) that has the highest alignment score. Depending on the demand, pairwise alignment can be performed as global alignment, where two sequences of the same length are aligned end-to-end, or local alignment, where subsequences of the two given sequences are aligned. It can also be performed as semi-global alignment (called glocal), where the entirety of one sequence is aligned towards one of the ends of the other sequence.

The edit distance and pairwise alignment approaches are non-additive measures [101]. This means that if we divide the sequence pair into two consecutive subsequence pairs, the edit distance of the entire sequence pair is not necessarily equivalent to the sum of the edit distances of the shorter pairs. Instead, we need to examine all possible prefixes of the two input sequences and keep track of the pairs of prefixes that provide an optimal solution. Enumerating all possible prefixes is necessary for tolerating edits that result from both sequencing errors [36] and genetic variations [37]. Therefore, they are typically implemented as dynamic programming algorithms to avoid re-computing the edit distance of the prefixes many times. These implementations, such as Levenshtein distance [38], Smith-Waterman [26], and Needleman-Wunsch [39], are still inefficient as they have quadratic time and space complexity (i.e., $O(m^2)$ for a sequence length of m). Many attempts were made to boost the performance of existing sequence aligners. Despite more than three decades of attempts, the fastest known edit distance algorithm [102] has a running time of $O(m^2/\log_2 m)$ for sequences of length m, which is still nearly quadratic [103].

Therefore, more recent works tend to follow one of two key new directions to boost the performance of sequence alignment and edit distance implementations: (1) Accelerating the dynamic programming algorithms using hardware accelerators. (2) Developing filtering heuristics that reduce the need for the dynamic programming algorithms, given an edit distance threshold. **Hardware accelerators are becoming increasingly popular for speeding up the computationally-expensive alignment and edit distance algorithms [104, 105, 106, 107]**. Hardware accelerators include multi-core and SIMD (single instruction multiple data) capable central processing units (CPUs), graphics processing units (GPUs), and field-programmable gate arrays (FPGAs). The classical dynamic programming algorithms are typically accelerated by computing only the necessary regions (i.e., diagonal vectors) of the dynamic programming matrix rather than the entire matrix, as proposed in Ukkonen's banded algorithm [96]. The number of the diagonal bands required for computing the dynamic programming matrix is $2E+1$, where E is a user-defined edit distance threshold. The banded algorithm is still beneficial even with its recent sequential implementations as in Edlib [8]. The Edlib algorithm is implemented in C for standard CPUs and it calculates the banded Levenshtein distance. Parasail [9] exploits both Ukkonen's banded algorithm and SIMD-capable CPUs to compute a banded alignment for a sequence pair with user-defined scoring matrix and affine gap penalty. SIMD instructions offer significant parallelism to the matrix computation by executing the same vector operation on multiple operands at once.

Multi-core architecture of CPUs and GPUs provides the ability to compute alignments of many sequence pairs independently and concurrently [57, 58]. GSWABE [58] exploits GPUs (Tesla K40) for a highly-parallel computation of global alignment with affine gap penalty. CUDASW++ 3.0 [59] exploits the SIMD capability of both CPUs and GPUs (GTX690) to accelerate the computation of the Smith-Waterman algorithm with affine gap penalty. CUDASW++ 3.0 provides only the optimal score, not the optimal alignment (i.e., no backtracking step).

Other designs, for instance FPGASW [56], exploit the very large number of hardware execution units in FPGAs (Xilinx VC707) to form a linear systolic array [108]. Each execution unit in the systolic array is responsible for computing the value of a single entry of the dynamic programming matrix. The systolic array computes a single vector of the matrix at a time. The data dependencies between the entries restrict the systolic array to computing the vectors sequentially (e.g., top-to-bottom, left-to-right, or in an anti-diagonal manner). FPGA acceleration platform can also provide more speedup to big-data computing frameworks -such as Apache Spark- for accelerating BWA-MEM [30]. By this integration, Chen et al. [109] achieve 2.6x speedup over the same cloud-based implementation but without FPGA acceleration [110]. FPGA accelerators seem to yield the highest performance gain compared to the other hardware accelerators [55, 56, 6, 5]. However, many of these efforts either simplify the scoring function, or only take into account accelerating the computation of the dynamic programming matrix without providing the optimal alignment as in [65, 66, 59]. Different scoring functions are typically needed to better quantify the similarity between two sequences [111, 112]. The backtracking step required for the optimal alignment computation involves unpredictable and irregular memory access patterns, which poses a difficult challenge for efficient hardware implementation. Comprehensive surveys on hardware acceleration for computational genomics appeared in [104, 105, 106, 107, 40]

2.2.3 False Mapping Filtering

The third approach to accelerate read mapping is to incorporate a pre-alignment filtering technique within the read mapper, before read alignment step. This filter is responsible for quickly excluding incorrect mappings in an early stage (i.e., as a pre-alignment step) to reduce the number of false mappings (i.e., mappings that have more edits than the user-defined threshold) that must be verified via dynamic programming. Existing filtering techniques include the so-called shifted Hamming distance (SHD) [7], which we explain next.

2.2.3.1 Shifted Hamming Distance (SHD)

SHD enables pre-alignment filtering with the existence of indels and substitutions. Instead of building a single bit-vector using a pairwise comparison as Hamming distance does, SHD builds $2E+1$ bit-vectors, where E is the user-defined edit distance threshold. This is similar to the Ukkonen's banded algorithm [96]. Each bit-vector is built by gradually shifting the read sequence and then performing a pairwise comparison. The shifting process is inevitable in order to skip the deleted (or inserted) character and examine the subsequent matches. SHD merges all masks using bitwise AND operation. Due to the use of AND operation, a zero (i.e., pairwise match) at any position in the $2E+1$ masks leads to a '0' in the resulting output of the AND operation at the same position. The last step is to count the positions that have a value other than '0'. SHD decides if the mapping is correct based on whether the number of the mismatches exceeds the edit distance threshold or not. SHD heavily relies on bitwise operations such as shift, XOR, and AND. This makes SHD suitable for bitwise hardware implementations (e.g., FPGAs and SIMD-enabled CPUs).

Our crucial observation is that SHD examines each mapping, throughout the filtering process, by performing expensive computations unnecessarily; as SHD uses the same amount of computation regardless the type of edit. SHD is also implemented using Intel SSE, which limits the supported read length up to only 128 bp (due to SIMD register size). The filtering mechanism of SHD also introduces inaccuracy in its filtering decision as we investigate and demonstrate in Chapter 3 and in our experimental evaluation, Chapter 9.

2.3 Summary

We survey in this chapter the existing key directions that aim at accelerating all or part of the operations of modern read mappers. We analyze these attempts and provide the pros and cons of each direction.

We present three main acceleration approaches, including (1) seed filtering, (2) accelerating the dynamic programming algorithm, (3) pre-alignment filtering. In Figure 1.1, we illustrate that the state-of-the-art mapper mrFAST with FastHASH [28] generates more than 90% of the potential mappings as incorrect ones, although it implements a seed filtering mechanism (Adjacency Filter) and SIMD-accelerated banded Levenshtein edit distance algorithm. This demonstrates that the development of a fundamentally new, fast, and efficient pre-alignment filter is the utmost necessity. Note that there is still no work, to best of our knowledge, on specialized hardware acceleration of pre-alignment filtering techniques.

Chapter 3

Understanding and Improving Pre-alignment Filtering Accuracy

In this chapter, we firstly provide performance metrics used to evaluate pre-alignment filtering techniques. The essential performance metrics are filtering speed and filtering accuracy. We secondly study the causes of filtering inaccuracy of the state-of-the-art pre-alignment filter, SHD [7], aiming at eliminating them. We find four key causes and provide a detailed investigation along with examples on these inaccuracy sources. This is the first work to comprehensively assess the filtering inaccuracy of the SHD algorithm [7] and provide recommendations for desirable improvements.

3.1 Pre-alignment Filter Performance Metrics

An ideal pre-alignment filter should be both fast and accurate in rejecting the incorrect mappings. Meanwhile, it should also preserve all correct mappings. Incorrect mapping is defined as a sequence pair that differs by more than the edit distance threshold. Correct mapping is defined as a sequence pair that has edits less than or equal to the edit distance threshold.

Next, we describe the performance metrics that are necessary to evaluate the speed and accuracy of existing and future pre-alignment filtering algorithms.

3.1.1 Filtering Speed

The filtering speed is defined as the time spent by the pre-alignment filter in examining all the incoming mappings. We always want to increase the speed of the pre-alignment filter to compensate the computation overhead introduced by its filtering technique.

3.1.2 Filtering Accuracy

3.1.2.1 False Accept Rate

The false accept rate (or false positive rate) is the ratio between the incorrect mappings that are falsely accepted by the filter and the incorrect mappings that are rejected by optimal read alignment algorithm. Similarly, a mapping is considered as a false positive if read alignment accepts it but pre-alignment filter rejects it. We always want to minimize the false accept rate.

3.1.2.2 True Accept Rate

The true accept rate (or true positive rate) is the ratio between the correct mappings that are accepted by the filter and the correct mappings that are accepted by optimal read alignment algorithm. The true accept rate should always equal to 1.

3.1.2.3 False Reject Rate

The false reject rate (or false negative rate) is the ratio between the correct mappings that are rejected by the filter and the correct mappings that are accepted by optimal read alignment algorithm. The false reject rate should always equal to 0.

3.1.2.4 True Reject Rate

The true reject rate (or true negative rate) is the ratio between the incorrect mappings that are rejected by the filter and the incorrect mappings that are rejected by optimal read alignment algorithm. We always want to maximize the true reject rate. In fact, the true reject rate is inversely proportional to the false accept rate. However, they can be equivalent in ratio in case of all mappings are correct and accepted by the filter.

3.1.3 End-to-End Alignment Speed

Can very fast filter with high false accept rate be better than more accurate filter at the cost of its speed? The answer to this question is not trivial because both speed and accuracy contribute to the overall speed of read alignment. The only way to answer this question is evaluate the effect of such filter on the overall speed of read alignment step. Thus, we need to evaluate the end-to-end alignment speed. This includes the integration of pre-alignment filter with read alignment step and evaluate the acceleration rate. Another crucial observation is that pre-alignment filter applies heuristic approach, which can be optimal for some alignment cases while it fails in other cases. Thus, filter that performs best for specific read set and edit distance threshold may not perform well for other read sets and edit distance thresholds. The user-defined edit distance threshold, E, is usually less than 5% of the read length [27, 7, 113, 62].

3.2 On the False Filtering of SHD algorithm

In this section, we investigate the potential causes of filtering inaccuracy that are introduced by the state-of-the-art filter, SHD [7] (we describe the algorithm in Chapter 2). We also provide examples that illustrate each of these causes. Adding an additional fast filtering heuristic before the verification step in a read mapper can be beneficial. But, such a filter can be easily worthless if it allows a high false accept rate. Even though the incorrect mappings that pass SHD are discarded later by the read alignment step (as it has zero false accept rate and zero false reject rate), they can dramatically increase the execution time of read mapper by causing a mapping to be examined twice unnecessarily by both the filtering step as well as read alignment step. Below, we describe four major sources of false positives that are introduced by the filtering strategy of SHD.

3.2.1 Random Zeros

The first source of false accept rate of SHD [7] is the random zeros that appear in the individual shifted Hamming mask. Although they result from a pairwise comparison between a shifted read and a reference segment, we refer to them as random zeros because they are sometimes meaningless and are not part of the correct alignment. SHD ANDs all shifted Hamming masks together with the idea that all '0's in the individual Hamming masks propagate to the final bit-vector, thereby preserving the information of individual matching subsequences. Due to the use of AND operation, a zero at any position in the $2E+1$ Hamming masks leads to a '0' in the resulting final bit-vector at the same position. Hence, even if some Hamming masks show a mismatch at that position, a zero in some other masks leads to a match ('0') at the same position. This tends to underestimate the actual number of edits and eventually causes some incorrect mappings to pass. To fix this issue, SHD proposes the so-called *speculative removal of short-matches (SRS)* before ANDing the masks, which flips short streaks of '0's in each mask into '1's such that they do not mask out '1's in other Hamming masks.

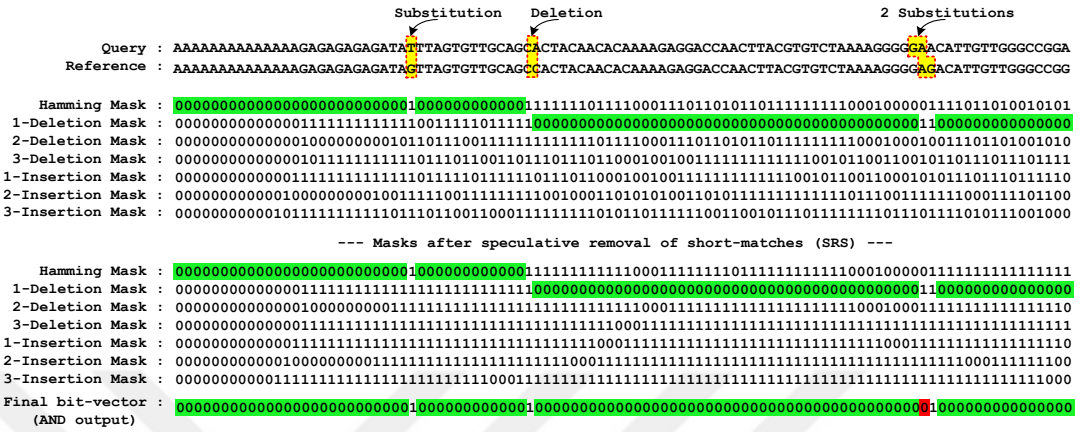

Figure 3.1: An example of a mapping with all its generated masks, where the edit distance threshold (E) is set to 3. The green highlighted subsequences are part of the correct alignment. The red highlighted bit in the final bit-vector is a wrong alignment provided by SHD. The correct alignment (highlighted in yellow) shows that there are three substitutions and a single deletion, while SHD detects only two substitutions and a single deletion.

We illustrate this method in Figure 3.1. The number of zeros to be amended (SRS threshold) is set by default to two. That is, bit streams such as 101, 1001 are replaced with 111 and 1111, respectively. The $2E+1$ masks contain other zeros that are part of the correct alignment. For example, Figure 3.1 shown a segment of consecutive matches in one-step right-shifted mask. This segment indicates that there is a single deletion that occurred in the read sequence. Unlike these meaningful zeros, random amended zeros can be anywhere in the masks except the two ends of each mask. However, the length and the position of these zeros are unpredictable. They can have any length that makes the SRS method ineffective at handling these random zeros. There is no clear theory behind the exact SRS threshold to be used to eliminate such zeros. SRS successfully reduce some of the falsely accepted mappings, but it also introduces its own source of falsely accepted mappings. Choosing a small SRS threshold helps, but does not provide any guarantee, to get rid of some of these random zeros. Choosing a larger SRS threshold can be risky, since, with such a large threshold, SHD might no longer be able to distinguish whether any streak of consecutive zeros is generated by random chance or it is part of the correct alignment. This results in SHD ignoring most of the exact matching subsequences and causes an all-'1' final bit-vector.

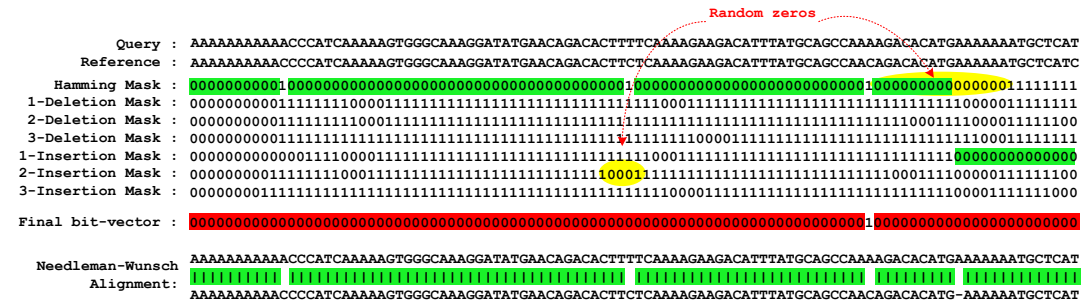

Figure 3.2: Examples of an incorrect mapping that passes the SHD filter due to the random zeros. While the edit distance threshold is 3, a mapping of 4 edits (as examined at the end of the figure by Needleman-Wunsch algorithm) passes as a falsely accepted mapping.

In Figure 3.2, we provide an example where random zeros dominate and lead to a zero in the final bit-vector at their corresponding locations. SRS can address the inaccuracy caused by the random 3-bit zeros, which are highlighted by the left arrow, using an SRS threshold of 3. However, SRS is still unable to solve the inaccuracy caused by the 15-bit zeros that are highlighted by the right arrow. This is due to the fact that the 15-bit zeros are part of the correct alignment and hence amending them to ones can introduce more falsely accepted mappings.

3.2.2 Conservative Counting

The second source of high false accept rate of SHD [7] is related to the way in which SHD counts the edits in the final bit-vector. Amending short streaks of '0's to '1's could cause correct mappings to be mistakenly filtered out, as it may produce multiple ones in the final bit-vector. To ensure that it does not overcount the number of edits, SHD always assumes the streaks of '1's in the final bit-vector as a side effect of the SRS amendment, and counts only the minimum number of edits that potentially generate such a streak of '1's. The total number of edits reported by SHD can be much smaller than the actual number of edits.

Figure 3.3: An example of an incorrect mapping that passes the SHD filter due to conservative counting of the short streak of '1's in the final bit-vector.

For instance, as illustrated in Figure 3.3, three consecutive substitutions render a streak of three '1's in the final bit-vector. But since SHD always assumes the middle '1' is the result of an amended '0' by SRS, SHD will only consider the streak of three '1's as a single edit and let it pass, even if the edit distance threshold is less than three.

3.2.3 Leading and Trailing Zeros

The third source of high false accept rate of SHD [7] is the streaks of zeros that are located at any of the two ends of each mask. Hence we refer to them as leading and trailing zeros. These streaks of zeros can be in two forms: (1) the vacant bits that are caused by shifting the read against the reference segment and (2) the streaks of zeros that are not vacant bits. SHD generates $2E+1$ masks using arithmetic left-shift and arithmetic right-shift operations. For both the left and right directions, the right-most and the left-most vacant bits, respectively, are filled with '0's. The number of vacant zeros depends on the number of shifted steps for each mask, which is at most equal to the edit distance threshold. The second form of the leading and trailing zeros is the zeros that are located at the two ends of the Hamming masks and are not vacant zeros. These streaks of zeros result from the pairwise comparison (i.e., bitwise XOR). They differ from the vacant bits in that their length is independent of the edit distance threshold.

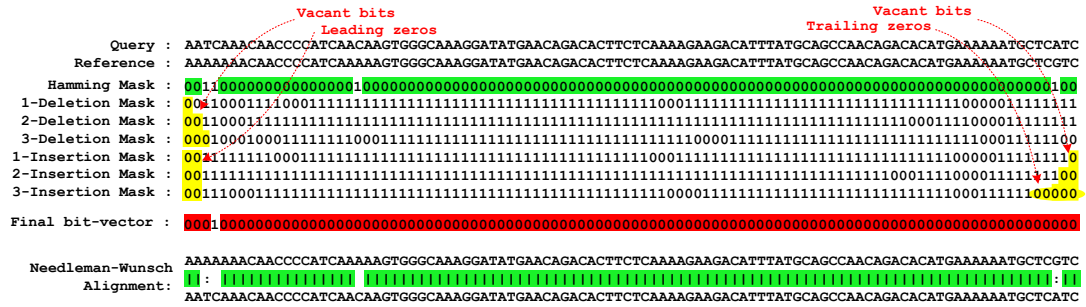

Figure 3.4: Examples of an invalid mapping that passes the SHD filter due to the leading and trailing zeros. We use an edit distance threshold of 3 and an SRS threshold of 2. While the regions that are highlighted in green are part of the correct alignment, the wrong alignment provided by SHD is highlighted in red. The yellow highlighted bits indicate a source of falsely accepted mapping.

The main issue with both forms of leading and trailing zeros is that they always dominate, even if some Hamming masks show a mismatch at that position (due to the use of the AND operation). This gives the false impression that the read and the reference have a smaller edit distance, even when they differ significantly, as explained in Figure 3.4. SRS does not address the inaccuracy caused by the leading and trailing zeros by amending such zeros to ones, due to two reasons: (1) the number of these consecutive zeros is not fixed and thus they can be longer than the SRS threshold, (2) these consecutive zeros are not surrounded by ones and hence even if SRS threshold is greater than two bits, they are not eligible to be amended.

3.2.4 Lack of Backtracking

The last source of false accept rate of in SHD [7] is the inability of SHD to backtrack (after generating the final bit-vector) the location of each long identical subsequence (i.e., the mask that originates the identical subsequence), which is part of the correct alignment. The source of each subsequence provides a key insight into the actual number of edits between each two subsequences. That is, if a subsequence is located in a 2-step right shifted mask, it should indicate that there are two deletions before this subsequence.

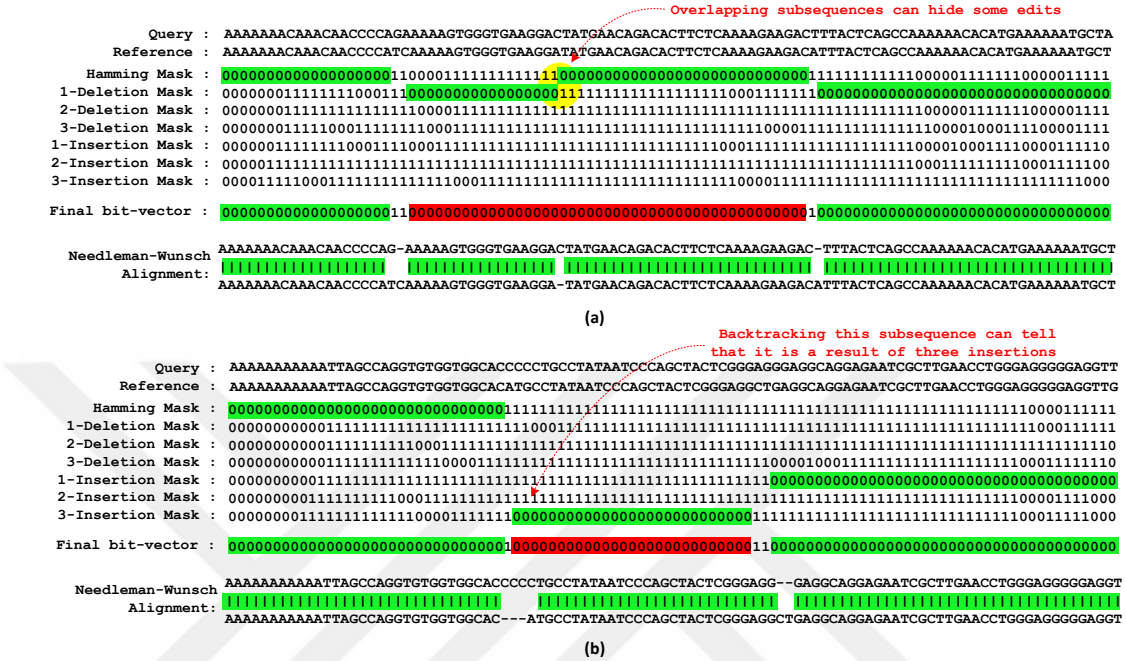

Figure 3.5: Examples of incorrect mappings that pass the SHD filter due to (a) overlapping identical subsequences, and (b) lack of backtracking.

SHD does not relate this important fact to the number of edits in the final bit-vector. The lack of backtracking causes two types of falsely accepted mapping: (1) the first type appears clearly when two of the identical subsequences, in the individual Hamming masks, are overlapped or nearly overlapped, (2) the second type happens when the identical subsequences come from different Hamming masks. The issue with the first type (i.e., overlapping subsequences) is the fact that they appear as a single identical subsequence in the final bit-vector, due to the use of AND operation. An example of this scenario is given in Figure 3.5 (a). This tends to hide some of the edits and eventually causes some invalid mappings to pass. The second type of false positives caused by the lack of backtracking happens, for example, when an identical subsequence comes from the first Hamming mask (i.e., with no shift) and the next identical subsequence comes from the 3-step left shifted mask. This scenario reveals that the number of edits between the two subsequences should not be less than three insertions.

However, SHD inaccurately reports it as a single edit (due to ANDing all Hamming masks without backtracking the source of each streak of zeros), as illustrated in Figure 3.5 (b). Keeping track of the source mask of each identical subsequence prevents such false positives and helps to reveal the correct number of edits.

3.3 Discussion on Improving the Filtering Accuracy

In this section, we provide our own observations and recommendations based on our comprehensive accuracy analysis of SHD filter [7]. We make two crucial observations. (1) The **first observation** is that handling the short streaks of '0's (i.e., using the SRS method that we discuss above) is indeed inefficient. These "noisy" streaks do not have determined properties, as their length and number are unpredictable (random-like). They introduce their own sources of falsely accepted mappings and do not contribute any useful information. Therefore, future filtering strategies should avoid processing such short streaks of '0's. (2) The **second observation** is that the correct (desired) alignment always contains all the longest non-overlapping identical subsequences. This turns our attention to focusing on the long matches (that are highlighted in green in all previous figures, i.e., Figure 3.1 to Figure 3.5) in each Hamming mask. We find that the long non-overlapping subsequences of consecutive zeros have two interesting properties. (1) There is an upper bound on their quantity. With the existence of E edits, there are at most $E+1$ non-overlapping identical subsequences shared between a pair of sequences. The total length of these non-overlapping subsequences is equal to m-E, where m is the read length. (2) The source mask of each long subsequence provides an insight into the number of edits between this subsequence and its preceding one. These two observations motivate us to incorporate long-match-awareness into the design of our filtering strategy and ignore processing noisy short matches.

3.4 Summary

We identify four causes that introduce the filtering inaccuracy of the SHD [7] algorithm, namely, the random zeros, conservative counting, leading and trailing zeros, and lack of backtracking. Based on these four sources of falsely accepted mapping, we observe that there are still opportunities for further improvements on the accuracy of the state-of-the-art filter, SHD, which we discuss next.

Chapter 4

The First Hardware Accelerator for Pre-Alignment Filtering

In this chapter, we introduce a new FPGA-based accelerator architecture for hardware-aware pre-alignment filtering algorithms. To our knowledge, this is the first work that exploit reconfigurable hardware platforms to accelerate pre-alignment filtering. A fast filter designed on a specialized hardware platform can drastically expedite alignment by reducing the number of locations that must be verified via dynamic programming. This eliminates many unnecessary expensive computations, thereby greatly improving overall run time.

4.1 FPGA as Acceleration Platform

We select FPGA as an acceleration platform for our proposed pre-alignment filtering algorithms, as its architecture offers large amounts of parallelism [105, 114, 115]. The use of FPGA as an acceleration platform can yield significant performance improvements, especially for massively parallel algorithms.

FPGAs are the most commonly used form of reconfigurable hardware engines today in bioinformatics [116, 117, 106], and their computational capabilities are greatly increasing every generation due to increased number of transistors on the FPGA chip. An FPGA chip can be programmed (i.e., configured) to include a very large number of hardware execution units that are custom-tailored to the problem at hand.

4.2 Overview of Our Accelerator Architecture

One of our aims is to accelerate our new pre-alignment filtering algorithms (that we describe in the next three chapters) by leveraging the capabilities and parallelism of FPGAs. To this end, we build our own hardware accelerator that consists of an FPGA engine as an essential component and a CPU. We present in Figure 4.1 the overall architecture of our FPGA-based accelerator. The CPU is responsible for acquiring and encoding the short reads and transferring the data to and from the FPGA. The FPGA engine is equipped with PCIe transceivers, Read Controller, Result Controller, and a set of filtering units that are responsible for examining the read alignment. The workflow of the accelerator starts with reading a repository of short reads and seed locations. All reads are then converted into their binary representation that can be understood by the FPGA engine. Encoding the reads is a preprocessing step and accomplished through a Read Encoder at the host before transmitting the reads to the FPGA chip. Next, the encoded reads are transmitted and processed in a streaming fashion through the fastest communication medium available on the FPGA board (i.e., PCIe). We design our system to perform alignment filtering in a streaming fashion: the accelerator receives a continual stream of short reads, examines each alignment in parallel with others, and returns the decision (i.e., a single bit of value '1' for an accepted sequences and '0' for a rejected sequences) back to the CPU instantaneously upon processing.

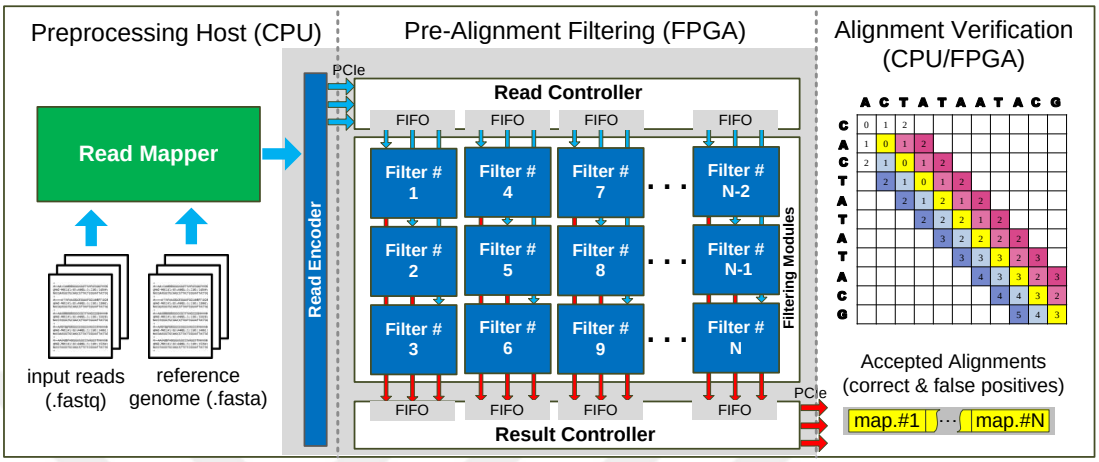

Figure 4.1: Overview of our accelerator architecture.

4.2.1 Read Controller

The Read Controller on the FPGA side is responsible for two main tasks. First, it permanently assigns the first data chunk as a reference sequence for all processing cores. Second, it manages the subsequent data chunks and distributes them to the processing cores. The first processing core receives the first read sequence and the second core receives the second sequence and so on, up to the last core. It iterates the data chunk management task until no more reads are left in the repository.

4.2.2 Result Controller

Following similar principles as the Read Controller, the Result Controller gathers the output results of the filtering units. Both the Read Controller and the Result Controller preserve the original order of reads as in the repository (i.e., at the host). This is critical to ensure that each read will receive its own alignment filtering result. The results are transmitted back to the CPU side in a streaming fashion and then saved in the repository.

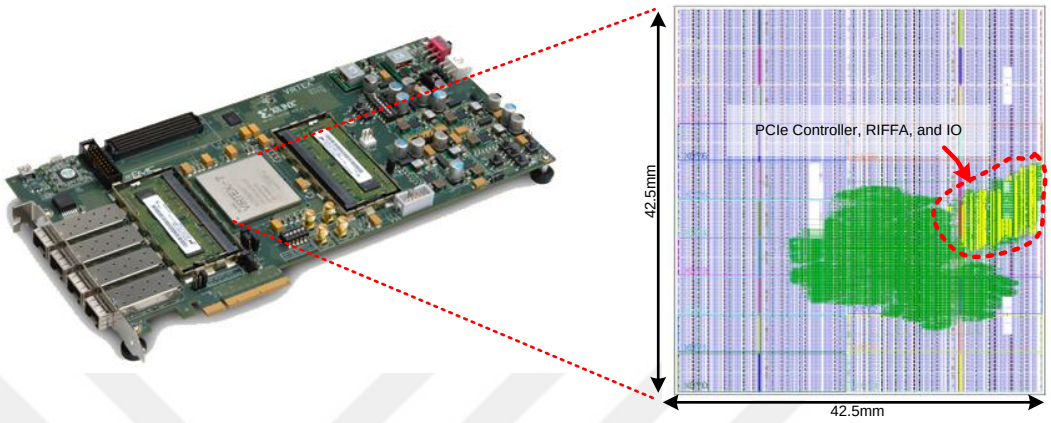

Figure 4.2: Xilinx Virtex-7 FPGA VC709 Connectivity Kit and chip layout of our implemented accelerator.

4.3 Parallelization

We design our hardware accelerator to exploit the large amounts of parallelism offered by FPGA architectures [105, 114, 115]. We take advantage of the fact that alignment filtering of one read is inherently independent of filtering of another read. We therefore can examine many reads in a parallel fashion. In particular, instead of handling each read in a sequential manner, as CPU-based filters (e.g., SHD) do, we can process a large number of reads at the same time by integrating as many hardware filtering units as possible (constrained by chip area) in the FPGA chip. Each filtering unit is a complete alignment filter and can handle a single read at a time. Our hardware accelerator contains large number of filtering units that their number can be configured by the user. Each filtering unit provides pre-alignment filtering individually from all other units. We use the term "filtering unit" in this work to refer to the entire operation of the filtering process involved. Filtering units are part of our architecture and are unrelated to the term "CPU core" or "thread".

4.4 Hardware Implementation

Our hardware implementation of our accelerator is independent from specific FPGA-platform as it does not rely on any vendor-specific computing elements (e.g., intellectual property cores). However, each FPGA board has different features and hardware capabilities that can directly or indirectly affect the performance and the data throughput of the design. In fact, the number of filtering units is determined by the maximum data throughput and the available FPGA resources. We use a Xilinx Virtex 7 VC709 board [118] to implement our accelerator architecture. We build the FPGA design with Vivado 2015.4 in synthesizable Verilog. We preset the chip layout of out hardware accelerator in Figure 4.2. The maximum operating frequency of our accelerator and the VC709 board is 250 MHz. At this frequency, we observe a data throughput of nearly 3.3 GB/s, which corresponds to 13.3 billion bases per second. This nearly reaches the peak throughput of 3.64 GB/s provided by the RIFFA [119] communication channel that feeds data into the FPGA using Gen3 4-lane PCIe.

4.5 Summary

We introduce in this chapter a new hardware accelerator architecture that exploit the large amounts of parallelism offered by FPGA architectures to boost the performance of our pre-alignment filters. Our hardware accelerator processes the pre-alignment filtering for each sequence pair independently from each another. We therefore can examine many reads in a parallel fashion. We build the hardware architecture of our hardware accelerator using many hardware filtering units, where each filtering unit is a complete pre-alignment filter and can handle a single read at a time. To take full advantage of the capabilities and parallelism of our FPGA accelerator, each pre-alignment filtering unit needs to be designed and implemented using FPGA-supported operations such as bitwise operations, bit shifts, and bit count. Next, we discuss our proposed pre-alignment filters that can be included in our FPGA accelerator as a filtering unit.

Chapter 5

GateKeeper: Fast Hardware Pre-Alignment Filter

In this chapter, we introduce a new FPGA-based fast alignment filtering technique (called GateKeeper) that acts as a pre-alignment step in read mapping. Our filtering technique improves and accelerates the state-of-the-art SHD filtering algorithm [7] using new mechanisms and FPGAs.

5.1 Overview

Our new filtering algorithm has two properties that make it suitable for an FPGA-based implementation: (1) it is highly parallel, (2) it heavily relies on bitwise operations such as shift, XOR, and AND. Our architecture discards the incorrect mappings from the candidate mapping pool in a streaming fashion – data is processed as it is transferred from the host system. Filtering the mappings in a streaming fashion gives the ability to integrate our filter with any mapper that performs alignment, such as Bowtie 2 [31] and BWA-MEM [30]. Our current filter implementation relies on several optimization methods to create a robust and efficient filtering approach.

At both the design and implementation stages, we satisfy several requirements: (1) Ensuring a lossless filtering algorithm by preserving all correct mappings. (2) Supporting both Hamming distance and edit distance. (3) Examining the alignment between a read and a reference segment in a fast and efficient way (in terms of execution time and required resources).

5.2 Methods

Our primary purpose is to enhance the state-of-the-art SHD alignment filter such that we can greatly accelerate pre-alignment by taking advantage of the capabilities and parallelism of FPGAs. To achieve our goal, we design an algorithm inspired by SHD to reduce both the utilized resources and the execution time. These optimizations enable us to integrate more filtering units within the FPGA chip and hence examine many mappings at the same time. We present three new methods that we use in each GateKeeper filtering unit to improve execution time. Our first method introduces a new algorithmic method for performing alignment very rapidly compared to the original SHD. This method provides: (1) fast detection for exact matching alignment and (2) handling of one or more base-substitutions. Our second method supports calculating the edit distance with a new, very efficient hardware design. Our third method addresses the problem of hardware resource overheads introduced due to the use of FPGA as an acceleration platform. All methods are implemented within the hardware filtering unit of our accelerator (see Chapter 4) and thus are performed highly efficiently. We present a flowchart representation of all steps involved in our algorithm in Figure 5.1. Next, we describe the three new methods.

5.2.1 Method 1: Fast Approximate String Matching

We first discuss how to examine a mapping with a given Hamming distance threshold, and later extend our solution to support edit distance.

Our first method aims to quickly detect the obviously-correct alignments that contain no edits or only few substitutions (i.e., less than the user-defined threshold). If the first method detects a correct alignment, then we can skip the other two methods but we still need the optimal alignment algorithms. A read is mappable if the Hamming distance between the read and its seed location does not exceed the given Hamming distance threshold. Hence, the first step is to identify all bp matches by calculating what we call a Hamming mask. The Hamming mask is a bit-vector of '0's and '1's representing the comparison of the read and the reference, where a '0' represents a bp match and a '1' represents a bp mismatch. We need to count only occurrences of '1' in the Hamming mask and examine whether their total number is equal to or less than the user-defined Hamming distance threshold. If so, the mapping is considered to be valid and the read passes the filter. Similarly, if the total number of '1' is greater than the Hamming distance threshold then we cannot be certain whether this is because of the high number of substitutions, or there exist insertions and/or deletions; hence, we need to follow the rest of our algorithm. Our filter can detect not only substitutions but also insertions and deletions in an efficient way, as we discuss next.

5.2.2 Method 2: Insertion and Deletion (Indel) Detection

Our indel detection algorithm is inspired by the original SHD algorithm presented in [7]. If the substitution detection rejects an alignment, then GateKeeper checks if an insertion or deletion causes the violation (i.e., high number of edits). Figure 5.2 illustrates the effect of occurrence of edits on the alignment process. If there are one or more base-substitutions or the alignment is exactly matching, the matching and mismatching regions can be accurately determined using Hamming distance. It also helps detect the matches that are located before the first indel. However, this mask is already generated as part of the first method of the algorithm (i.e., Fast Approximate String Matching). On the other hand, each insertion and deletion can shift multiple trailing bases and create multiple edits in the Hamming mask. Thus, our indel detection method identifies whether the alignment locations of a read are valid, by shifting individual bases.

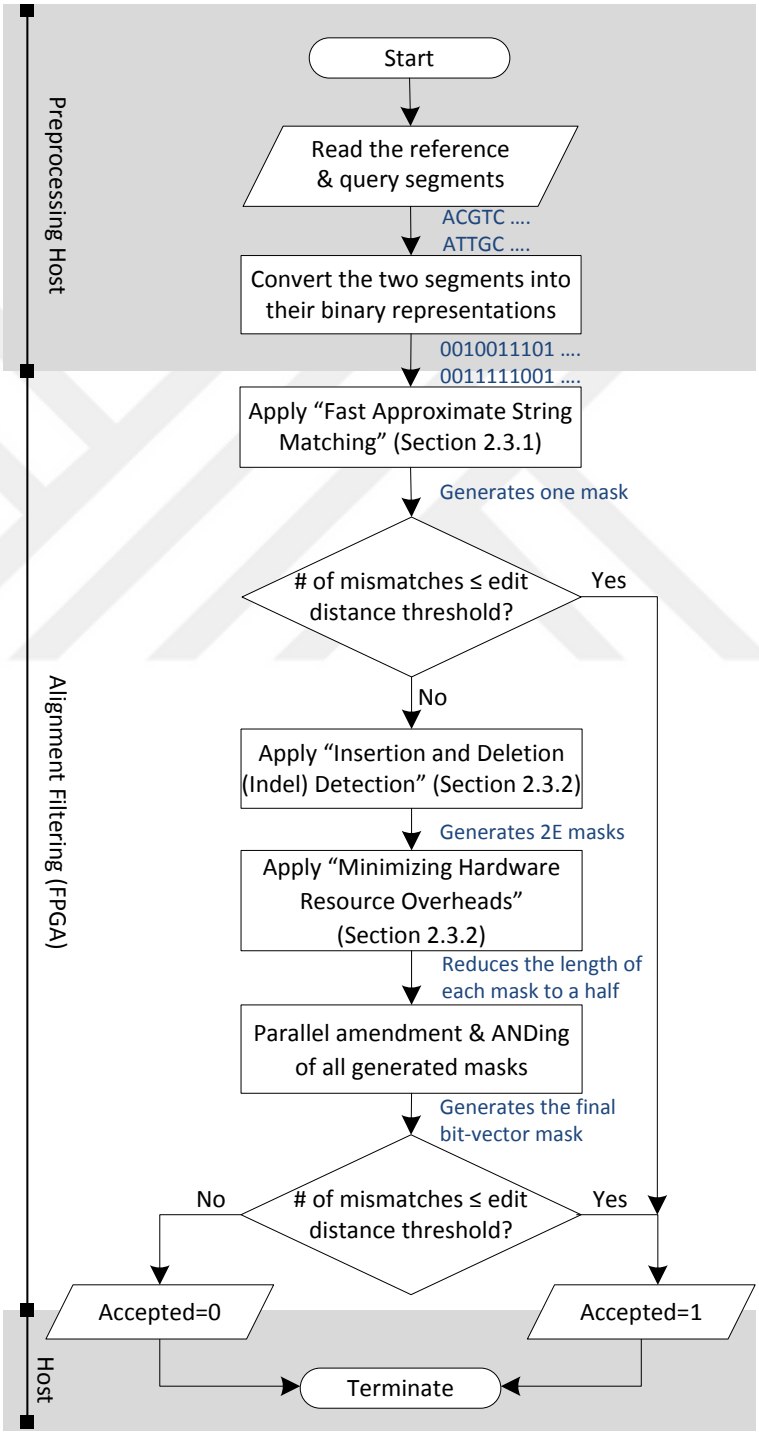

Figure 5.1: A flowchart representation of the GateKeeper algorithm.

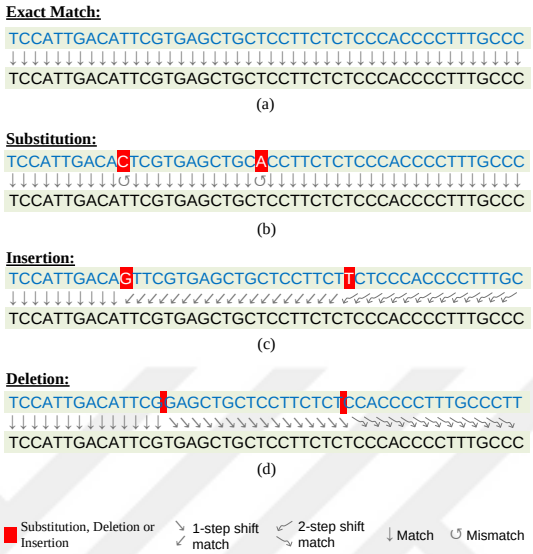

Figure 5.2: An example showing how various types of edits affect the alignment of two reads. In (a) the upper read exactly matches the lower read and thus each base exactly matches the corresponding base in the target read. (b) shows base-substitutions that only affect the alignment at their positions. (c) and (d) demonstrate insertions and deletions, respectively. Each edit has an influence on the alignment of all the subsequent bases.

We need to perform E incremental shifts to the right (or left) direction to detect any read that has E deletions (or insertions), where E is the edit distance threshold. The shift process guarantees to cancel the effect of indel. As we do not have prior knowledge about whether there exist substitutions, or indels, or combination of both, we need to test for every possible case in our algorithm. Thus, GateKeeper generates $2E+1$ Hamming masks regardless the source of the edit. The last step is to merge all the $2E+1$ Hamming masks using a bitwise AND operation. This step tells us where the relevant matching and mismatching regions reside in the presence of edits in the read compared to the reference segment. Similarly to SHD, we apply amending process to the $2E+1$ masks before performing AND operation. In SHD, the amending process is accomplished using a 4-bit packed shuffle (SIMD parallel table-lookup instruction), shift, and OR operations. The number of computations needed is 4 packed shuffle, $4m$ bitwise OR, and three shift operations for each Hamming mask, which is $(7+4m)(2E+1)$ operations, where m is the read length. We find that this is very inefficient for FPGA implementation.

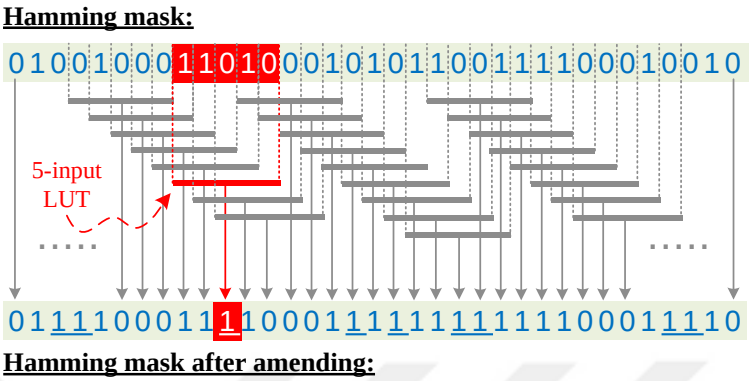

Figure 5.3: Workflow of the proposed architecture for the parallel amendment operations.

To reduce the number of operations, we build a new hardware-based amending process. We propose using dedicated hardware components in FPGA slices. More precisely, rather than shifting the read and then performing packed shuffle to replace patterns of 101 or 1001 to 111 or 1111 respectively, we perform only packed shuffle independently and concurrently for each bit of each Hamming mask. We present the proposed architecture for amendment operation in Figure 5.3. In order to replace all patterns of 101 or 1001 to 111 or 1111 respectively, we use a single 5-input look-up table (LUT) for each bit of the Hamming mask. The first LUT copies the bit value of the first input regardless of its value; even if it is zero, it will not be amended as it is not contributing to the 101 or 1001 pattern.

Likewise for the last LUT. Thus, the total number of LUTs needed is equal to the length of the short read in bases minus 2 for the first and last bases. In each LUT, we consider a single bit of the Hamming mask and two of its right neighboring bits and two of its left neighboring bits. If the input that corresponds to the output has a bit value of one, then the output copies the value of that input bit (as we only amend zeros). Otherwise, using the previous two bits and the following two bits with respect to the input bit, we can replace any zero of the "101" or "1001" patterns independently from other output bits. All bits of the amended masks are generated at the same time, as the propagation delay through an FPGA look-up table is independent of the implemented function [120].

Thus we can process all masks in a parallel fashion without affecting the correctness of the filtering decision. Using this dedicated architecture, we are able to get rid of the four shifting operations and perform the amending process concurrently for all bits of any Hamming mask. Thus, the required number of operations is only $(2E+1)$ instead of $(7+4m)(2E+1)$ for a total of $(2E+1)$ Hamming masks. This saves a considerable amount of the filtering time, reducing it by two orders of magnitude for a read that is 100bp long.

5.2.3 Method 3: Minimizing Hardware Resource Overheads

The short reads are composed of a string of nucleotides from the DNA alphabet {A, C, G, T}. Since the reads are processed in an FPGA platform, the symbols have to be encoded into a unique binary representation. We need 2 bits to encode each symbol. Hence, encoding a read sequence of length m results in a $2m$-bit word. Encoding the reads into a binary representation introduces overhead to accommodate not only the encoded reads but also the Hamming masks as their lengths also double (i.e., $2m$). The issue introduced by encoding the read can be even worse when we apply certain operations on these Hamming masks. For example, the number of LUTs required for performing the amending process on the Hamming masks will be doubled, mainly due to encoding the read.

To reduce the complexity of the subsequent operations on the Hamming masks and save about half of the required amount of FPGA resources, we propose a new solution. We observe that comparing a pair of DNA nucleotides is similar to comparing their binary representations (e.g., comparing A to T is similar to comparing '00' to '11'). Hence, comparing each two bits from the binary representation of the read with their corresponding bits of the reference segment generates a single bit that represents one of two meanings; either match or mismatch between two bases.

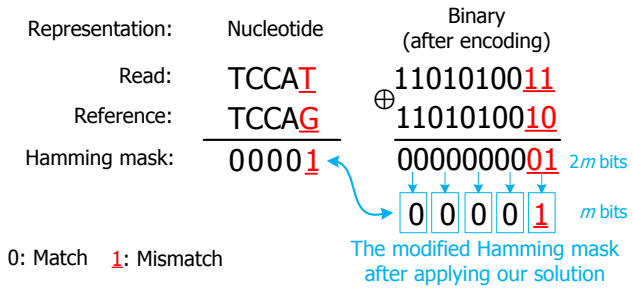

Figure 5.4: An example of applying our solution for reducing the number of bits of each Hamming mask by half. We use a modified Hamming mask to store the result of applying the bitwise OR operation to each two bits of the Hamming mask. The modified mask maintains the same meaning of the original Hamming mask.

This is performed by encoding each two bits of the result of the pairwise comparison (i.e., bitwise XOR) into a single bit of '0' or '1' using OR operations in a parallel fashion, as explained in Figure 5.4. This makes the length of each Hamming mask equivalent to the length of the original read, without affecting the meaning of each bit of the mask. The modified Hamming masks are then merged together in $2E$ bitwise AND operations. Finally, we count the number of ones (i.e., edits) in the final bit-vector mask; if the count is less than the edit distance threshold, the filter accepts the mapping.

5.3 Analysis of GateKeeper Algorithm

In this section, we provide the pseudocode of our GateKeeper algorithm. Algorithm 5.1 presents the main functions. Algorithms 5.2 presents the details of the amending process using in Algorithm 5.1.

Algorithm 5.1: GateKeeper filtering algorithm

Input: Candidate read bit-vector B = { b_1,b_2...b_m }, Reference bit-vector A={ a_1,a_2...a_m }, edit distance threshold E
Output: Pass (return True if the read passes the GateKeeper filter).
Functions: *Amend*: Encodes/amends the masks.
Pseudocode:

```
1: //Calculate Hamming distance first.
2: HammingMask[2E+1] ← A ⊕ B;
3: AmendedMask[2E+1] ← Amend (HammingMask[2E+1]);
4: e ← # of '1's in HammingMask[2E+1] after encoding;
5: if e ≤ E then
6:       Pass ← True;
7: else //Generate 2E masks with incremental shift.
8:      for i ← 1 to E do
9:           HammingMask[i] ← (b>>i) ⊕ a;
10:          AmendedMask[i] ← Amend (HammingMask[i]);
11:          HammingMask[i+E] ← (b<<i) ⊕ a;
12:          AmendedMask[i+E] ←
13:                              Amend(HammingMask[i+E]);
14:     FinalMask = AND(AmendedMask[1 .... 2E+1];
15:     i←1; e←0;
16:     while i < m do //Count the differences.
17:          case (FinalMask[i, i+1, i+2, i+3]):
18:          [0101],[0110],[1001],[1010],[1011],[1101]:
19:          e ← e +2;
20:          [0000]: e ← e;    default: e ← e+1;    i ← i+4;
21:     if e ≤ E then
22:          Pass ← True;
23:     else
24:          Pass ← False;
25: return Pass;
```

Algorithm 5.2: Amend function

Input: Hamming mask bit-vector, $H = \{ H_1, H_2 ... H_{2m} \}$
Output: modified Hamming mask, $C = \{ C_1, C_2 ... C_m \}$
Pseudocode:

```
1:  i ← 0;
2:  while i <m do //Encode Hamming masks
3:      E_i ← H_i | H_{i+1};
4:      i ← i+2;
5:  //Amend 101 and 1001 patterns
6:  C_1 ← E_1;      //Initialization
7:  C_m ← E_m;
8:  C_2 ← (E_1 Ē_2 E_3) | (E_1 Ē_2 Ē_3 E_4);
9:  C_{m-1} ← (E_{m-2} Ē_{m-1} E_m) | (E_{m-3} Ē_{m-2} Ē_{m-1} E_m);
10: for i ← 3 to m-2 do
11:   if E_i ==1 then C_i ← E_i;
12:   else C_i ← (E_{i-1} Ē_i E_{i+1}) | (E_{i-2} Ē_{i-1} Ē_i E_{i+1}) |
                                          (E_{i-1} Ē_i Ē_{i+1} E_{i+2});
13: return C;
```

5.4 Discussion and Novelty

GateKeeper is the only read mapping filter that takes advantage of the parallelism offered by FPGA architectures in order to expedite the alignment filtering process. GateKeeper supports both Hamming distance and edit distance in a fast and efficient way. Each GateKeeper filtering unit performs all operations defined in the GateKeeper algorithm. Table 5.1 summarizes the relative benefits gained by each of the aforementioned optimization methods over the best previous filter, SHD (E is the user-defined edit distance threshold and m is the read length). When a read matches the reference exactly, or with few substitutions, GateKeeper requires only $2m$ bitwise XOR operations, providing substantial speedup compared to SHD, which performs a much greater number of operations. However, this is not the only benefit we gain from our first proposed method (i.e., Fast Approximate String Matching).

Table 5.1: Overall benefits of GateKeeper over SHD in terms of number of operations performed.

# of operations for SHD:	# of operations for GateKeeper:
For Substitution & Indel Detection	**For Substitution Detection**
- $m(2E+1)$ bitwise XOR**.	- $2m$ bitwise XOR.
- $2E$ shift.	**For Indel Detection**
- $3(2E+1)$ shift.*	- $2m(2E+1)$ bitwise XOR.
- $4m(2E+1)$ bitwise OR.*	- $2E$ shift.
- $4(2E+1)$ packed shuffle.*	- $m(2E+1)$ bitwise OR.
	- $(2E+1)$ look-up table.*

* This operation is required for the amending process.
** E: edit distance threshold. m: read length.

As this method provides an accurate examination for alignments with only substitutions (i.e., no deletions or insertions), we can directly skip calculating their optimal alignment using the computationally expensive alignment algorithms. For more general cases such as deletions and insertions, GateKeeper still requires far fewer operations (as shown in Table 5.1) than the original SHD filter, due to the optimization methods outlined above. Our improvements over SHD help drastically reduce the execution time of the filtering process. The rejected alignments by our GateKeeper filter are not further examined by read alignment.

5.5 Summary

We introduce the first hardware acceleration system for alignment filtering, called GateKeeper. We develop both a hardware-acceleration-friendly filtering algorithm and a highly-parallel hardware accelerator design. GateKeeper is a standalone filter and can be integrated with any existing reference-based mapper. GateKeeper does not replace read alignment. GateKeeper should be followed by read alignment step, which precisely verifies the mappings that pass our filter and eliminates the falsely accepted ones.

Chapter 6

SLIDER: Fast and Accurate Hardware Pre-Alignment Filter

Our primary purpose is to reject incorrect mappings accurately and quickly such that we reduce the need for the computationally expensive alignment step. In this chapter, we propose the SLIDER algorithm to achieve highly accurate filtering. We then accelerate SLIDER by taking advantage of the capabilities and parallelism of FPGAs to achieve fast filtering operations. SLIDER's filtering strategy is inspired by our analytical study of SHD's filtering accuracy (see Chapter 3). We discuss the details of the SLIDER algorithm next.

6.1 Overview

The key filtering strategy of SLIDER is inspired by the pigeonhole principle, which states that if E items are distributed into $E+1$ boxes, then one or more boxes would be empty. In the context of pre-alignment filtering, this principle provides the following key observation. If two sequences differ by E edits, then they should share at least a single identical subsequence (i.e., free of edits) among $E+1$ non-overlapping subsequences, where E is the edit distance threshold.

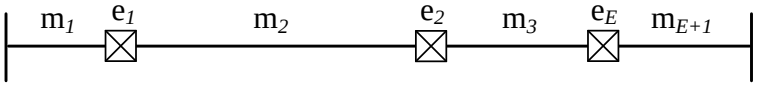

Figure 6.1: Random edit distribution in a read sequence. The edits (e_1, e_2, ..., e_E) act as dividers resulting in several identical subsequences (m_1, m_2, ..., m_{E+1}) between the read and the reference.

This is due to the fact that the E edits would result in dividing the alignment into $E+1$ segments of exact matches in accordance with their correspondences in the reference, as illustrated in Figure 6.1. SLIDER aims at identifying these $E+1$ segments of exact matches without calculating the optimal alignment. Next, we discuss the details of SLIDER.

6.2 Methods

Given reference sequence $A[1...m]$, read sequence $B[1...m]$, and an edit distance threshold E, the SLIDER algorithm finds out whether or not the total length of all non-overlapping identical subsequences of B in A is less than m-E (without knowing the actual edit distance) in three main steps. (1) The first step is to construct what we call a *neighborhood map* that visualizes the pairwise matches and mismatches between two sequences given an edit distance threshold of E characters. (2) The second step is to find all the non-overlapping diagonally-consecutive matches in the neighborhood map using a sliding window approach. (3) The last step is to make a decision (i.e., it accepts or rejects the given sequence pair) based on the length of the found matches. If the length of the found matches is small, then SLIDER rejects the input sequence pair.

6.2.1 Method 1: Building the Neighborhood Map

The neighborhood map, N, is a binary m by m matrix, where m is the read length. It represents the comparison result of each character of B with its corresponding character of A and the nearest neighbor characters. The comparison with the nearest neighbor characters is necessary for tolerating indels. An inserted character causes all the trailing bases to be shifted to the right-hand side. But a deleted character causes all the trailing bases to be shifted to the left-hand side. As we do not have prior knowledge about whether there exist insertion, or deletion, or substitution events (or a combination of them), we need to test for every possible case in our algorithm. Given i and j (where $1 \leq i \leq m$ and $i\text{-}E \leq j \leq i+E$), the entry $N[i, j]$ of the neighborhood map can be calculated as follows:

$$N[i,j] = \begin{cases} 0 & A[i] = B[j] \\ 1 & A[i] \neq B[j] \end{cases} \tag{6.1}$$

We present in Figure 6.2 an example of a neighborhood map for two sequences, where sequence B differs from sequence A by three edits. The entry $N[i,j]$ is set to 0 if the i^{th} character of the read sequence matches the j^{th} character of the reference sequence. Otherwise, it is set to 1. The way we build our neighborhood map ensures that computing each of its entries is independent of every other and can be computed all at once in a parallel fashion. Hence, our neighborhood map is well suited for highly parallel computing platforms [69, 121]. Notice that in sequence alignment algorithms, computing each entry of the dynamic programming matrix depends on the values of the immediate left, upper left, and upper entries of its own. Different from *"dot plot"* or *"dot matrix"* (visual representation of the similarities between two closely similar genomic sequences) that is used in FASTA/FASTP [122], our neighborhood map computes *only* some necessary diagonals near the main diagonal of the matrix (e.g., for E=3, SLIDER computes only $2E$+1 diagonal vectors of the neighborhood map).

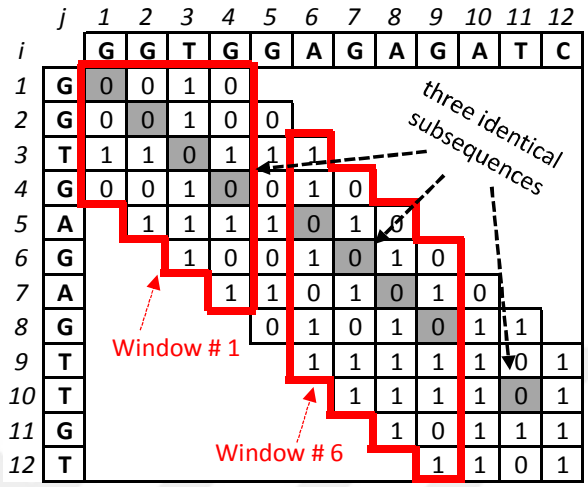

Figure 6.2: Neighborhood map (N), for reference sequence A = GGTGGAGAGATC, and read sequence B = GGTGAGAGTTGT for E=3. The three identical subsequences (i.e., GGTG, AGAG, and T) are highlighted in gray. We use a search window of size 4 columns with a step size of a single column.

6.2.2 Method 2: Identifying the Diagonally-Consecutive Matches

With the existence of E edits, there are *at most* E+1 non-overlapping segments of pairwise matches (based on the pigeonhole principle) shared between a pair of sequences. Each non-overlapping matching segment (or identical subsequence) is represented as a streak of diagonally-consecutive zeros in the neighborhood map. The accuracy of finding these subsequences is crucial for the overall filtering accuracy, as the filtering decision is solely made based on their total length. As we illustrate in Figure 6.2, SLIDER uses a search space of 4 columns wide. It finds the longest segment of diagonally-consecutive zeros within the search space (search window) and stores the segment in an additional bit-vector (we call it SLIDER bit-vector) at its corresponding location. SLIDER uses the stored segment to calculate the total length of all the non-overlapping segments found in the neighborhood map. The total length of these segments reveals the approximate edit distance. In order to locate the other E non-overlapping segments, SLIDER slides the search window by a single column towards the last bottom right entry of the neighborhood map.

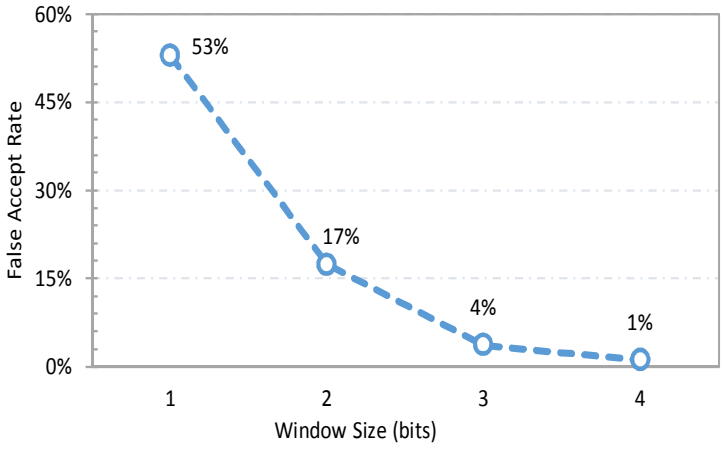

Figure 6.3: The effect of the window size on the rate of the falsely accepted sequences (i.e., dissimilar sequences that are considered as similar ones by SLIDER filter). We observe that a window width of 4 columns provides the highest accuracy. We also observe that as window size increases beyond 4 columns, more similar sequences are rejected by SLIDER, which should be avoided.

As we do not have prior knowledge about the length and the location of each segment of the diagonally-consecutive matches, SLIDER searches for the non-overlapping segments at each column of the neighborhood map. This maximizes the overlaps between the search windows and ensures finding all the non-overlapping segments. The reason behind the selection of the window size is due to the minimal possible length of the identical subsequence that is a single match located between two mismatches (i.e., '101'). However, we observe that the neighborhood map can contain also short streaks of diagonally-consecutive zeros that are not part of the desired alignment. In Figure 6.2, the bit pattern '101' exists once as a part of the correct alignment. However, the same pattern appears also about six times in other different locations that are not included in the correct alignment. To exclude this case and improve the accuracy of finding the diagonally-consecutive matches, we increase the length of the segment to be examined to four bits. We also experimentally evaluate different window sizes in Figure 6.3. We find that a window size of 4 columns provides the highest filtering accuracy without falsely rejecting similar sequences.

Figure 6.4: An example of applying SLIDER filtering algorithm to a sequence pair, where the edit distance threshold is set to 4. We present the content of the neighborhood map along with the SLIDER bit-vector. We apply the SLIDER algorithm starting from the leftmost column towards the rightmost column.

Meanwhile, we always want to find the maximum possible number of matches between two given sequences in order to avoid overestimating the total number of edits and eventually preserving all similar sequences. SLIDER filter examines the 2E+1 diagonal bit-vectors of the neighborhood map and finds the 4-bit subsequence that has the largest number of zeros within the search window. Once found, our filter examines if the selected subsequence maximizes the number of zeros in the SLIDER bit-vector at the same corresponding location. If so, then the selected 4-bit subsequence is stored in the SLIDER bit-vector at the same corresponding location. Otherwise, the search window slides one step ahead and repeat the previous steps until it reaches the last bottom right entry in the neighborhood map. In case of two diagonal vectors have the same number of diagonally-consecutive zeros, SLIDER breaks the ties by selecting the one that has a leading zero. This guarantees to always maximize the number of matches as the first bit (i.e., the leading zero) of the selected segment is not replaced during the next search window. We present an example of applying SLIDER filtering algorithm in Figure 6.4.

6.2.3 Method 3: Filtering Out Dissimilar Sequences

The last step of SLIDER is to calculate the total length of the found identical subsequences once after reaching the last bottom right entry of the neighborhood map.

The found identical subsequences are represented as '0's in the SLIDER bit-vector. Thus, SLIDER counts the occurrence of '0's in the SLIDER bit-vector. If their number is less than m-E then the two given sequences are rejected. Otherwise, the two sequences are considered similar within the allowed edit distance threshold and the optimal alignment can be generated using sophisticated alignment algorithms.

6.3 Analysis of SLIDER algorithm

SLIDER filter does not filter out similar sequences; hence, it provides zero false reject rate. The reason behind that is the way we find the identical subsequences. We always look for the subsequences that has the largest number of zeros, such that we maximize the number of matches and minimize number of edits that cause the division of one long identical sequence into shorter subsequences. This also allows for a very small portion of dissimilar sequences to pass. Next, we analyze the computational complexity (i.e., asymptotic run time and space complexity) of our SLIDER filter. SLIDER filter divides the problem of finding the identical subsequences into at most m subproblems, as described in Algorithm 6.1. Each subproblem examines each of the $2E$+1 bit-vectors and finds the 4-bit subsequence that has the largest number of zeros within the sliding window. Once found, SLIDER filter also compares the found subsequence with its corresponding subsequence of the SLIDER bit-vector. Now, let c be a constant representing the run time of examining each 4 bits of each bit-vector. Then the time complexity of the SLIDER algorithm is as follows:

$$T_{SLIDER}(m) = c \,.\, m \,.\, (2E + 1) \tag{6.2}$$

This demonstrates that the SLIDER algorithm runs in linear time with respect to the sequence length and edit distance threshold. SLIDER algorithm maintains $2E$+1 diagonal bit-vectors and an additional auxiliary bit-vector (i.e., SLIDER bit-vector) for each two given sequences. The space complexity of the SLIDER algorithm is as follows:

$$D_{SLIDER}(m) = m \,.\, (2E + 2) \tag{6.3}$$

Hence, the SLIDER algorithm requires linear space with respect to the read length and edit distance threshold. Next, we outline the hardware implementation details of SLIDER filter.

Algorithm 6.1: SLIDER

```
Input: Seq#1, Seq#2, Edit distance threshold (E).
Output: 1 (Similar/Accepted) / 0 (Dissimilar/Rejected).
Pseudocode:
 1: m ← length(Seq#1);
 2: // Build Neighborhood map (N)
 3: for i ← 1 to m do
 4:     for j ← i-E to i+E do
 5:         if Seq#1[i] == Seq#2[j] then
 6:             N[i,j] ← 0;
 7:         else N[i,j]← 1;
 8: // Sliding window search, function CZ() returns # of zeros
 9: for i ← 1 to m do SLIDER[i] ← 1;
10: for i ← 1 to m do
11:     for j ← 1 to E do
12:         // Compare with upper diagonal and lower diagonal
13:         if CZ(N[i+j:i+3+j,i:i+3]) > CZ(N[i:i+3,i+j:i+3+j]) then
14:             Z ← N[i+j:i+3+j,i:i+3];
15:         else if CZ(N[i+j:i+3+j,i:i+3]) == CZ(N[i:i+3,i+j:i+3+j]) then
16:             if N[i+j,i]==0 then Z ← N[i+j:i+3+j,i:i+3];
17:             else if N[i,i+j]==0 then Z ← N[i:i+3,i+j:i+3+j];
18:         else Z ← N[i:i+3,i+j:i+3+j];
19:     // Compare Z with main diagonal and SLIDER bit-vector
20:     if CZ(N[i:i+3,i:i+3]) > CZ(Z) then  Z ← N[i:i+3,i:i+3];
21:     if CZ(Z) > CZ(SLIDER[i:i+3]) then
22:         SLIDER[i:i+3] ← Z;
23: if CZ(SLIDER) ≥ m-E then return 1;
24: else return 0;
```

6.4 Discussion

To make the best use out of the available resources in the FPGA chip, our algorithms should utilize the FPGA supported operations such as bitwise operations.

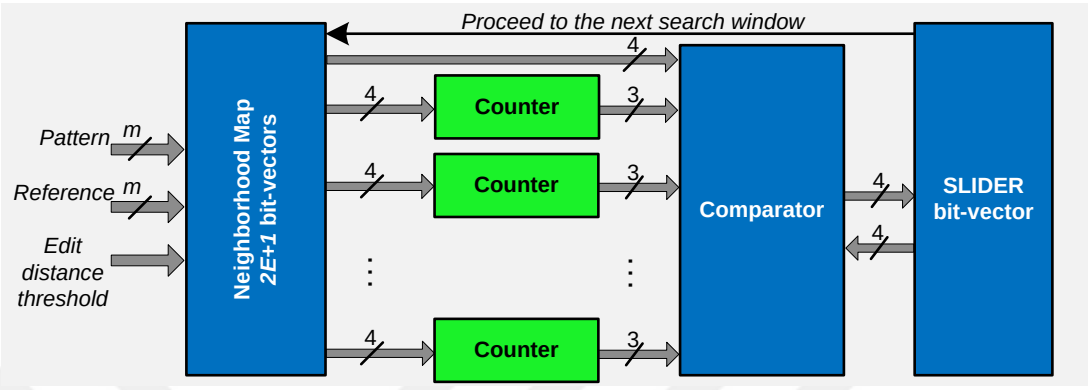

Figure 6.5: A block diagram of the sliding window scheme implemented in FPGA for a single filtering unit.

To build the neighborhood map on the FPGA, we observe that the main diagonal can be implemented using a bitwise XOR operation between the two given sequences. The upper E diagonals can be implemented by gradually shifting the read sequence (B) to the right-hand direction and then perform bitwise XOR with the reference sequence (A). This allows each character of B to be compared with the right-hand neighbor characters of its corresponding character of A. The lower E diagonals can be implemented in a similar way to the upper E diagonals, but the shift operation is performed to the left-hand direction. Likewise, this ensures the comparison between each character of B and the left-hand neighbor characters of its corresponding character of A. The vacant bits are filled with ones.

The second step of the SLIDER algorithm is identifying the diagonally-consecutive matches. This key step involves finding the 4-bit subsequence that has the largest number of zeros. For each search window, there are 2E+1 diagonal bit-vectors and an additional SLIDER bit-vector. To enable the computation to be performed in a parallel fashion, we build 2E+2 counters. As presented in Figure 6.5, each counter counts the number of zeros in a single bit-vector. The counter takes four bits as input and generate three bits that represents the number of zeros within the window. Each counter requires three 4-input LUTs. In total, we need 6E+6 4-input LUTs to build a single search window.

All bits of the counter output are generated at the same time, as the propagation delay through an FPGA look-up table is independent of the implemented function [120]. The comparator is responsible for selecting the 4-bit subsequence that maximizes the number of consecutive matches based on the output of each counter and the SLIDER bit-vector. Finally, the selected 4-bit subsequence is then stored in the SLIDER bit-vector at the same corresponding location.

6.5 Summary

We propose SLIDER, a highly parallel and accurate pre-alignment filter designed on a specialized hardware platform. The first key idea of our proposed pre-alignment filter is to provide high filtering accuracy by correctly detecting all identical subsequences shared between two given subsequences. This way leads to address the first two causes of filtering inaccuracy in SHD (i.e., random zeros and conservative counting). The second key idea is to avoid the filtering inaccuracy caused by the leading and trailing zeros (as we discuss in Chapter 3). SLIDER replaces the vacant bits that result from shifting the read with ones.

Chapter 7

MAGNET: Accurate Hardware Pre-Alignment Filter

In this chapter, we introduce MAGNET, a pre-alignment filtering algorithm to achieve highly accurate filtering. MAGNET is a filtering heuristic that aims at finding all non-overlapping long streaks of consecutive zeros in the neighborhood map using a divide-and-conquer approach. We discuss the details of MAGNET algorithm next.

7.1 Overview

MAGNET uses a divide-and-conquer technique to find all the $E+1$ identical subsequences, if any, and summing up their length. By calculating their total length, we can estimate the total number of edits between the two given sequences. If the total length of the $E+1$ identical subsequences is less than m-E, then there exist more identical subsequences than $E+1$ that are associated with more edits than allowed. If so, then MAGNET rejects the two given sequences without performing the alignment step. The filtering strategy of MAGNET makes three observations based on the pigeonhole principle to examine each mapping accurately.

1. Given that the user-defined edit distance threshold, E, is usually less than 5% of the read length (m) [27, 7, 113, 62], the identical subsequence is usually long and ranges from a single pairwise-match to m pairwise-matches long.

2. The length of the longest identical subsequence is strictly not less than $\lceil((m - E))/((E + 1))\rceil$ and can be at most m pairwise-matches long (i.e., equivalent to the sequence length). The upper bound is trivial and holds when the alignment is free of edits. The lower bound equality occurs when all edits are equispaced and all E+1 subsequences are of the same length.

3. We observe that the E+1 identical subsequences that are part of the correct alignment are always non-overlapping. We also observe that the $2E$+1 diagonal bit-vectors of the neighborhood map contain other identical subsequences that are always short. This raises a fundamental question about whether the E+1 identical subsequences need to be strictly non-overlapping or only long and not necessarily non-overlapping. Next, we investigate both cases by providing two algorithms for finding the longest identical subsequences.

7.2 Methods

MAGNET pre-alignment filter identifies the incorrect mappings, without calculating the optimal alignment, in three main steps. (1) It first constructs the neighborhood map (described in Chapter 6). (2) It then identifies all non-overlapping identical subsequences in the neighborhood map using a divide-and-conquer approach. (3) And finally it makes a decision (it accepts or rejects the given sequences) based on the length of the found identical subsequences.

7.2.1 Method 1: Building the Neighborhood Map

The first step of MAGNET algorithm is building our binary m by m neighborhood map that we describe in Chapter 6. Similarly with SLIDER filter, MAGNET filter starts with building the $2E+1$ diagonal bit-vectors of the neighborhood map for the two given sequences. We use the neighborhood map to represent all the pairwise matches between the read and the reference sequences. The neighborhood map considers also the presence of substitutions, insertions, and deletions. Next step is to find the consecutive matches that are part of the correct alignment.

7.2.2 Method 2: Identifying the $E+1$ Identical Subsequences

There are two methods to identify the identical subsequences. Either to find the non-overlapping subsequences of consecutive zeros or find the $E+1$ top longest subsequences of consecutive zeros. We describe both methods and show which one is more effective.

7.2.2.1 Identifying $E+1$ non-overlapping subsequences

Finding the $E+1$ non-overlapping subsequences in the neighborhood map involves three main steps.

1. **Extraction.** Each diagonal bit-vector nominates its local longest subsequence of consecutive zeros. Among all nominated subsequences, a single subsequence is selected as a global longest subsequence based on its length. (Once found, MAGNET evaluates if its length is less than is strictly not less than $\lceil((m-E))/((E+1))\rceil$, then the two sequences contains more edits than allowed, which cause the identical subsequences to be shorter (i.e.,

each edit results in dividing the sequence pair into more identical subsequences). If so, then the two sequences are rejected. Otherwise, MAGNET stores its length to be used towards calculating the total length of all $E+1$ identical subsequences.

2. **Encapsulation.** The next step is essential to preserve the original edit (or edits) that causes a single identical sequence to be divided into smaller subsequences. MAGNET penalizes the found subsequence by two edits (one for each side). This is achieved by excluding from the search space of all bit-vectors the indices of the found subsequence in addition to the index of the surrounding single bit from both left and right sides.

3. **Divide-and-Conquer Recursion.** In order to locate the other E non-overlapping subsequences, MAGNET applies a divide-and-conquer technique where we decompose the problem of finding the non-overlapping identical subsequences into two subproblems. While, the first subproblem focuses on finding the next long subsequence that is located on the right-hand side of the previously found subsequence in the first extraction step, the second subproblem focuses on the other side of the found subsequence. Each subproblem is solved by recursively repeating all the three steps mentioned above, but without evaluating again the length of the longest subsequence. MAGNET applies two early termination methods that aim at reducing the execution time of the filter. The first method is evaluating the length of the longest subsequence in the first recursion call. The second method is limiting the number of the subsequences to be found to at most $E+1$, regardless their actual number for each two sequences.

7.2.2.2 Identifying top $E+1$ longest subsequences

Alternatively, MAGNET can be changed to find only top $E+1$ longest subsequences without the restriction of being non-overlapping ones. This can be achieved by maintaining a binary max-heap priority queue [123], where it stores the length of each of the $E+1$ subsequences from each mask.

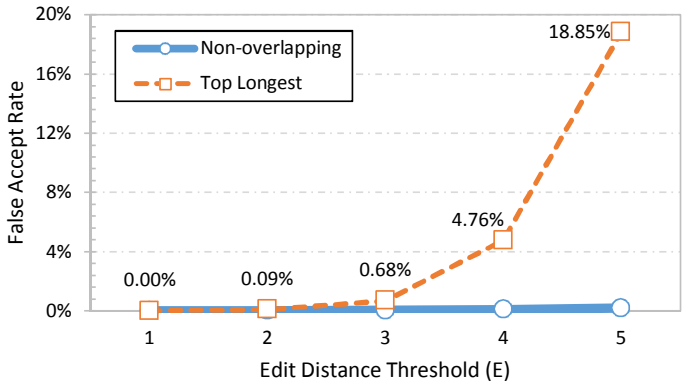

Figure 7.1: The false accept rate of MAGNET using two different algorithms for identifying the identical subsequences. We observe that finding the E+1 non-overlapping identical subsequences leads to a significant reduction in the incorrectly accepted sequences compared to finding the top E+1 longest identical subsequences.

In total, the queue stores up to $(E+1).(2E+1)$ elements. The length of the top longest subsequence is always stored at the root of the heap (the heap property). We need to extract the root of the heap structure E+1 times in order to find the total length of the top E+1 longest subsequences. We evaluate both algorithms for finding the longest identical subsequences in Figure 7.1. We observe that the accuracy of finding top E+1 longest subsequences degrades exponentially as the edit distance threshold increases. We also observe that the accuracy of finding the non-overlapping subsequences remains almost linear and provides considerably more accurate filtering. Thus, we consider identifying only non-overlapping subsequences throughout the following sections.

7.2.3 Method 3: Filtering Out Dissimilar Sequences

The last step of MAGNET is to decide if the mapping is potentially correct and needs to be examined by read alignment. Once after the termination, if the total length of all found identical subsequences is less than m-E then the two sequences are rejected. Otherwise, they are considered to be similar and the alignment can be measured using sophisticated alignment algorithms.

Figure 7.2: An example of applying MAGNET filtering algorithm with an edit distance threshold of 4. MAGNET finds all the longest non-overlapping subsequences of consecutive zeros in the descending order of their length (as numbered in yellow).

We provide an example of applying MAGNET filtering algorithm in Figure 7.2.

7.3 Analysis of MAGNET algorithm

In this section, we analyze the asymptotic run time and space complexity of MAGNET algorithm. We provide the pseudocode of MAGNET in Algorithm 7.1. We also provide the divide-and-conquer algorithm that MAGNET uses in Algorithm 7.2. MAGNET applies a divide-and-conquer technique that divides the problem of finding the identical subsequences into two subproblems in each recursion call. In the first recursion call, the extracted identical subsequence is of length at least a=$\lceil((m\text{-}E))/((E+1))\rceil$ bases. This reduces the problem of finding the identical subsequences from m to at most m-a, which is further divided into two subproblems: a left subproblem and a right subproblem. For the sake of simplicity, we assume that the size of the left and the right subproblems decreases by a factor of b and c, respectively, as follows:

$$m = a + 2 + m/b + m/c \tag{7.1}$$

The addition of 2 bases is for the encapsulation bits added at each recursion call. Now, let $T_{MAGNET}(m)$ be the time complexity of MAGNET algorithm, for identifying non-overlapping subsequences.

Algorithm 7.1: MAGNET

Input: *Seq1, Seq2, Edit distance threshold (E).*
Output: *1 (Similar/Accepted) / 0 (Dissimilar/Rejected).*
Pseudocode:

```
1: m ← length(Seq1);
2: // Build Neighborhood map (N)
3: for i ← 1 to m do
4:     for j ← i-E to i+E do
5:         if Seq1[i] = Seq2[j] then
6:             N[i,j] ← 0;
7:         else
8:             N[i,j] ← 1;
9: for i ← 1 to m do
10:    MAGNET[i] ← 1;
11: [MAGNET, calls] ← EXEN(N, 1, m, E, MAGNET, 1);
12: // Function CZ() returns number of zeros
13: if CZ(MAGNET) ≥ m-E then return 1; else return 0;
```

If it takes O(km) time to find the global longest subsequence and divide the problem into two subproblems, where $k = 2E+1$ is the number of bit-vectors, we get the following recurrence equation:

$$T_{MAGNET}(m) = T_{MAGNET}(m/b) + T_{MAGNET}(m/c) + O(km) \tag{7.2}$$

Given that the early termination condition of MAGNET algorithm restricts the recursion depth as follows:

$$Recursion\ tree\ depth = \lceil log_2(E+1) \rceil - 1 \tag{7.3}$$

Solving the recurrence in equation (7.2) using equation (7.1) and equation (7.3) by applying the recursion-tree method provides a loose upper-bound on the time complexity as follows:

$$T_{MAGNET}(m) = O(km)\ .\ \sum_{x=0}^{\lceil log_2(E+1) \rceil - 1} \left(\frac{1}{c} + \frac{1}{b}\right)^x$$

$$T_{MAGNET}(m) \approx O(fkm) \tag{7.4}$$

Where f is a fraction number satisfies the following range: $1 \leq f < 2$. This in turn demonstrates that the MAGNET algorithm runs in a linear time with respect to the sequence length and edit distance threshold and hence it is computationally inexpensive. The space complexity of MAGNET algorithm is as follows:

$$D_{MAGNET}(m) = D_{MAGNET}(m/b) + D_{MAGNET}(m/c) + O(km + m)$$

$$D_{MAGNET}(m) \approx O(fkm + fm) \tag{7.5}$$

Hence, MAGNET algorithm requires a linear space with respect to the read length and edit distance threshold. Next, we outline the hardware implementation details of MAGNET filter.

7.4 Discussion

In this section, we outline the challenges that are encountered in implementing MAGNET filter to be used in our accelerator design. Implementing MAGNET algorithm is challenging due to the random location and variable length of each of the E+1 identical subsequences. The Verilog-2011 imposes two challenges on our architecture as it does not support variable-size partial selection and indexing of a group of bits from a vector [124]. In particular, the first challenge lies in excluding the extracted identical subsequence along with its encapsulation bits from the search space of the next recursion call. The second challenge lies in dividing the problem into two subproblems, each of which has an unknown size at design time. To address these limitations and tackle the two design challenges, we keep the problem size fixed and at each recursion call. We exclude the found longest subsequence from the search space by amending all bits of all $2E$+1 bit-vectors that are located within the indices (locations) of the encapsulation bits to '1's. This ensures to omit the found longest subsequence and all its corresponding locations in the other bit-vectors from the following recursion calls.

Algorithm 7.2: EXEN function

Function: EXEN() extracts the top longest subsequence of consecutive zeros and generate two subproblems.
Input: Neighborhood map (*N*), start index (*SI*), end index (*EI*), *E*, MAGNET bit-vector, number of recursion calls.
Output: updated MAGNET bit-vector, updated number of calls.
Pseudocode:

```
1: // Early termination condition
2: if (SI ≤ EI and calls ≤ E+1) then
3:     // Function CCZ() returns number and indices of longest
4:     // subsequence of diagonally consecutive zeros
5:     for j ← 1 to E do //Extraction
6:         [X,s1,e1] ← CCZ(N[SI+j,SI],EI); // Lower diagonal
7:         [Y,s2,e2] ← CCZ(N[SI,SI+j],EI); // Upper diagonal
8:         if X > Y then
9:             s ← s1; e ← e1;
10:        else
11:            s ← s2; e ← e2;
12:    [X,s1,e1] ← CCZ(N[SI,SI],EI);
13:    if X > (e-s+1) then
14:        s ← s1; e ← e1;
15:    // Early termination condition (only in first call)
16:    if (calls=1 and (e-s+1)<⌈(m − E)/(E + 1)⌉) then
17:        return [MAGNET, 0];
18:    // Left subproblem with encapsulation
19:    [MAGNET, calls] ← EXEN(N,SI, s-2, E, MAGNET, calls+1);
20:    // Right subproblem with encapsulation
21:    [MAGNET, calls] ← EXEN(N,e+2,EI, E,MAGNET, calls+1);
22:    return [MAGNET, calls];
23: else return [MAGNET, calls-1];
```

7.5 Summary

We introduce MAGNET, a new filtering strategy that remarkably improves the accuracy of pre-alignment filtering and provides a minimal number of falsely accepted mappings.

MAGNET gets rid of the first three causes of filtering inaccuracy that we observed in SHD [7] (see Chapter 3). We believe that MAGNET is the most accurate pre-alignment filter in literature today but this comes at the expense of hardware implementation challenges.

Chapter 8

SneakySnake: Fast, Accurate, and Cost-Effective Filter

In this chapter, we address the issue of the long execution time of read alignment using a different approach. The use of specialized hardware chips can yield significant performance improvements. However, the main drawbacks are the higher design effort, the limited data throughput, and the necessity of algorithm redesign and tailored implementation to fully leverage the FPGA architecture. To tackle these challenges and obviate these difficulties, we introduce fast, accurate, and yet cost-effective pre-alignment filter. We call it SneakySnake. It leverages the today's general purpose processor (i.e., CPU), which is largely available to bioinformaticians at no additional cost. Beside the CPU implementation of SneakySnake, we also provide an efficient hardware architecture to further accelerate the computations of the SneakySnake algorithm with high parallelism.

8.1 Overview

The key idea of SneakySnake filter is that it needs to know only if two sequences are dissimilar by more than the edit distance threshold or not.

This fact leads to the following question. **Can one approximate the edit distance between two sequences much faster than calculating the edit distance?** Given reference sequence $A[1 \dots m]$, read sequence $B[1 \dots m]$, and an edit distance threshold E, SneakySnake calculates the approximate edit distance between A and B and then checks if it is greater than the edit distance threshold then the two sequences A and B are rejected. Otherwise, the two sequences A and B are accepted by SneakySnake and the optimal alignment is calculated. The approximate edit distance calculated by SneakySnake should always be less than or equal to the edit distance threshold as long as the actual edit distance does not exceed the edit distance threshold.

8.2 The Sneaky Snake Problem

We convert the approximate edit distance problem to as a restricted optimal path finding problem in a grid graph. We call this the Sneaky Snake Problem. It can be summarized as a traversal problem in a special grid graph, where a snake travels in the grid graph with the presence of randomly distributed obstacles. The goal is to find the path that connects the origin and the destination points with the minimal number of obstacles along the path.

We describe the grid graph of the Sneaky Snake Problem as follows:

- There is a two dimensional m by m grid graph, where m is the read length. The grid cells are aligned in rows and columns and all cells have the same cost. The snake wants to travel through the grid from the top left corner towards the bottom right corner of the grid. There are $E+1$ entrances that are next to the cells of the first column and similarly, there are $E+1$ exits that are located next to the cells of the last column.
- The snake is allowed to travel through dedicated pipes. Each pipe is represented as a stretch of diagonally consecutive pairwise matches. We define the path in this problem as a sequence of diagonal grid cells.

In other words, the movement through the vertical cells is not counted towards the length of the path. The length of the path is simply the total number of diagonal cells along the path.

- The snake is only able to switch between the pipes after skipping an obstacle (i.e., end of the pipe). The obstacle represents a pairwise mismatch. We model the obstacle as a solid black grid cell. Traveling across an obstacle requires the snake to diagonally move one step ahead, and hence the obstacle cell is counted towards the total length of the path. The snake is allowed to avoid only E obstacles by transferring to another pipe or jumping over the obstacle within the same diagonal vector. This can be seen as the number of E tries required to arrive the destination.

The general goal of this problem is to find a path in the grid graph with the minimum number of obstacles. The path starts at the first leftmost cell and travels out of the grid while exiting onto a destination. SneakySnake filtering algorithm can approximate the edit distance using two different approaches. While one of them relies on what we call *weighted neighborhood maze*, the other approach relies on *unweighted neighborhood maze*. Both approaches involve three main steps.

8.3 Weighted Maze Methods

Next, we describe the three steps of the first approach. (1) It constructs the weighted neighborhood maze. (2) It identifies the optimal travel path that has the least number of obstacles. (3) The mapping is accepted if and only if the snake survives the grid maze.

8.3.1 Method 1: Building the Weighted Neighborhood Maze

The first step in the SneakySnake algorithm is to build the m by m weighted neighborhood maze (WN). It is a modified version of our original neighborhood maze (see Chapter 6), where we change the meaning of the content of each cell. First, we change the value of the grid cell that represents a pairwise match from zero to the total number of the consecutive zeros in its lower right neighbors within the same diagonal vector. Second, we change the value of the grid cell that represents a pairwise mismatch from one to zero (modeled as a solid black cell). Given i and j (where $1 \leq i \leq m$ and $i\text{-}E \leq j \leq i\text{+}E$), the entry $WN[i, j]$ of the weighted neighborhood maze can be calculated as follows:

$$WN[i,j] = \begin{cases} WN[i+1, j+1] + 1 & A[i] = B[j] \\ 0 & A[i] \neq B[j] \end{cases} \tag{8.1}$$

Computing each cell depends on its immediate lower right cell. This restricts the order of the computations to be performed starting from the lower right corner towards the upper left corner. We present in Figure 8.1 an example of a weighted neighborhood maze for two sequences, where the sequence B differs from the sequence A by three edits (i.e., obstacles).

8.3.2 Method 2: Finding the Optimal Travel Path

The second step of the SneakySnake algorithm is to decide on which pipe the snake should travel through, using three main steps. (1) The snake essentially uses the precomputed weight of each cell in the weighted neighborhood maze to make the decision. This is the key reason behind filling the weighted neighborhood maze from the lower right corner to the upper left corner, while the snake travels from the upper left corner towards the lower right corner.

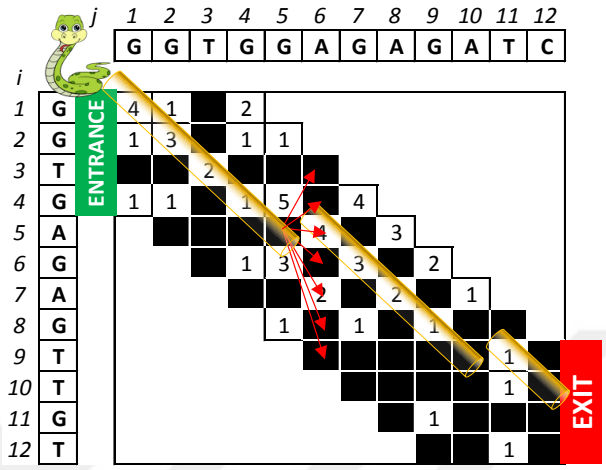

Figure 8.1: Weighted Neighborhood maze (WN), for reference sequence A = GGTGGAGAGATC, and read sequence B = GGTGAGAGTTGT for E=3. The snake traverses the path that is highlighted in yellow which includes three obstacles and three pipes, that represents three identical subsequences: GGTG, AGAG, and T

The snake examines the first cell of each diagonal vector in the first column of the grid and finds the cell with the largest weight. (2) If the snake is unable to decide on which pipe to follow (i.e., all cells of the first column are obstacles), the snake skips the current column and consumes one attempt out of the E tries. The snake then finds the maximum weight of the cells in the next column. If it finds more than one pipe with the same length, it always chooses the first one (starting from the upper diagonal). (3) Once found, it travels through the pipe until it faces an obstacle. It repeats these three steps until it reaches its destination or consumes all the E tries.

8.3.3 Method 3: Examining the Snake Survival

The last step is to check if the snake makes it through the grid maze or not. Given a limited number of tries (equals to E), if the snake arrives the destination, then there exists a path that has at most E obstacles.

In the context of approximate edit distance calculation, it means that there exists an alignment that has at most E edits. The SneakySnake algorithm considers this mapping as a correct one and accepts this mapping. Otherwise, it rejects the mapping without performing read alignment step.

8.4 Unweighted Maze Methods

Here, we explain the three steps of solving the Sneaky Snake Problem using the unweighted neighborhood maze approach. (1) It first constructs the unweighted neighborhood maze. (2) It then identifies the optimal travel path that has the least number of obstacles. (3) And finally the mapping is accepted if and only if the snake survives the grid maze without exceeding the limited number of tries.

8.4.1 Method 1: Building the Unweighted Neighborhood Maze

In the unweighted neighborhood maze, we introduce two changes to the way we build the neighborhood maze. First, each grid cell has no weight assigned to it. Second, we define a state for each grid cell, whereas a cell can be either in an available state or a blocked state. We set the grid cell to be in an available state if it represents a pairwise match. If the grid cell represents a pairwise mismatch, then we set its state to blocked. In this way, the unweighted neighborhood maze eliminates the data dependency between the grid cells that exists in the weighted neighborhood maze. Given i and j (where $1 \leq i \leq m$ and $i\text{-}E \leq j \leq i\text{+}E$), the state of the entry $UN[i, j]$ of the unweighted neighborhood maze can be chosen as follows:

$$UN[i,j] = \begin{cases} Available & A[i] = B[j] \\ Blocked & A[i] \neq B[j] \end{cases} \tag{8.2}$$

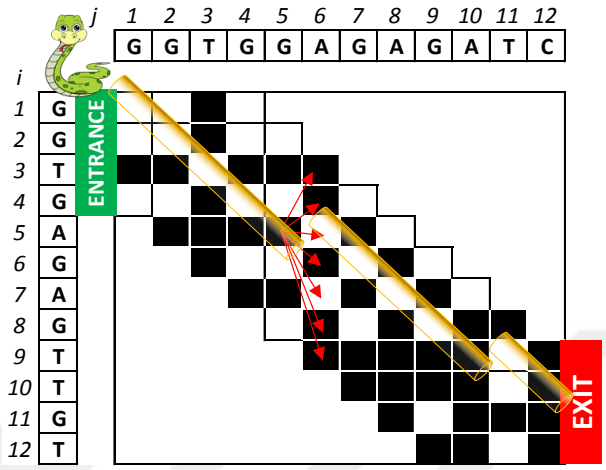

Figure 8.2: Unweighted neighborhood maze (UN), for reference sequence A = GGTGGAGAGATC, and read sequence B = GGTGAGAGTTGT for E=3. The snake traverses the path that is highlighted in yellow which includes three obstacles and three pipes, that represents three identical subsequences: GGTG, AGAG, and T.

We present in Figure 8.2 an example of an unweighted neighborhood maze for two sequences, where the sequence B differs from the sequence A by three edits. The state of the entry $UN[i,j]$ is set to available if the i^{th} character of the read sequence matches the j^{th} character of the reference sequence. Otherwise, it is set to blocked (highlighted in a black).

8.4.2 Method 2: Finding the Optimal Travel Path

The second step of the SneakySnake algorithm is to decide on which pipe the snake should consider, in three main steps. (1) As the snake has no prior information on the length of each pipe (as in the weighted neighborhood maze), it uses its telescoping lens to perform depth-first search (DFS) [125] each time it faces a blocked cell (i.e., obstacle). The DFS algorithm traverses the first upper diagonal vector to reach the maximum possible depth (i.e., until it arrives a blocked cell or the end of the vector). If the DFS algorithm reaches the exit of the maze, then SneakySnake terminates the DFS search.

The depth of each grid cell is equal to the value of its j index. It stores the maximum possible depth and backtracks to the starting point (i.e., the obstacle cell). It then continues traversing the next unsearched diagonal vector and repeats the previous steps. (2) The snake selects the path with the largest depth. If it is unable to find one (i.e., all cells of the first column are obstacles), it skips the current column and consumes one path transfer out of the E allowed transfers. (3) If it finds multiple equi-length pipes, it always chooses the first one (starting from the upper diagonal). Once found, it travels through the pipe until it faces another obstacle. It repeats these three steps until it reaches its destination or consumes all the E tries.

8.4.3 Method 3: Examining the Snake Survival

The last step is to accept the mapping based on whether the snake arrives its destination given a limited number of path transfers (equals to E). This means that there is an alignment for the two given sequences that has at most E edits. Otherwise, The SneakySnake algorithm rejects the mapping without performing read alignment step.

8.5 Analysis of SneakySnake Algorithm

In this section, we analyze the asymptotic run time and space complexity of the SneakySnake algorithm. The SneakySnake algorithm builds the weighted neighborhood maze by traversing through each vector starting from the lower right corner. It compares the corresponding characters of the two given sequences and if they match each other, then it updates the current cell with the result of summing up one and the value of the previous cell. Assuming it takes O(m) time to build one diagonal vector, where m is the read length. Then it takes O(km) to build the entire weighted maze, where $k = 2E+1$ is the number of the grid vectors.

Upon arriving an obstacle, the SneakySnake algorithm examines the weight of each cell following the obstacle cell and picks the cell with the largest weight as a new starting point. With the existence of E obstacles, this step is repeated at most E times. Each time it takes O(k) time to compare the weight of k cells. Thus, the upper-bound on the time complexity of SneakySnake using weighted neighborhood maze is given as follows:

$$T_{Weighted\_SneakySnake}(m) = O(Em + hk) \tag{8.3}$$

Where h is a number satisfies the following range: $0 \leq h \leq E$.

Using the unweighted neighborhood maze, the SneakSnake algorithm does not necessarily traverse all the cells of each diagonal vector (as in the weighted approach described above). Thus, its asymptotic run time is not determined. On the one hand, the lower-bound on the its time complexity is O(m), which is achieved when the DFS algorithm reaches the exist of the maze without facing any obstacle along the path. On the other hand, the loose upper-bound run time complexity is also equal to O(km+hk) when the DFS traverses through nearly the entire unweighted neighborhood maze. However, it is unrealistic to traverse the entire maze, as in this case the value of each and every cell of the entire maze should be equal to '0' (for example, when all characters of the two sequences are 'A's). If this is the case, then the DFS algorithm traverses only through the first upper diagonal and then get terminated. Thus, the run time complexity of SneakySnake with the unweighted neighborhood maze is given as follows:

$$O(m) \leq T_{Unweighted\_SneakySnake}(m) < O(km + hk) \tag{8.4}$$

Where h is a number satisfies the following range: $0 \leq h \leq E$. This in turn demonstrates that the unweighted neighborhood maze algorithm is asymptotically inexpensive compared to the weighted SneakySnake algorithm. Hence, we consider the unweighted SneakySnake algorithm for further analysis and evaluation. We provide the pseudocode of SneakySnake in Algorithm 8.1. While the weighted SneakySnake requires no additional auxiliary space, the weighted approach requires only storing the depth of at most $2E$+1 vectors.

Algorithm 8.1: SneakySnake

Input: *Seq1, Seq2, Edit distance threshold (E).*
Output: *1 (Similar/Accepted) / 0 (Dissimilar/Rejected).*
Pseudocode:

```
 1: m ← length(Seq1);
 2: // Build unweighted Neighborhood maze (UN)
 3: for i ← 1 to m do
 4:       for j ← i-E to i+E do
 5:             if Seq1[i] == Seq2[j] then
 6:                   N[i,j] ← 0;
 7:             else N[i,j]← 1;
 8: // Find Best Path with least number of obstacles
 9: i ← 1; e ← 0;
10: while i ≤ m and e ≤ E do
11:      LargestDepth = 0;
12:      for j ← 2E+1 to 1 do
13:           count ← 0;
14:           ii ← 0;
15:           //UN_Diagonal[diagonal id][cell index]
15:           while UN_Diagonal[j][i+ii] == 0 do
16:                Depth ← i;
17:                ii++;
18:           if (Depth>LargestDepth)
19:                LargestDepth ← Depth;
20:      i ← LargestDepth + 1;
21:      e ← e + 1;
22: // Examine the snake arrival
23: if e-1 ≤ E then return 1;
23: else return 0;
```

8.6 SneakySnake on an FPGA

We introduce an FPGA-friendly architecture for the unweighted SneakySnake algorithm. Achieving an efficient hardware architecture raises the following question: **Can one solve many small sub-problems of the Sneaky Snake Problem with a high parallelism by reducing the search space of the SneakySnake algorithm?**

The main idea behind the hardware architecture of the SneakySnake algorithm is to partition the unweighted neighborhood maze into small non-overlapping sub-matrices with a width of t columns (the height is always fixed to $2E+1$ rows) for some parameter t. We then apply the SneakySnake algorithm to each sub-matrix independently from the other sub-matrices. This results in three key benefits. First, downsizing the search space into a reasonably small sub-matrix with a known dimension at the design time limits the number of all possible solutions for that sub-matrix. This reduces the size of the look-up tables (LUTs) required to build the architecture and simplifies the overall design. Second, this approach helps to maintain a modular and scalable architecture that can be implemented for any read length and edit distance threshold. Third, all the smaller sub-problems can be solved independently and rapidly with a high parallelism. This reduces the execution time of the overall algorithm as the SneakySnake algorithm does not need to evaluate the entire path. However, these benefits come at the cost of accuracy degradation. The solution for each sub-problem is not necessarily part of the solution for the main problem (with the original size of m by m). Though it is guaranteed to always choose the path with the least obstacles within the search space, our hardware architecture can underestimate the number of obstacles found and thereby increase the false accept rate.

Next, we present the details of our hardware architecture. We choose the parameter t to be 8 columns. This results in partitioning the unweighted neighborhood maze of size 100 by 100 (or $2E+1$ by 100 computed cells) into 13 sub-matrices, each of size $2E+1$ by 8. Each row in the sub-matrix is part of the diagonal vector of the unweighted neighborhood maze. Each sub-matrix represents an individual Sneaky Snake Problem. Solving each problem requires determining the optimal path (with the least number of obstacles) along each sub-matrix using five main steps. (1) The first step is to perform the DFS search for finding the optimal path along the $2E+1$ rows of the sub-matrix. We implement the DFS algorithm on FPGA as a leading-zero counter (LZC). We use the LZC design proposed in [126]. It counts the number of consecutive zeros that appear in a n-bit input word before the first more significant bit that is equal to one.

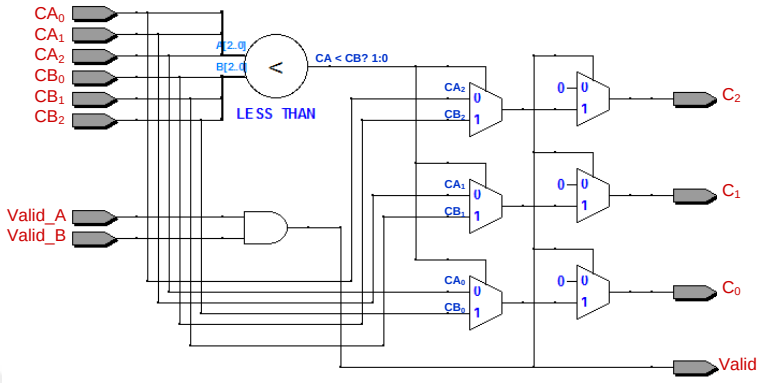

Figure 8.3: Proposed 4-bit LZC comparator.

It generates two output signals, the $\log_2$n bits of the leading-zero count C and a flag *Valid*, for an input word $A = A_{n-1}, A_{n-2}, \ldots, A_0$, where A_{n-1} is the most-significant bit. When all input bits are set to zero, the *Valid* flag is set to one. For other cases of input, the value of C represents the number of leading zeros. For the case of an 8-bit input operand, the two output signals of the LZC are given by:

$$\begin{aligned} \overline{C_2} &= A_7 + A_6 + A_5 + A_4 \\ \overline{C_1} &= A_7 + A_6 + \overline{A_5} \cdot \overline{A_4} \cdot (A_3 + A_2) \\ \overline{C_0} &= A_7 + \overline{A_6} \cdot A_5 + \overline{A_6} \cdot \overline{A_4} \cdot A_3 + \overline{A_6} \cdot \overline{A_4} \cdot \overline{A_2} \cdot A_1 \\ Valid &= A_7 + A_6 + A_5 + A_4 + A_3 + A_2 + A_1 + A_0 \end{aligned} \tag{8.5}$$

(2) The second step is to find the row that has the largest number of leading zeros. We build a hierarchical comparator structure with $\log_2(2E+1)$ levels. We use a single *LZC comparator* for comparing the number of leading zeros of two rows. The LZC comparator compares two numbers of leading zeros produced by two LZC circuits and passes the largest number as output signals without changing their values (maintaining the same meaning). We provide in Figure 8.3 the proposed architecture of the 4-bit LZC comparator. For an edit distance thresholds of 5 bp, we need 12 LZC comparators arranged into 4 levels (6, 3, 2, and 1 LZC comparators in each level, respectively). We provide the overall architecture of the 4-level LZC comparator tree including the 11 LZC block diagrams in Figure 8.4.

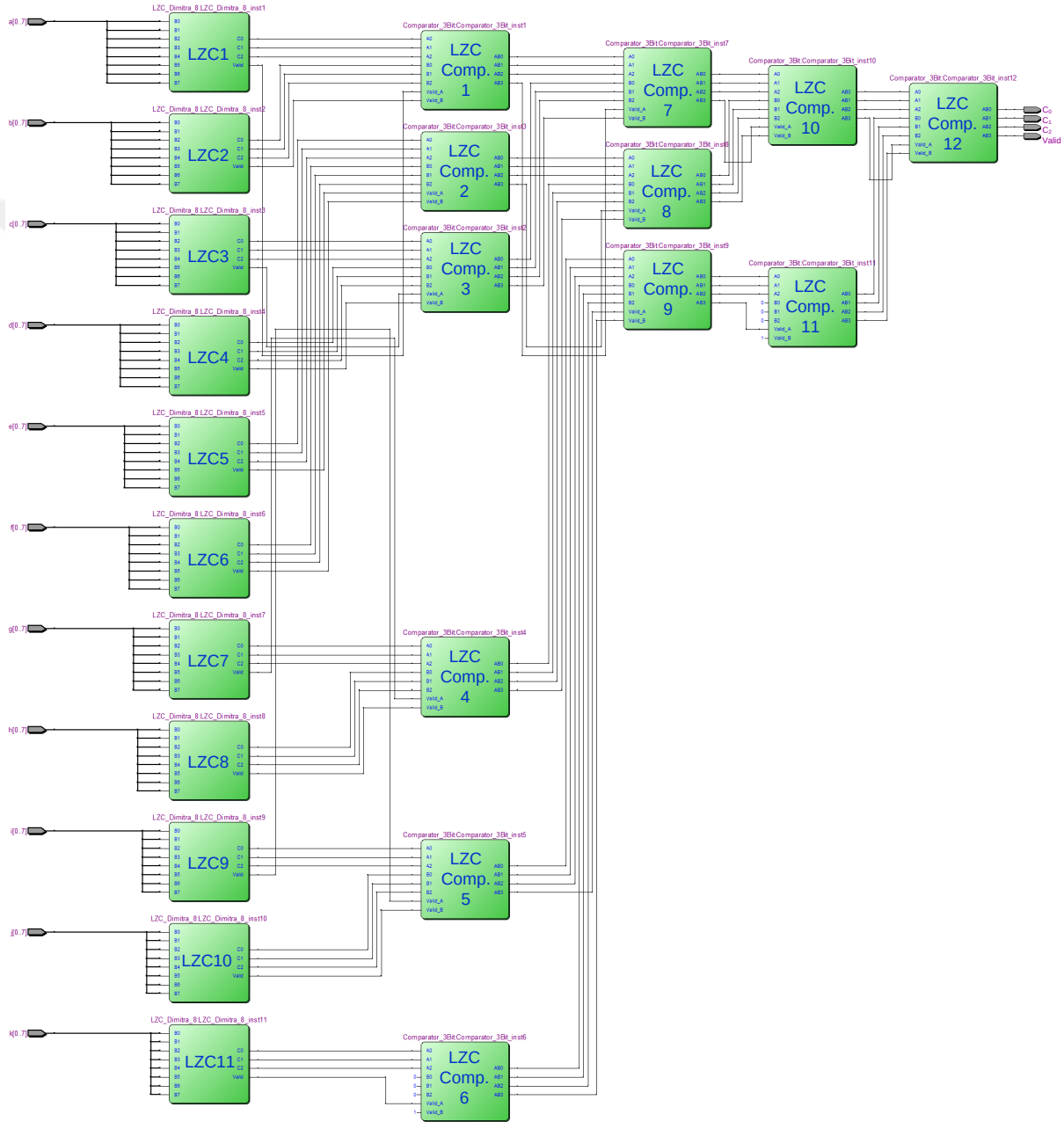

Figure 8.4: Block diagram of the 11 LZCs and the hierarchical LZC comparator tree for computing the largest number of leading zeros in 11 rows.

(3) The third step is to store the largest number of leading zeros and shift the bits of all rows to the right direction. As we implement the DFS search as a LZC circuit, we always need to start the DFS search space from the least significant bit of each row. The shift operation is necessary for deleting the leading zeros and allowing the snake to continue the process of finding the optimal path. We shift each row of the sub-matrix by x+1 bits, where x is the largest number of leading zeros found in step 2. This guarantees to exclude the found path from the next search round along with the edit, which divides the optimal path into shorter paths.

(4) The fourth step is to repeat the previous three steps in order to find the optimal path from the least significant bit all the way to the most significant bit. The number of replications needed depends on the desired accuracy of the SneakySnake algorithm. If our target is to find at most a single edit within each sub-matrix, then we need to build two replications for the steps described above. For example, let A be 00010000, where $t = 8$. The first replication computes the value of x as four zeros and updates the bits of A to 11111000. The second replication computes the value of x as three zeros and updates the bits of A to 11111111.

(5) The last step is to calculate the total number of obstacles along the entire optimal path from the least significant bit towards the most significant bit. We first find out the number of replications that produces at least a single leading zero (i.e., x¿0). If it equals the total number of replications (y), then the number of obstacles (edits) equals to the total number of replications included in the design. Otherwise, we compute the number of the obstacles as follows:

$$min(y, t - \sum_{o=1}^{y} x_o) \tag{8.6}$$

where y is the total number of replications involved in the design and x_o is the largest number of leading zeros produced by the replication of index o.

8.7 Discussion

We introduce the SneakySnake algorithm and hardware architecture in order to address the question that we raise earlier in this chapter about weather approximating the edit distance calculation can be faster than calculating the exact edit distance. The accuracy of the edit distance approximation is another concern. Our SneakySnake filter always underestimates the total number of edits. This is mainly due to the goal of the Sneaky Snake Problem that is the search for optimal path with the least number of obstacles (or edits). This concern raises two key questions. (1) **Can one improve the accuracy of edit distance approximation such that we achieve either a new optimal read aligner or a highly-accurate pre-alignment filter?** The SneakySnake algorithm can be slightly modified such that it considers a penalty on the selected path. If the snake selects the upper diagonals, then this means that the obstacle is basically a deleted character at the i^{th} index of the read sequence. Similarly, if the current path is at the main diagonal and the next selected path is at the second upper diagonal, this means that the obstacle is two deleted characters. The current implementation of SneakySnake considers this two deleted characters as a single edit. The accuracy improvement of SneakySnake is yet to be explored in this thesis.

The second question is (2) Similarly to the way we partition the search space of the SneakySnake algorithm, **Can one reduce the search space of exact edit distance algorithms such that only the necessary cells are computed?** We observe the fact that the exact edit distance algorithms explore a large area of the dynamic programming matrix (even with the banded matrix), which is unnecessary for highly dissimilar sequences. The edit distance is considered to be a non-additive distance measure [101]. This means that its calculations can not be distributed over concatenated subsequences of the long sequence. In other words, we can not divide the read-reference pair into shorter pairs and calculate the exact edit distance for each short read-reference pair individually (aiming at concurrently computing them) and then accumulate the results.

For example, take A = "CGCG", B = "GCGC" and observe that calculating the edit distance for each character of A with its corresponding character of B yields four edits, while the edit distance between A and B, each as a whole, is only 2 edits. This is clear from the way that its dynamic programming matrix is computed. The computations always depend on the prefixes of both the read and the reference sequences. As a workaround, we can compute the exact edit distance in an incremental approach. One can divide the dynamic programming matrix into sub matrices, where each smaller sub-matrix is fully covered by all the larger sub-matrices. Such that each sub-matrix UN_s strictly starts from the index i of value 1 up to some s, where the following equation holds $1 \leq i \leq s \leq m$ and $i\text{-}E \leq j \leq i\text{+}E$. We refer to this edit distance measure as ***a prefix edit distance***. In the next chapter, we evaluate all these scenarios in details.

8.8 Summary

In this chapter, we introduce the Sneaky Snake Problem, and we show how an approximate edit distance problem can be converted to an instance of the Sneaky Snake Problem. Subsequently, we propose a new pre-alignment filtering algorithm (we call it SneakySnake) that obviates the need for expensive specialized hardware. The solution we provide is cost-effective given a limited resources environment. Our algorithm does not exploit any SIMD-enabled CPU instructions or vendor-specific processor. This makes it superior and attractive. We also provide efficient and scalable hardware architecture along with several design optimizations for the SneakySnake algorithm. Finally, we discuss several optimizations and challenges of accelerating both approximate and exact edit distance calculations.

Chapter 9

Evaluation

In this chapter, we evaluate the FPGA resource utilization, the filtering accuracy, and the memory utilization of all our proposed pre-alignment filters. We also investigate the benefits of using our hardware and CPU-based pre-alignment filtering solutions along with the state-of-the-art aligners. We compare the performance of our proposed pre-alignment filters (SneakySnake, MAGNET, SLIDER, and GateKeeper) with the state-of-the-art existing pre-alignment filter, SHD [7] and read aligners, Edlib [8], Parasail [9], GSWABE [58], CUDASW++ 3.0 [59], and FPGASW [56]. We run all experiments using 3.6 GHz Intel i7-3820 CPU with 8 GB RAM. We use a Xilinx Virtex 7 VC709 board [118] to implement our accelerator architecture and our hardware filters. We build the FPGA designs using Vivado 2015.4 in synthesizable Verilog.

9.1 Dataset Description

Our experimental evaluation uses 12 different real datasets. Each dataset contains 30 million real sequence pairs. Next, we elaborate on how we obtain them.

Table 9.1: Benchmark illumina-like read sets of whole human genome, obtained from EMBL-ENA.

Accession no.	**ERR240727_1**	**SRR826460_1**	**SRR826471_1**
Read length (bp)	100	150	250
No. of reads	4 million	89 million	186 million
HTS	Illumina HiSeq 2000	Illumina HiSeq 2000	Illumina HiSeq 2000

Table 9.2: Benchmark illumina-like datasets (read-reference pairs). We map each read set, described in Table 9.1, to the human reference genome in order to generate four datasets using different mapper's edit distance thresholds (using *-e* parameter).

Accession no.	**ERR240727_1**				**SRR826460_1**				**SRR826471_1**			
Dataset no.	1	2	3	4	5	6	7	8	9	10	11	12
mrFAST *-e*	2	3	5	40	4	6	10	70	8	12	15	100

We obtain three different read sets (ERR240727_1, SRR826460_1, and SRR826471_1) of whole human genome that include three different read lengths (100 bp, 150 bp, and 250 bp, respectively), as summarized in Table 9.1. We download these three read sets from EMBL-ENA (http://www.ebi.ac.uk/ena). We map each read set to the human reference genome (GRCh37) using mrFAST [127] mapper. We obtain the human reference genome from 1000 Genomes Project [128]. For each read set, we use four different maximum number of edits using the *-e* parameter of mrFAST to generate four real datasets. We summarize the details of these 12 datasets in Table 9.2. For the convenience of referring to these datasets, we number them from 1 to 12 (e.g., set_1 represents 30 million reads from ERR240727_1 mapped with $-e = 2$ edits). The 12 real datasets enable us to measure the effectiveness of the filters in tolerating low number of edits and far more edits than the allowed edit distance threshold. We provide detailed information on the number of correct and incorrect pairs of each of the 12 datasets for different user-defined edit distance thresholds in Table A.1, Table A.2, and Table A.3 in Appendix A.

9.2 Resource Analysis

We now examine the FPGA resource utilization for the hardware implementation of GateKeeper, SLIDER, MAGNET, and SneakySnake pre-alignment filters. We provide several hardware designs for two commonly used edit distance thresholds, 2 bp and 5 bp as reported in [27, 7, 113, 62], for a sequence length of 100 bp. The VC709 FPGA chip contains 433,200 slice LUTs (look-up tables) and 866,400 slice registers (flip-flops). Table 9.3 lists the FPGA resource utilization for a single filtering unit. We make five main observations.

1. The design for a single MAGNET filtering unit requires about 10.5% and 37.8% of the available LUTs for edit distance thresholds of 2 bp and 5 bp, respectively. Hence, MAGNET can process 8 and 2 sequence pairs concurrently for edit distance thresholds of 2 bp and 5 bp, respectively, without violating the timing constraints of our hardware accelerator.

2. The design for a single SLIDER filtering unit requires about 15x-21.9x less LUTs compared to MAGNET. This enables SLIDER to achieve more parallelism over MAGNET design as it can have 16 filtering units within the same FPGA chip.

3. GateKeeper requires about 26.9x-53x and 1.7x-2.4x less LUTs compared to MAGNET and SLIDER, respectively. GateKeeper can also examine up to 16 sequence pairs at the same time.

4. SneakySnake requires 15.4x-26.6x less LUTs compared to MAGNET. While SneakySnake requires a slightly less LUTs compared to SLIDER, it requires about 2x more LUTs compared to GateKeeper. SneakySnake can also examine up to 16 sequence pairs concurrently.

5. We observe that the hardware implementations of SLIDER, MAGNET, and SneakySnake require pipelining the design (i.e., shortening the critical path delay of each processing core by dividing it into stages or smaller tasks) to enable meeting the timing constraints and achieve more parallelism.

Table 9.3: FPGA resource usage for a single filtering unit of GateKeeper, SLIDER, MAGNET, and SneakySnake for a sequence length of 100 and under different edit distance thresholds (E).

	E (bp)	Slice LUT	Slice Register	No. of Filtering Units
GateKeeper	2	0.39%	0.01%	16
	5	0.71%	0.01%	16
SLIDER	2	0.69%	0.08%	16
	5	1.72%	0.16%	16
MAGNET	2	10.50%	0.80%	8
	5	37.80%	2.30%	2
SneakySnake	2	0.68%	0.16%	16
	5	1.42%	0.34%	16

We build 8 pipeline stages for SLIDER, 22 pipeline stages for MAGNET, and 5 pipeline stage for SneakySnake to satisfy the timing constraints. However, pipelining the design comes with the expense of increased register utilization.

We conclude that the FPGA resource usage is correlated with the filtering accuracy. For example, the least accurate filter, GateKeeper, occupies the least FPGA resource that can be integrated into the FPGA. We also conclude that the less the logic utilization of a single filtering unit, the more the number of filtering units.

9.3 Filtering Accuracy

Next, we assess the false accept rate and false reject rate of GateKeeper, SLIDER, MAGNET and SneakySnake across our 12 datasets. We also investigate and address several concerns that we raise in Chapter 8. We compare the accuracy performance of our proposed pre-alignment filters with the best performing existing pre-alignment filter, SHD [7].

SHD supports a sequence length of up to only 128 characters (due to the SIMD register size). To ensure as fair a comparison as possible, we allow SHD to divide the long sequences into batches of 128 characters, examine each batch individually, and then sum up the results. As we describe in Chapter 3, we aim to minimize the false accept rate so that the elimination of dissimilar sequences is maximized. We also aim to maintain a 0% false reject rate. We use Edlib [8] (set to edit distance mode) to generate the ground truth edit distance value for each sequence pair as it has a zero false accept rate and a zero false reject rate.

9.3.1 Partitioning the Search Space of Approximate and Exact Edit Distance

We raise two key questions in Chapter 8. (1) Can one approximate the edit distance between two sequences much faster than calculating the edit distance? (2) Can one reduce the search space of approximate and exact edit distance algorithms? To answer these two questions, we first evaluate the performance of SneakySnake (approximate edit distance algorithm) and then examine the feasibility of reducing its search space without causing falsely-rejected mappings. Secondly, we examine the ability to implement the best performing existing exact edit distance algorithm, Edlib [8] such that it calculates the prefix edit distance. As we discuss in Chapter 8, we column-wise partition the unweighted neighborhood maze of SneakySnake algorithm into adjacent non-overlapping sub-matrices of the same size ($2E+1$ by t). In Figure 9.1, we illustrate the effects of this partitioning on the false accept rate and the execution time of our SneakySnake algorithm. We make two observations.

1. **Partitioning the search space of SneakySnake with a partition size of 5 (t=5) reduces its execution time by up to 5.12x (Figure 9.1a), 7.4x (Figure 9.1c), and 13.2x (Figure 9.1e)** at the expense of increased false accept rate by up to 55.4x (Figure 9.1 b), 43.5x (Figure 9.1d), and 67.3x (Figure 9.1f).

2. There is a trade-off between the speed and the accuracy of SneakySnake algorithm. For example, the least accurate filter, SneakySnake with a partition size of 5 columns (or in short SneakySnake-5) yields the fastest speed.

Next, we assess the effect of the number of replications on the filtering accuracy of the hardware implementation of the SneakySnake algorithm. We use a sub-matrix's width of 8 columns (t=8) and we vary the height of the sub-matrix from 1 row (i.e., E=0 bp) up to 21 rows (i.e., E=10 bp). Based on Figure 9.2, we make two observations:

1. We observe that increasing the number of the replications in the design improves the filtering accuracy of the SneakySnake algorithm. This observation is in accord with our expectation as each replication detects at most a single edit within each sub-matrix.

2. We also observe that the hardware implementation of SneakySnake using 3 replications (3 iterations for finding the optimal path within each sub-matrix) achieves a similar accuracy performance (or slightly better) as that of the SneakySnake-5.

We conclude that partitioning the search space of the SneakySnake algorithm is also beneficial for building an efficient hardware architecture while maintaining high filtering accuracy.

Now, we modify Edlib algorithm such that it applies *a prefix edit distance* (we provide the definition in Chapter 8). It starts computing only a small square sub-matrix. If the computed edit distance meets the user-defined edit distance threshold, then it extends the sub-matrix into a larger square one (which overlaps entirely with the smaller one) by increasing the number of columns and rows by a constant. For example, if the initial sub-matrix size is 5 columns by 5 rows, then we extend it into a larger one of size 10 columns by 10 rows, and next we extend it into a sub-matrix of size 15 columns by 15 rows.

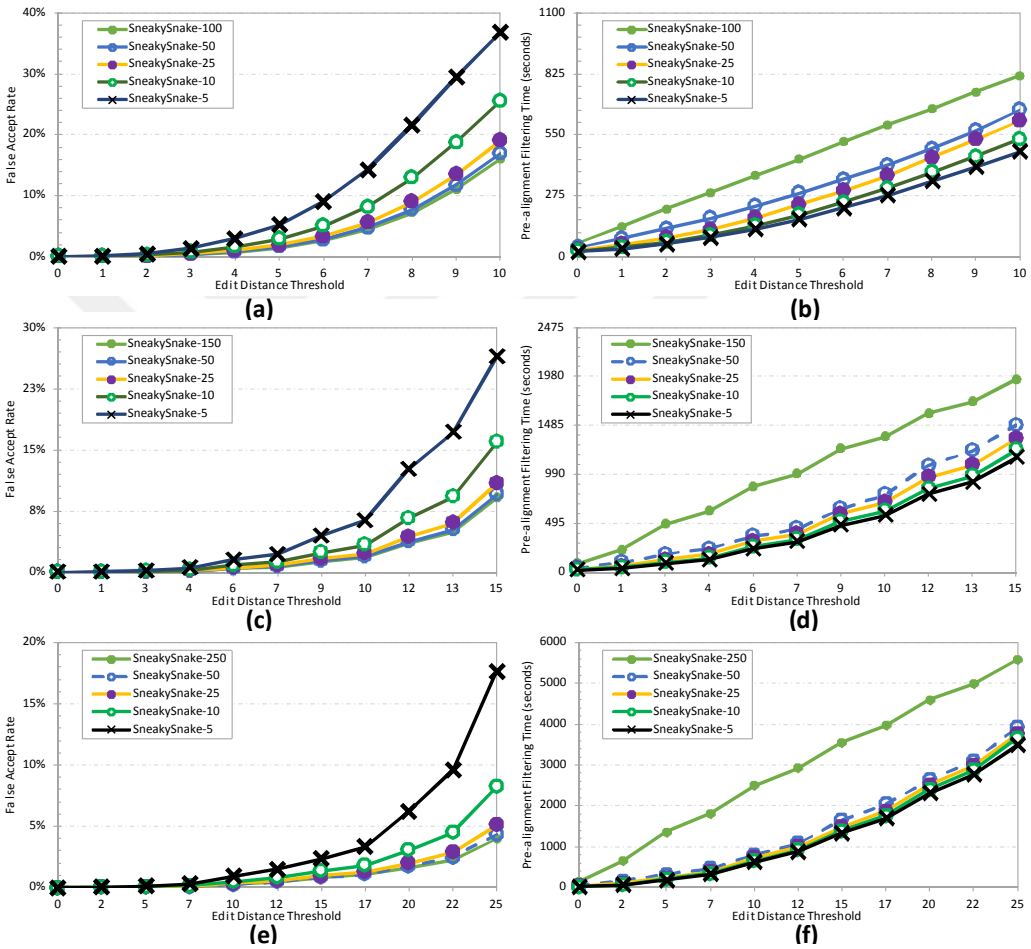

Figure 9.1: The effects of column-wise partitioning the search space of SneakySnake algorithm on the average false accept rate ((a), (c), and (e)) and the average execution time ((b), (d), and (f)) of examining set_1 to set_4 in (a) and (b), set_5 to set_8 in (c) and (d), and set_9 to set_12 in (e) and (f). Besides the default size (equals the read length) of the SneakySnake's unweighted neighborhood maze, we choose partition sizes (the number of grid's columns that are included in each partition) of 5, 10, 25, and 50 columns.

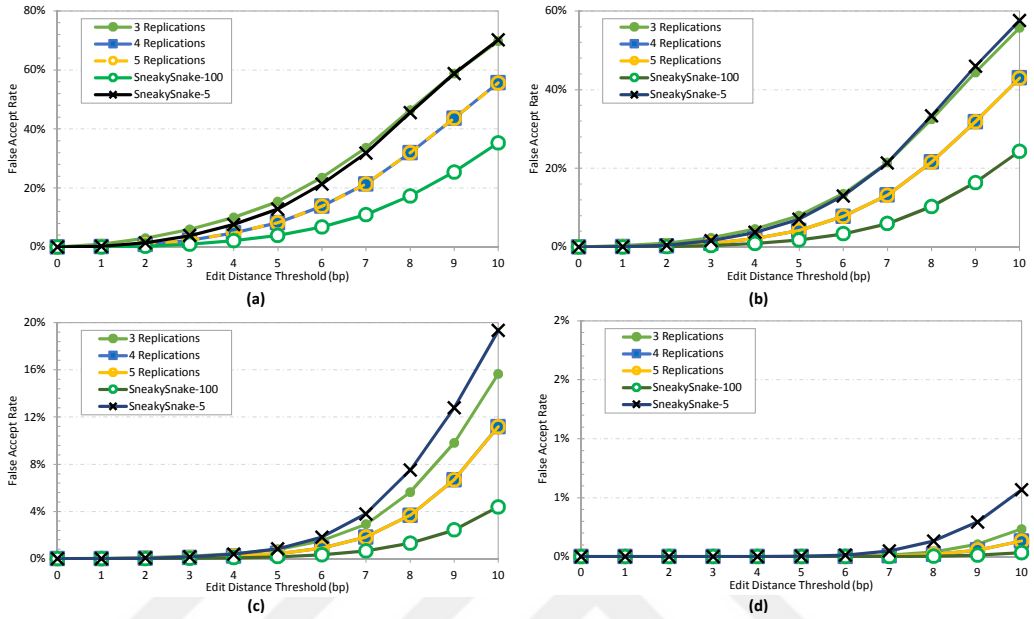

Figure 9.2: The effects of the number of replications of the hardware implementation of SneakySnake algorithm on its filtering accuracy (false accept rate). We use a wide range of edit distance thresholds (0 bp-10 bp for a read length of 100 bp) and four datasets: (a) set_1, (b) set_2, (c) set_3, and (d) set_4.

We keep extending the size of the sub-matrix until we cover the entire dynamic programming matrix or the prefix edit distance exceeds the edit distance threshold. In Figure 9.3, we present the effects of computing the prefix edit distance (partitioning the edit distance matrix) on the overall execution time. We also provide the performance of the original and the partitioned SneakySnake algorithm for a better comparison. We provide more detailed results in Table A.4, Table A.5, and Table A.6 in Appendix A. We make two key observations.

1. **Our SneakySnake algorithm with a partition size of 5 (SneakySnake-5) is up to 25.5x (Figure 9.3a), 52.5x (Figure 9.3b), and 94.5x (Figure 9.3c) faster than the best performing edit distance algorithm, Edlib** [8].

2. Prefix edit distance with large enough initial sub-matrix size provides a slight reduction in the execution time of Edlib.

We observe that setting the initial sub-matrix size to 50 bp provides the highest reduction in the Edlib's execution time over a wide range of edit distance thresholds. Edlib with an initial sub-matrix size of 50 bp (or in short Edlib-50) is up to 1.9x (Figure 9.3a), 4.4x (Figure 9.3b), and 7.8x (Figure 9.3c) faster than the original Edlib. However, SneakySnake-5 is still up to an order of magnitude (19.8x) faster than Edlib-50 when the edit distance threshold is set to 9 (for m=100) or 10 (for m=150 or 250) and below, as highlighted with a dashed vertical line in Figure 9.3. Note that E is typically less than or equal 5% of the read length [27, 7, 113, 62].

We conclude that reducing the search space of both approximate and exact edit distance algorithms is beneficial. Combining SneakySnake-5 and Edlib-50 can provide a fast examination for incorrect mappings across a wide range of edit distance thresholds (0% - 10% of the read length).

9.3.2 False Accept Rate

We present in Figure 9.4, Figure 9.5, and Figure 9.6 the false accept rate of our pre-alignment filters compared to SHD for read lengths of 100 bp, 150 bp, and 250 bp, respectively. We make six key observations.

1. We observe that SLIDER, MAGNET, GateKeeper, and SneakySnake are less accurate in examining the low-edit sequences (Figure 9.4a,b, Figure 9.5a,b, and Figure 9.6a,b) than the edit-rich sequences (Figure 9.4c,d, Figure 9.5c,d, and Figure 9.6c,d). While SneakySnake pre-alignment filter yields the highest accuracy, SHD [7] and GateKeeper provide the least accuracy compared to all other pre-alignment filters. The slope of MAGNET plot is almost comparable to that of the SneakySnake (with the default maze size).

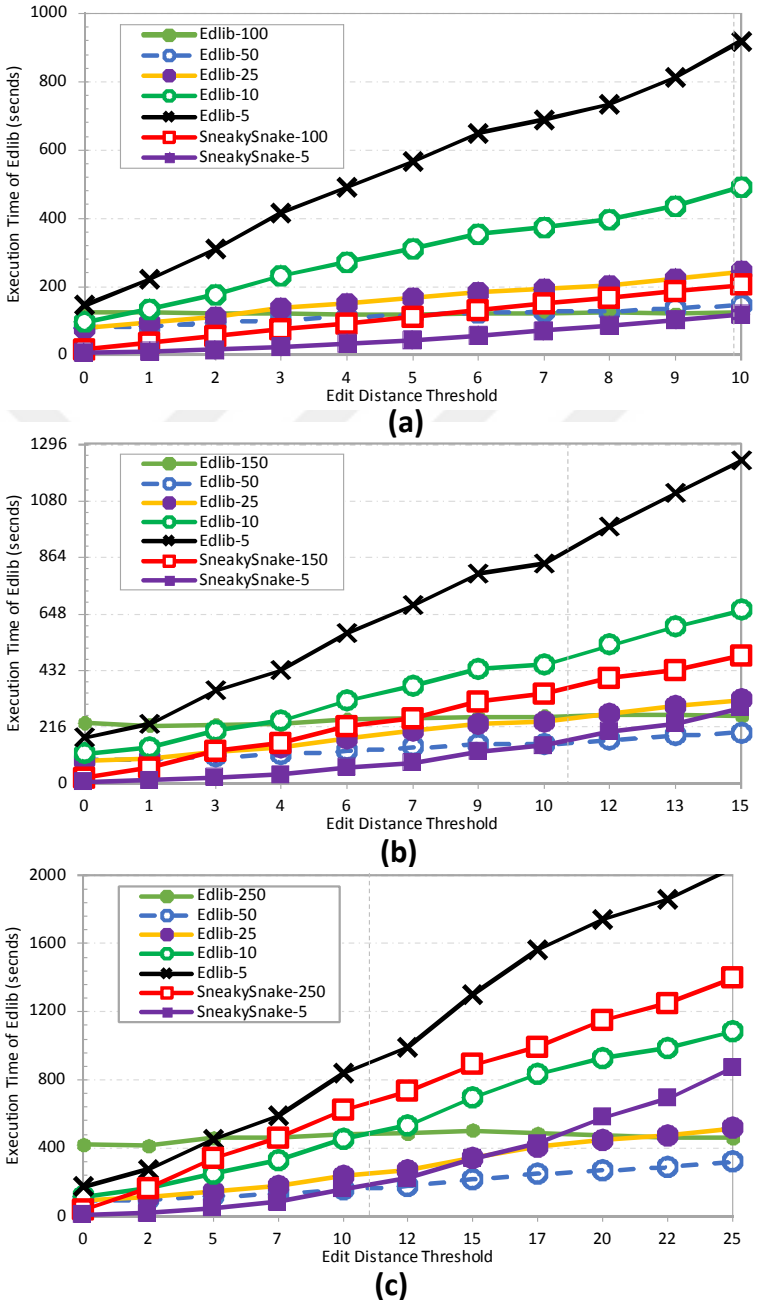

Figure 9.3: The effects of computing the *prefix edit distance* on the overall execution time of the edit distance calculations compared to the original Edlib (exact edit distance) and our partitioned implementation of SneakySnake algorithm. We present the average time spent in examining set_1 to set_4 in (a), set_5 to set_8 in (b), and set_9 to set_12 in (c). We choose initial sub-matrix sizes of 5, 10, 25, and 50 columns. We mark the intersection of SneakySnake-5 and Edlib-50 plots with a dashed vertical line.

2. GateKeeper and SHD [7] become ineffective for edit distance thresholds of greater than 8% for $m = 100$ bp (Figure 9.4), 5% for $m = 150$ bp (Figure 9.5), and 3% for $m = 250$ bp (Figure 9.6) for both low-edit and edit-rich sequences, where m is the read length. This leads to examining each sequence pair twice unnecessarily, by both GateKeeper or SHD and the full alignment step.

3. SLIDER provides up to 17.2x, 73x, and 467x less false accept rate compared to GateKeeper and SHD for read lengths of 100 bp, 150 bp, and 250 bp, respectively.

4. MAGNET, SneakySnake (with the default maze size), and SneakySnake-5 show a slow exponential degradation in their filtering inaccuracy for low-edit sequences with a false accept rate of up to 50%, 30%, and 70%, respectively, as we show in Figure 9.4a,b, Figure 9.5a,b, and Figure 9.6a,b. They also show almost a linear growth in their false accept rate of less than 4% for edit-rich sequences of different read lengths, as we show in Figure 9.4c,d, Figure 9.5c,d, and Figure 9.6c,d.

5. MAGNET shows up to 1577x, 3550x, and 25552x less false accept rate compared to GateKeeper and SHD for read lengths of 100 bp, 150 bp, and 250 bp, respectively. MAGNET also provides up to 205x, 951x, and 16760x less false accept rate compared to SLIDER for read lengths of 100 bp, 150 bp, and 250 bp, respectively.

6. SneakySnake (with the default maze size) produces up to four and five orders of magnitude less false accept rate compared to SLIDER and GateKeeper, respectively. SneakySnake shows up to 55.4x, 46.7x, and 67.1x less false accept rate compared to SneakySnake-5 for read lengths of 100 bp, 150 bp, and 250 bp, respectively. SneakySnake also shows up to 64.1x, 16x, and 22x less false accept rate compared to MAGNET for read lengths of 100 bp, 150 bp, and 250 bp, respectively.

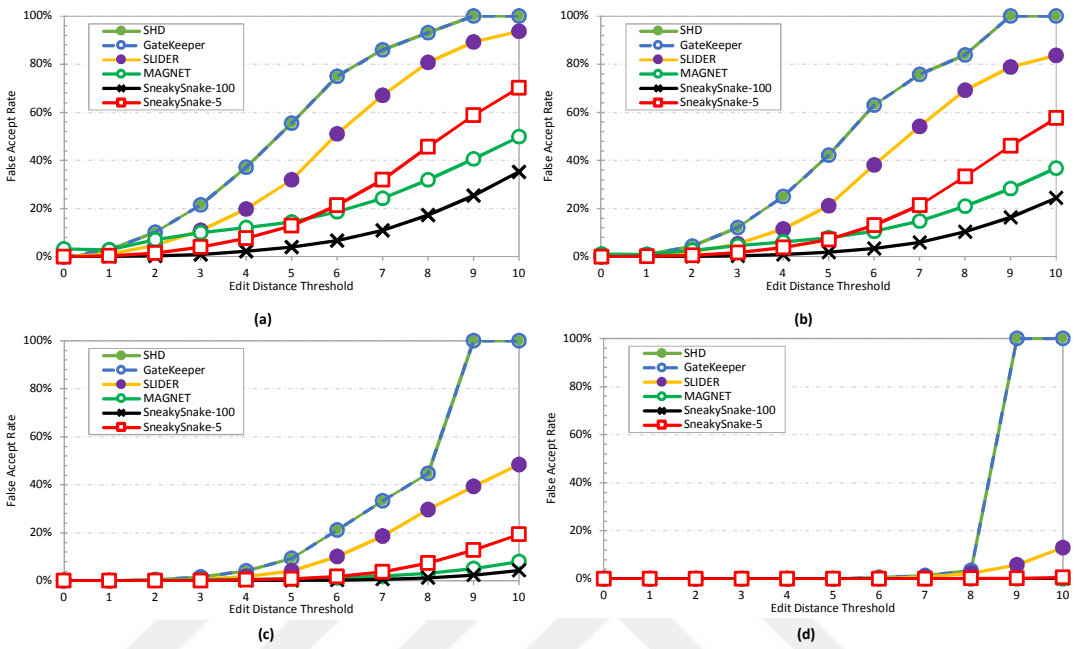

Figure 9.4: The false accept rate produced by our pre-alignment filters, GateKeeper, SLIDER, MAGNET, and SneakySnake, compared to the best performing filter, SHD [7]. We use a wide range of edit distance thresholds (0-10 edits for a read length of 100 bp) and four datasets: (a) set_1, (b) set_2, (c) set_3, and (d) set_4.

We conclude that SneakySnake, MAGNET, and SLIDER are very effective and superior to the state-of-the-art pre-alignment filter, SHD [7] in both situations (low-edit and edit-rich mappings). They maintain a very low rate of falsely-accepted incorrect mappings and significantly improves the accuracy of pre-alignment filtering by up to five orders of magnitude compared to GateKeeper and SHD.

9.3.3 False Reject Rate

Using our 12 low-edit and edit-rich datasets for three different read lengths, we observe that SneakySnake (for all partition sizes), SLIDER, and GateKeeper do not filter out correct mappings; hence, they provide a 0% false reject rate. The reason behind that is the way we find the identical subsequences.

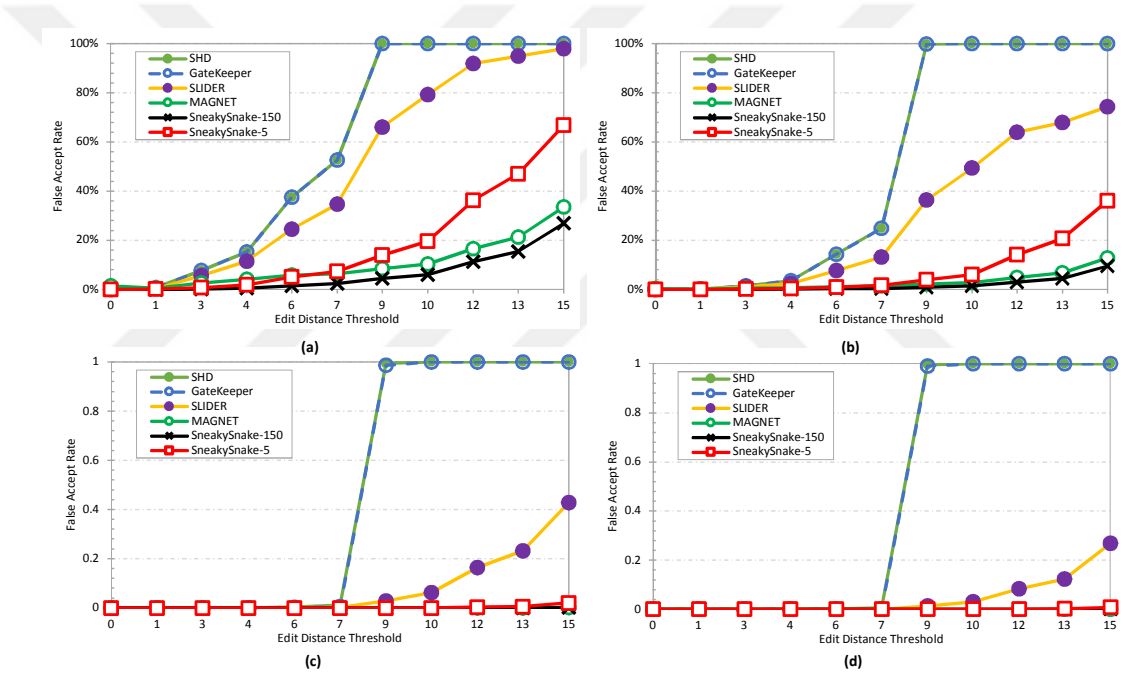

Figure 9.5: The false accept rate produced by our pre-alignment filters, GateKeeper, SLIDER, MAGNET, and SneakySnake, compared to the best performing filter, SHD [7]. We use a wide range of edit distance thresholds (0-15 edits for a read length of 150 bp) and four datasets: (a) set_5, (b) set_6, (c) set_7, and (d) set_8.

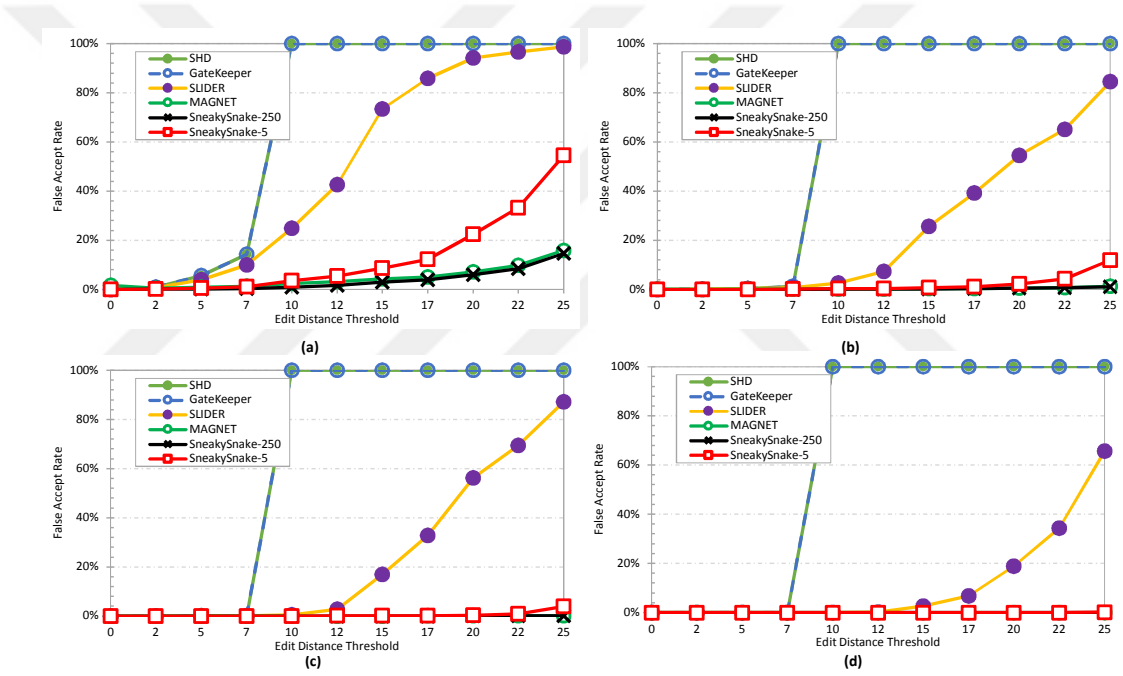

Figure 9.6: The false accept rate produced by our pre-alignment filters, GateKeeper, SLIDER, MAGNET, and SneakySnake, compared to the best performing filter, SHD [7]. We use a wide range of edit distance thresholds (0-25 edits for a read length of 250 bp) and four datasets: (a) set_9, (b) set_10, (c) set_11, and (d) set_12.

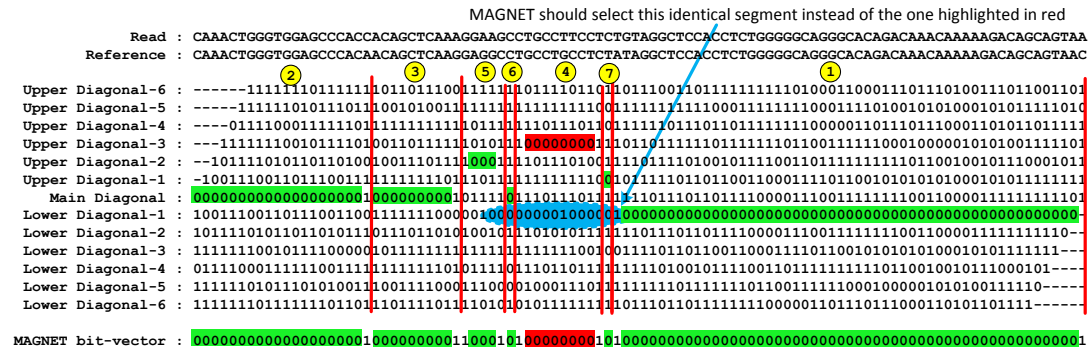

Figure 9.7: An example of a falsely-rejected mapping using MAGNET algorithm for an edit distance threshold of 6. The random zeros (highlighted in red) confuse MAGNET filter causing it to select shorter segments of random zeros instead of a longer identical subsequence (highlighted in blue).

We aim to find the subsequences that has the largest number of zeros, such that we maximize the number of matches and minimize the number of edits that cause the division of one long identical sequence into shorter subsequences. However, this is not the case for SHD and MAGNET. We observe that SHD [7] falsely rejects correct mappings of 100 bp long reads when the edit distance threshold is set to 0 bp. Using set_1, set_2, set_3, and set_4, SHD shows false reject rates of 45%, 51%, 64%, 100%, respectively. GateKeeper addresses this problem by calculating the Hamming distance [129] when the edit distance threshold is set to 0 bp.

MAGNET shows a very low false reject rate of less than 0.00045% for an edit distance threshold of at least 4 bp. This is due in large part to the greedy choice of always selecting the longest identical subsequence based on their descending length and regardless of their source (i.e. the vectors that originate them). We provide in Figure 9.7 an example of where MAGNET fails to maintain the correct mappings. A potential solution is to relate the number of encapsulated bits to the source of each segment. For instance, if any extracted segment in the MAGNET mask has a different source than its neighboring segments, then we need to penalize that segment by adding more encapsulation bits. This can help us to produce accurate *CIGAR* string for each mapping.

With the help of an auxiliary data structure, we can keep track of the source of each extracted segment at each position. While the matches coming from the "upper diagonal-1" mean there is a single deletion, the "upper diagonal-2" means that there are two deletions. A detailed algorithm of this topic is beyond the scope of this thesis and is part of our future work.

9.4 Effects of Hardware Pre-Alignment Filtering on Read Alignment

We first analyze the execution time of our hardware pre-alignment filters, GateKeeper, MAGNET, SLIDER, and SneakySnake. We build the FPGA implementation of SneakySnake using a sub-matrix's width of 8 columns (t=8) and we include 3 replications in the design. We use GateKeeper as an optimized and efficient hardware implementation of SHD [7]. We evaluate our four pre-alignment filters using a single FPGA chip. We use 120 million sequence pairs, each of which is 100 bp long, from set_1, set_2, set_3, and set_4. We summarize the execution time of the CPU implementations along with that of their hardware accelerators in Table 9.4. We make two key observations based on Table 9.4.

1. Our hardware accelerators provide two to three orders of magnitude (322x to 7,250x) speedup over their CPU implementations. GateKeeper provides up to two orders of magnitude of acceleration over SHD.

2. The execution time of the hardware implementation of SneakySnake and SLIDER are as low as that of GateKeeper and 2x-8x lower than that of MAGNET pre-alignment filter. This observation is in accord with our expectation and can be explained by the fact that MAGNET has more computational overhead that limits the number of filtering units. Yet SneakySnake is four and five orders of magnitude more accurate than both SLIDER and GateKeeper (as we show earlier).

Table 9.4: The execution time (in seconds) of GateKeeper, MAGNET, SLIDER, and SneakySnake under different edit distance thresholds. We use set_1 to set_4 with a read length of 100. We provide the performance results for the CPU implementations and the hardware accelerators with the maximum number of filtering units.

***E* (bp)**	**GateKeeper**	**MAGNET**	**SLIDER**	**SneakySnake-FPGA**
2	0.18	0.36	0.18	0.18
5	0.18	1.45	0.18	0.18
***E* (bp)**	**SHD**	**MAGNET-CPU**	**SLIDER-CPU**	**SneakySnake-5**
2	60.3	632	474.2	58.1
5	67.9	1,641.50	1,305.10	169

We conclude that our hardware accelerator provides two and three orders of magnitude of speedup over their CPU implementations. Additionally, the execution time of the hardware accelerator is proportional to the FPGA resource utilization (the less the resource utilization the lower the execution time).

Next we analyze the benefits of integrating our hardware pre-alignment filters with the state-of-the-art aligners. GateKeeper, MAGNET, SLIDER, and SneakySnake are standalone pre-alignment filters and can be integrated with any existing alignment algorithm. In Table 9.5, we present the effects of our four hardware pre-alignment filters on the overall alignment's execution time. We use a sub-matrix's width of 8 columns (t=8) and we include 3 replications in the design of the hardware architecture of the SneakySnake algorithm. We also compare the effect of our pre-alignment filters with that of SHD [7]. We select five best performing aligners, each of which is designed for different type of computing platforms. While Edlib [8] algorithm is implemented in C for standard CPUs, Parasail [9] exploits SIMD capable CPUs. GSWABE [58] is designed for GPUs. CUDASW++ 3.0 [59] exploits SIMD capability of both CPUs and GPUs. FPGASW [56] exploit the very large number of hardware execution units offered by the same FPGA chip (i.e., VC709) as our accelerator. We evaluate the execution time of Edlib [8] and Parasail [9] on our machine.

However, FPGASW [56], CUDASW++ 3.0 [59], and GSWABE [58] are not open-source and not available to us. Therefore, we scale the reported number of computed entries of the dynamic programming matrix in a second. We use a total of 120 million real sequence pairs from our previously described four datasets (set_1, set_2, set_3, and set_4) in this analysis. We make three key observations.

1. The execution time of Edlib [8] reduces by up to 21.4x, 18.8x, 16.5x, 13.9x, and 5.2x after the addition of SneakySnake, SLIDER, MAGNET, GateKeeper, and SHD, respectively, as a pre-alignment filtering step. We also observe nearly a similar trend for Parasail (Daily, 2016) combined with each of the four pre-alignment filters.

2. Aligners designed for FPGAs and GPUs follow a different trend than that we observe in the CPU aligners. We observe that the ability of SHD [7] to reduce the alignment time of GSWABE [58], CUDASW++ 3.0 [59], and FPGASW [56] diminishes. SHD even provides unsatisfactory performance as it increases the execution time of the aligner instead of reducing it. This is due to the fact that SHD is 6x slower than CUDASW++ 3.0 [59] and FPGASW [56] and it is lightly slower than GSWABE [58].

3. SneakySnake, SLIDER, MAGNET, and GateKeeper still contribute significantly towards reducing the overall execution time of FPGA and GPU based aligners. SneakySnake reduces the execution time of FPGASW [56], CUDASW++ 3.0 [59] and GSWABE [58] by factors of up to 16x, 15.5x, and 20.3x, respectively. This is slightly higher (up to 1.3x) than the effect of slider on the execution time of these aligners. SLIDER reduces the overall alignment time of FPGASW [56], CUDASW++ 3.0 [59] and GSWABE [58] by factors of up to 14.5x, 14.2x, and 17.9x, respectively. This is up to 1.5x, 1.4x, and 85x more than the effect of MAGNET, GateKeeper, and SHD on the end-to-end alignment time.

We conclude that among the four hardware pre-alignment filters, SneakySnake (3-replication design and with t=8) is the best performing filter in terms of both speed and accuracy.

Table 9.5: End-to-end execution time (in seconds) for several state-of-the-art sequence alignment algorithms, with and without pre-alignment filters (SneakySnake, SLIDER, MAGNET, GateKeeper, and SHD) and across different edit distance thresholds. We use four datasets (set_1, set_2, set_3, and set_4) across different edit distance thresholds.

***E* (bp)**	**Edlib**	**w/SneakySnake**	**w/ SLIDER**	**w/ MAGNET**	**w/ GateKeeper**	**w/ SHD**
2	506.66	23.65	26.86	30.69	36.39	96.54
5	632.95	106.48	147.20	106.80	208.77	276.51
***E* (bp)**	**Parasail**	**w/SneakySnake**	**w/ SLIDER**	**w/ MAGNET**	**w/ GateKeeper**	**w/ SHD**
2	1,310.96	60.92	69.21	78.83	93.87	154.02
5	2,044.58	343.54	475.08	341.77	673.99	741.73
***E* (bp)**	**FPGASW**	**w/SneakySnake**	**w/ SLIDER**	**w/ MAGNET**	**w/ GateKeeper**	**w/ SHD**
2	11.33	0.70	0.78	1.04	0.99	61.14
5	11.33	2.08	2.81	3.34	3.91	71.65
***E* (bp)**	**CUDASW++ 3.0**	**w/SneakySnake**	**w/ SLIDER**	**w/ MAGNET**	**w/ GateKeeper**	**w/ SHD**
2	10.08	0.65	0.71	0.96	0.90	61.05
5	10.08	1.87	2.52	3.13	3.50	71.24
***E* (bp)**	**GSWABE**	**w/SneakySnake**	**w/ SLIDER**	**w/ MAGNET**	**w/ GateKeeper**	**w/ SHD**
2	61.86	3.05	3.44	4.06	4.60	64.75
5	61.86	10.57	14.55	11.75	20.57	88.31

Integrating SneakySnake with aligner does not lead to negative effects. We also conclude that the concept of pre-alignment filtering is still effective in boosting the overall performance of the alignment step, even the dynamic programming algorithm is accelerated by the state-of-the-art hardware accelerators such as SIMD-capable CPUs, FPGAs, and GPUs.

9.5 Effects of CPU Pre-Alignment Filtering on Read Alignment

Now we analyze the benefits of integrating our CPU implementations of SneakySnake and the prefix edit distance with the state-of-the-art CPU aligners, Edlib [8] and Parasail [9].

We evaluate the end-to-end execution time of Edlib (referred to as path in [8] and configured as a banded global Levenshtein distance with CIGAR-enabled output) with and without the pre-alignment filtering step for read lengths of 100 bp (Figure 9.8), 150 bp (Figure 9.9), and 250 bp (Figure 9.10). We make three key observations.

1. SneakySnake-5 combined with the banded Edlib is up to 20.4x, 33x, and 43x faster than Edlib without pre-alignment filter for E<9 bp (m=100 bp), E<12 bp (m=150 bp), and E<15 bp (m=250 bp), where E is the edit distance threshold and m is the read length.

2. Edlib-50 combined with the banded Edlib is slower than SneakySnake-5 combined with the same aligner for low-edit datasets (Figure 9.8a-b, Figure 9.9a-b, and Figure 9.10a-b). Edlib-50 becomes more effective over SneakySnake-5 in reducing the execution time of Edlib across edit-rich datasets for E>6 bp (m=100 bp and 150 bp) and E>7 bp (m=250 bp).

3. SHD leads to a negative effect as it slows down the alignment speed of Edlib for E>8 (m=100 bp), E>7 (m=150 bp and 250 bp).

Secondly, we evaluate the effects of adding these pre-alignment filters to Parasail [9] for read lengths of 100 bp (Figure 9.11), 150 bp (Figure 9.12), and 250 bp (Figure 9.13). We configure Parasail as NW_banded and with CIGAR-enabled output. We make three key observations.

1. SneakySnake-5 still provides significant benefits to the the highest end-to-end speedup over all other pre-alignment filters when combined with Parasail for E<7 bp (m=100 bp), E<9 bp (m=150 bp), and E<10 bp (m=250 bp). SneakySnake-5 combined with Parasail yields up to 36.3x, 42x, and 57.9x speedup over Parasail without a pre-alignment filter for read lengths of 100 bp, 150 bp, and 250 bp, respectively.

2. Edlib-50 combined with Parasail becomes more effective than SneakySnake-5 combined with Parasail for E>6 bp (m=100 bp), E>7 bp (m=150 bp), and E>7 bp (m=250 bp).

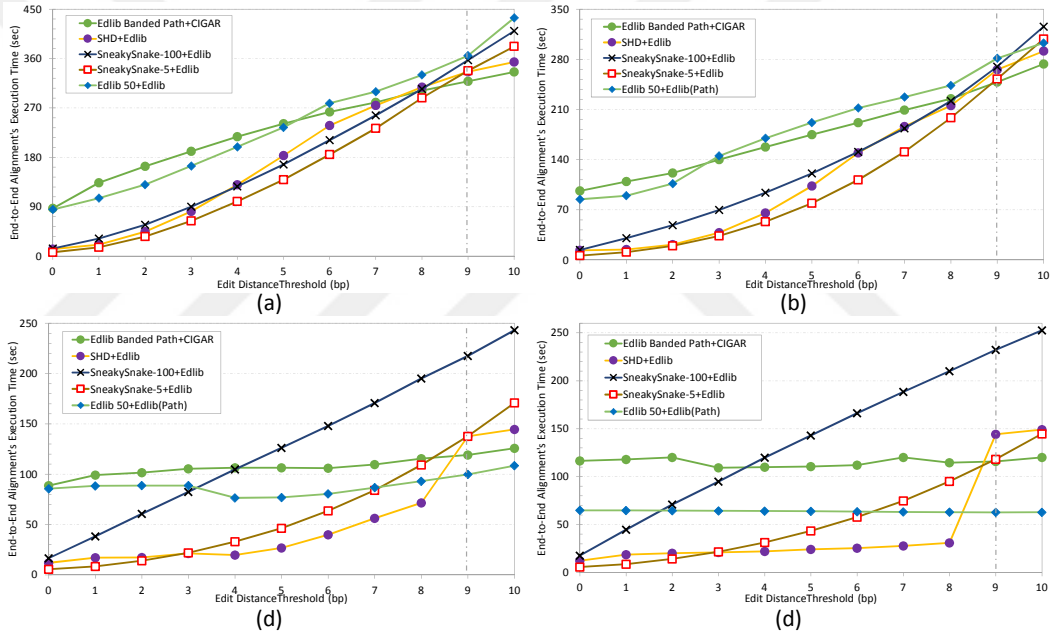

Figure 9.8: End-to-end execution time (in seconds) for Edlib [8] (full read aligner), with and without pre-alignment filters. We use four datasets ((a) set_1, (b) set_2, (c) set_3, and (d) set_4) across different edit distance thresholds. We highlight in a dashed vertical line the edit distance threshold where Edlib starts to outperform our SneakySnake-5 algorithm.

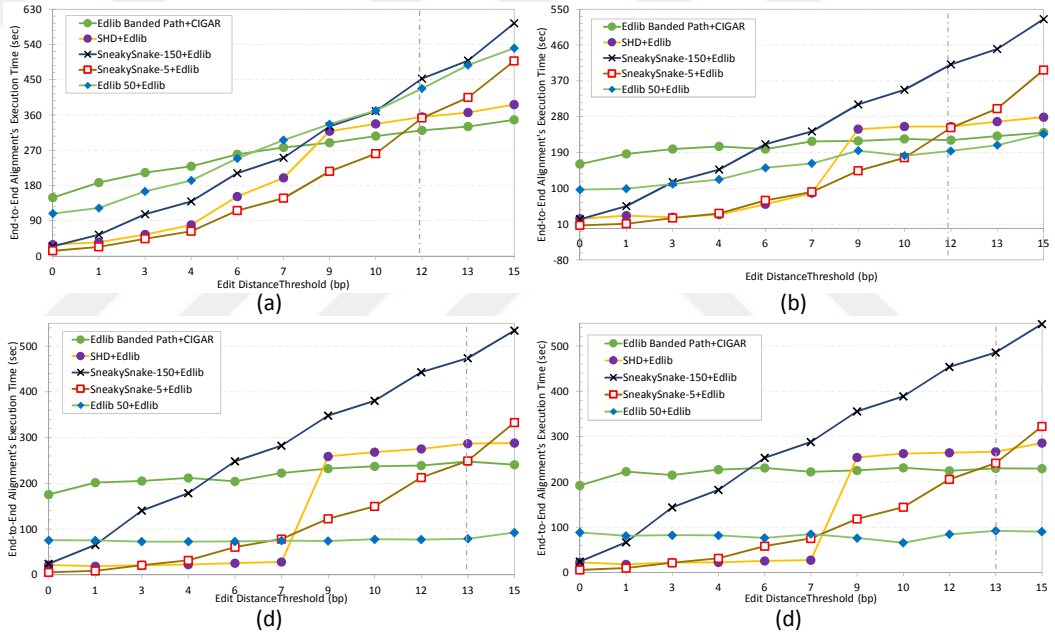

Figure 9.9: End-to-end execution time (in seconds) for Edlib [8] (full read aligner), with and without pre-alignment filters. We use four datasets ((a) set_5, (b) set_6, (c) set_7, and (d) set_8) across different edit distance thresholds. We highlight in a dashed vertical line the edit distance threshold where Edlib starts to outperform our SneakySnake-5 algorithm.

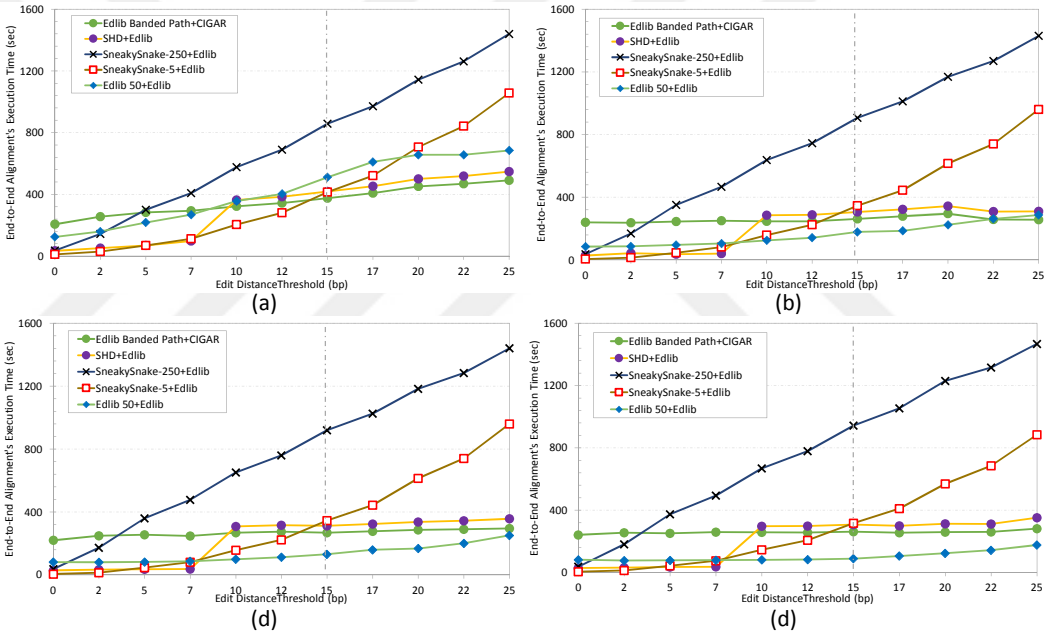

Figure 9.10: End-to-end execution time (in seconds) for Edlib [8] (full read aligner), with and without pre-alignment filters. We use four datasets ((a) set_9, (b) set_10, (c) set_11, and (d) set_12) across different edit distance thresholds. We highlight in a dashed vertical line the edit distance threshold where Edlib starts to outperform our SneakySnake-5 algorithm.

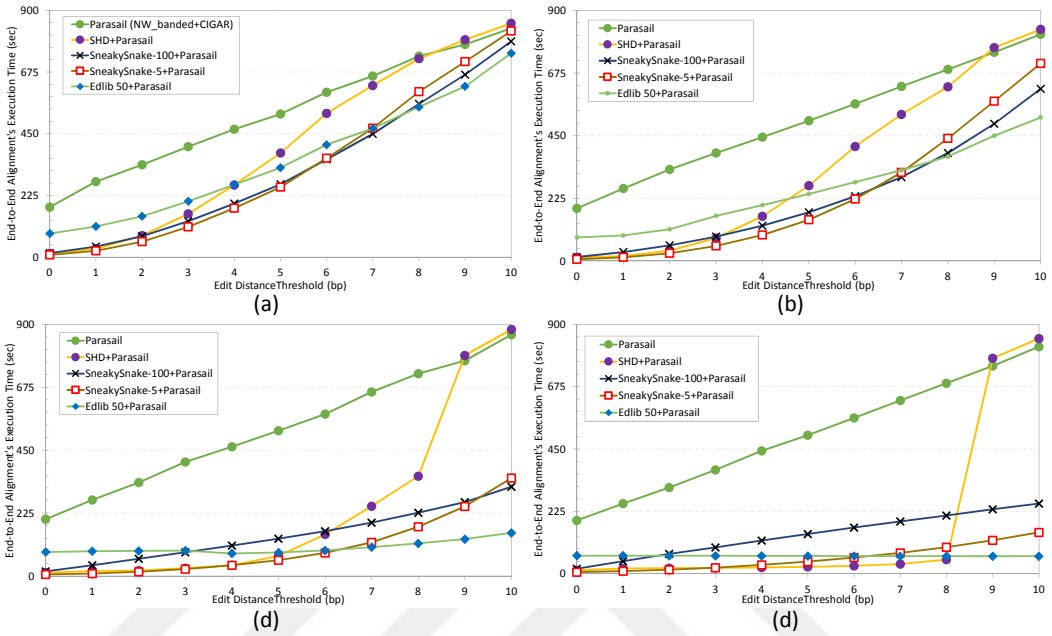

Figure 9.11: End-to-end execution time (in seconds) for Parasail [9] (full read aligner), with and without pre-alignment filters. We use four datasets ((a) set_1, (b) set_2, (c) set_3, and (d) set_4) across different edit distance thresholds.

3. SHD leads to slowing down the alignment speed of Parasail for E>8 (m=100 bp), E>7 (m=150 bp and 250 bp).

We conclude that our SneakySnake algorithm is the best performing CPU pre-alignment filter in terms of both speed and accuracy. It accelerates the state-of-the-art read alignment algorithms by up to an order of magnitude of acceleration. We demonstrate that SneakySnake algorithm does not lead to negative effects for edit distance thresholds of 0% to 10% of the read length. We also want to emphasize that combining our SneakySnake pre-alignment filter (i.e., SneakySnake-5) with a prefix edit distance algorithm (i.e., Edlib-50) provides the fastest and the most accurate pre-alignment filtering across wide range of edit distance thresholds and read lengths.

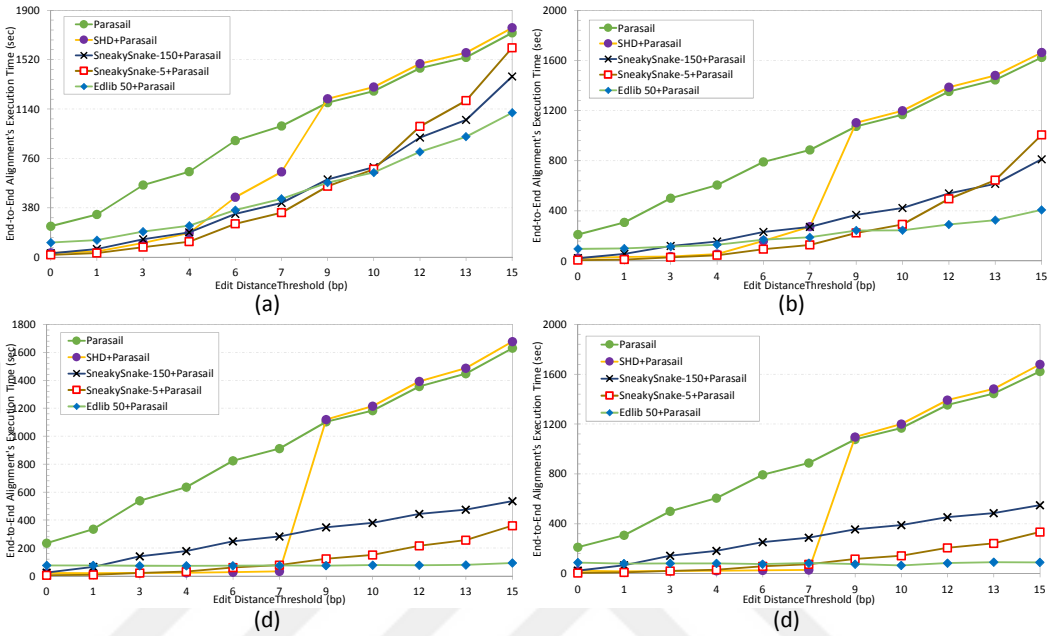

Figure 9.12: End-to-end execution time (in seconds) for Parasail [9] (full read aligner), with and without pre-alignment filters. We use four datasets ((a) set_5, (b) set_6, (c) set_7, and (d) set_8) across different edit distance thresholds.

9.6 Memory Utilization

In this section, we evaluate the space-efficiency benefits of integrating our SneakySnake algorithm with the state-of-the-art full read aligner algorithm, Edlib [8]. We use Valgrind massif tool to examine the memory utilization. We provide the memory footprint of Edlib in Figure 9.14. On average, Edlib shows a memory footprint of 150 KB. We then evaluate the memory utilization of integrating Edlib (in edit distance mode and without backtracking) with Edlib (path) in Figure 9.15. The addition of exact edit distance algorithm to the read alignment shows a slight reduction ($\sim$ 5%) in the memory utilization. With the addition of our SneakySnake-5 as a pre-alignment step before performing Edlib's full alignment, we observe that the memory footprint drops significantly by at least 50%, as we demonstrate in Figure 9.16.

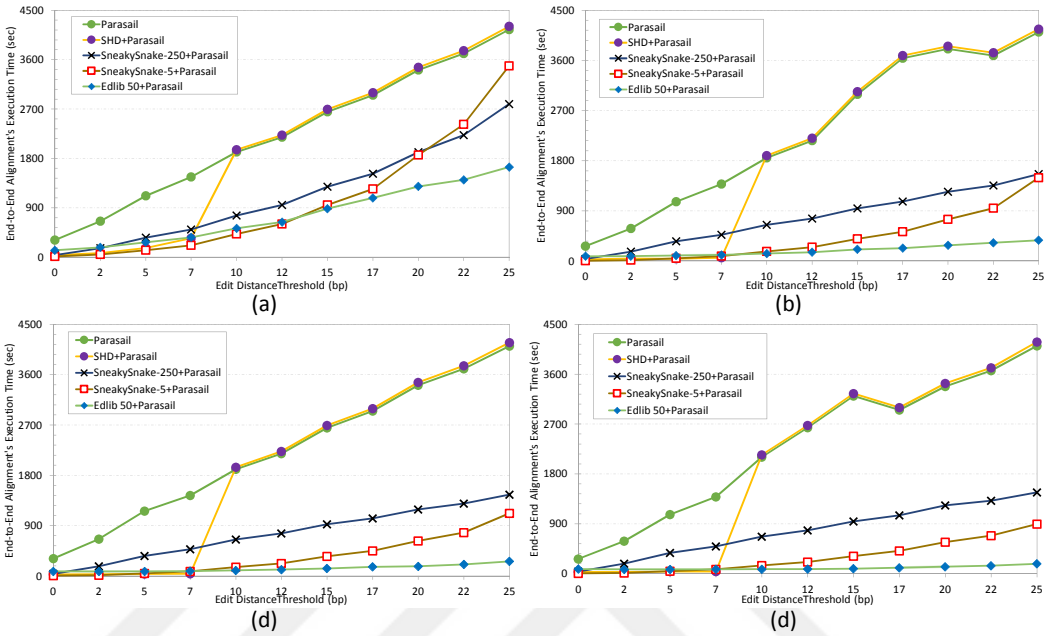

Figure 9.13: End-to-end execution time (in seconds) for Parasail [9] (full read aligner), with and without pre-alignment filters. We use four datasets ((a) set_9, (b) set_10, (c) set_11, and (d) set_12) across different edit distance thresholds.

This observation is in accord with our expectation and can be explained by the fact that SneakySnake-5 requires a spaces of as small as a sub-matrix of size 5 x $2E$+1 and at most m x $2E$+1, whereas Edlib always requires a space of m x m. We conclude that SneakySnake algorithm is fast, accurate, and yet memory-efficient.

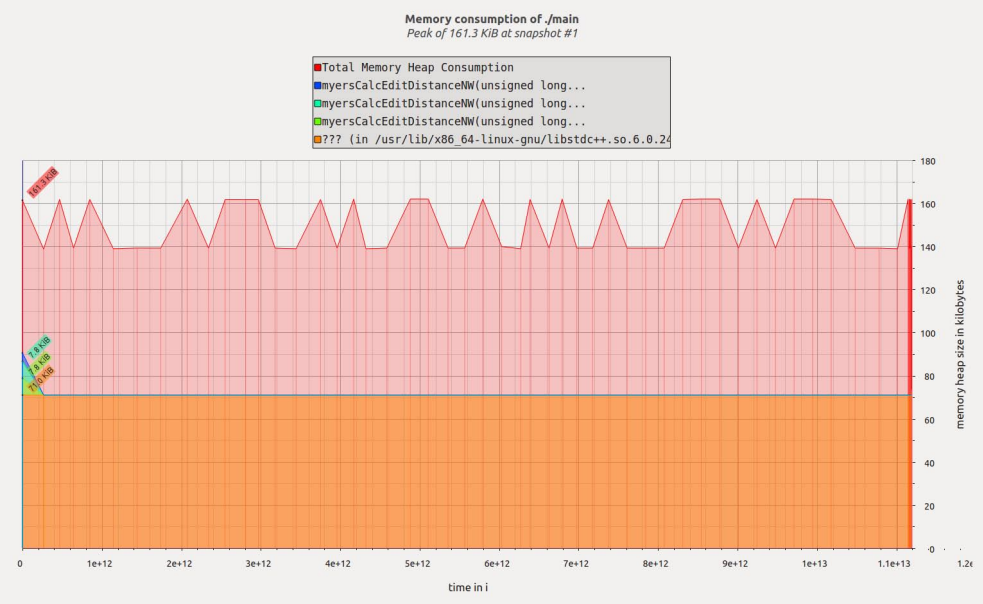

Figure 9.14: Memory utilization of Edlib (path) read aligner while evaluating set_12 for an edit distance threshold of 25.

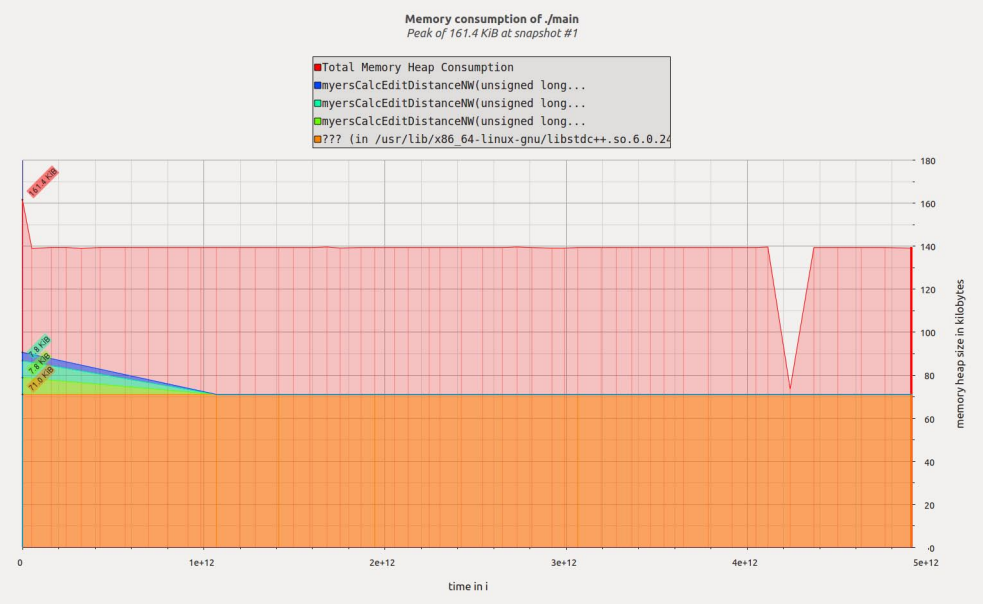

Figure 9.15: Memory utilization of exact edit distance algorithm (Edlib ED) combined with Edlib (path) read aligner while evaluating set_12 for an edit distance threshold of 25.

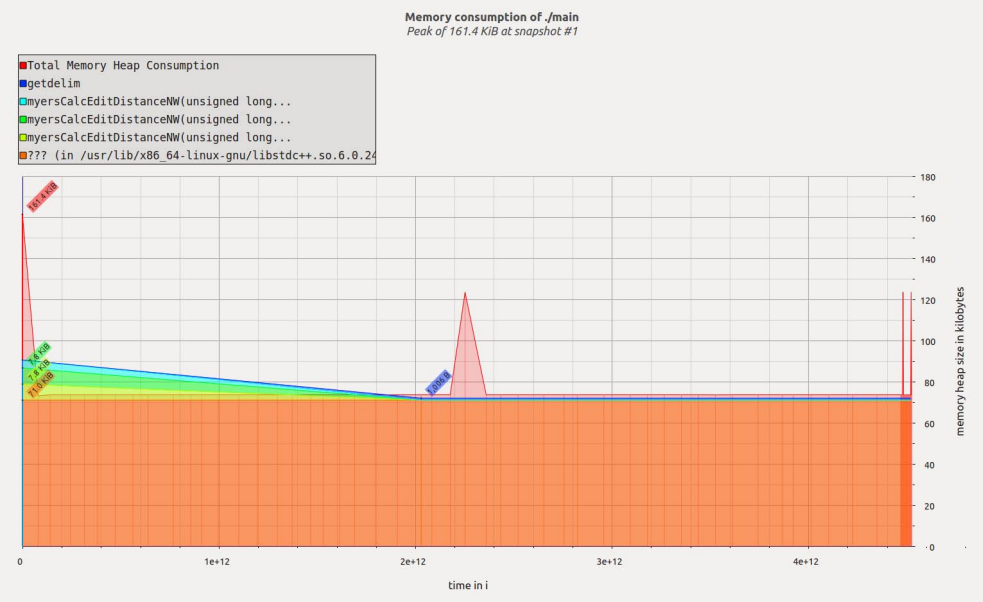

Figure 9.16: Memory utilization of SneakySnake-5 combined with Edlib (path) read aligner while evaluating set_12 for an edit distance threshold of 25.

Chapter 10

Conclusions and Future Directions

Our goal in this thesis is to considerably minimize the time spent on calculating the optimal alignment in genome analysis, given limited computational resources (i.e., personal computer or small hardware). To this end, we first provide a comprehensive accuracy analysis of the pre-alignment filtering. Understanding the causes for the filtering inaccuracy helps us to design new fast and accurate pre-alignment filters. Second, we propose the first hardware accelerator architecture for pre-alignment in genome read mapping. We leverage the large number of filtering units that our hardware accelerator offers for accelerating our proposed hardware-aware algorithms. We propose four hardware pre-alignment filters, GateKeeper, SLIDER, MAGNET, and SneakySnake. In our experimental evaluation, our hardware pre-alignment filters show, on average, three orders of magnitude speedup over their equivalent CPU implementations. We demonstrate that GateKeeper occupies the least percentage of the FPGA resource and it is the least accurate filter. We show that MAGNET provides a low false accept rate but incurs a very low rate of falsely rejected mappings. We also demonstrate that SLIDER is more accurate than GateKeeper and faster than MAGNET. However, SneakySnake is our best performing pre-alignment filter in terms of both speed and accuracy.

SneakySnake has a very low false accept rate and 0% false reject rate. We demonstrate that SneakySnake reduces the execution time of existing read aligners by up to an order of magnitude.

Third, we introduce a fast and cost-effective CPU implementation of our best performing pre-alignment filter, SneakySnake. In our comprehensive evaluation, we demonstrate that the CPU implementation of SneakySnake reduces the execution time of the best performing read aligner, Edlib and Parasail, by up to 43x and 57.9x, respectively. We also experimentally demonstrate that SneakySnake has 50% less memory footprint compared with that of Edlib. The CPU implementation of SneakySnake obviates the need for costly hardware and high hardware design efforts by providing a fast and cost-effective implementation.

We demonstrate that the concept of pre-alignment filtering provides substantial benefits to the existing and future read alignment algorithms. New accelerated sequence aligners are frequently introduced that offer different strengths and features. Our proposed pre-alignment filters offer the ability to accelerate existing aligners by an order of magnitude without sacrificing any of their capabilities and features. As such, we hope that it catalyzes the adoption of our proposed pre-alignment filters in genome sequence analysis, which are becoming increasingly necessary to cope with the processing requirements of greatly increasing amounts of genomic data.

10.1 Future Research Directions

This thesis opens up several avenues of future research directions. In this section, we describe five directions based on the ideas and approaches proposed in this thesis. These ideas can lead to a new read mapper or improve existing ones, which we will explore for various mappers in our future research.

1. The first potential target of our research is to influence the design of more intelligent and attractive sequencing machines by integrating SneakySnake or SLIDER inside them, to perform a real-time pre-alignment filtering. Sequencing machines (e.g., Illumina HiSeq 2500, HiSeq 4000 and MiSeq) are equipped with FPGA chips for accelerating their internal computations. Integrating our pre-alignment filters with the sequencing machine has two benefits. First, it allows a significant reduction in the total execution time of genome analysis by starting read mapping while still sequencing [130]. Second, it can hide the complexity and details of the underlying hardware from users who are not necessarily fluent in FPGAs.

2. Cloud-enabled pre-alignment filtering in a pay-per-use fashion is also a promising solution to encourage the use of such pre-alignment filters and address the concern of lack of experience in dealing with hardware designs. Cloud computing offers access to a large number of advanced FPGA chips that can be used concurrently via a simple user-friendly interface. However, such scenario requires the development of privacy-preserving pre-alignment filters due to privacy and legal concerns related to the share of sensitive genome data to a third party [131]. Our next efforts will focus on exploring privacy-preserving real-time pre-alignment filtering.

3. Since a single-core SneakySnake/GateKeeper/SLIDER has only a small footprint on the FPGA, we can combine our architecture with any of the FPGA-based accelerators for BWT-FM or hash-based mapping techniques on a single FPGA chip. With such a combination, the end result would be an efficient and fast multi-layer mapping system: alignments that pass our pre-alignment filter can be further verified using a dynamic programing based alignment algorithm within the same chip.

4. To further improve the performance of our SneakySnake pre-alignment filter, we can take full advantage of the redundancy across both reference genome and reads present in large sequencing data sets and store the optimal path for small subsequences of the read and the reference sequences. We later use the pre-computed path towards calculating the approximate edit distance quickly.

This requires the understanding of the trade-offs between the memory footprint of seed filtering and the speed of pre-alignment filtering. Another approach for accelerating our SneakySnake algorithm is to exploit GPUs or multi-threaded processors to achieve a highly parallel implementation.

5. We also aim to explore the possibility of achieving full alignment step in linear time complexity in term of read length. We believe that we can achieve this challenging goal by improving the accuracy of DFS algorithm in SneakySnake. If we penalize each potential path based on its distance from the main diagonal, we can eventually infer the the exact number of edit distance along the entire path. This leads to not only calculate the edit distance accurately, but also to on-the-fly provide the optimal alignment without performing "backtracking" by printing the snake's path. The path contains all needed information such as the location, the number, and the type of the edits involved.

Bibliography

[1] J. Arram, K. H. Tsoi, W. Luk, and P. Jiang, "Reconfigurable acceleration of short read mapping," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pp. 210–217, IEEE, 2013.

[2] E. J. Houtgast, V.-M. Sima, K. Bertels, and Z. Al-Ars, "An fpga-based systolic array to accelerate the bwa-mem genomic mapping algorithm," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pp. 221–227, IEEE, 2015.

[3] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, "Hardware acceleration of short read mapping," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pp. 161–168, IEEE, 2012.

[4] Y. Sogabe and T. Maruyama, "Fpga acceleration of short read mapping based on sort and parallel comparison," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pp. 1–4, IEEE, 2014.

[5] H. M. Waidyasooriya, M. Hariyama, and M. Kameyama, "Fpga-accelerator for dna sequence alignment based on an efficient data-dependent memory access scheme," *Highly-Efficient Accelerators and Reconfigurable Technologies*, pp. 127–130, 2014.

[6] H. M. Waidyasooriya and M. Hariyama, "Hardware-acceleration of short-read alignment based on the burrows-wheeler transform," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1358–1372, 2016.

[7] H. Xin, J. Greth, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu, "Shifted hamming distance: a fast and accurate simd-friendly filter to accelerate alignment verification in read mapping," *Bioinformatics*, vol. 31, no. 10, pp. 1553–1560, 2015.

[8] M. Šošić and M. Šikić, "Edlib: a c/c++ library for fast, exact sequence alignment using edit distance," *Bioinformatics*, vol. 33, no. 9, pp. 1394–1395, 2017.

[9] J. Daily, "Parasail: Simd c library for global, semi-global, and local pairwise sequence alignments," *BMC bioinformatics*, vol. 17, no. 1, p. 81, 2016.

[10] J. M. Lane, J. Liang, I. Vlasac, S. G. Anderson, D. A. Bechtold, J. Bowden, R. Emsley, S. Gill, M. A. Little, A. I. Luik, *et al.*, "Genome-wide association analyses of sleep disturbance traits identify new loci and highlight shared genetics with neuropsychiatric and metabolic traits," *Nature genetics*, vol. 49, no. 2, p. 274, 2017.

[11] G. Dobigny, J. Britton-Davidian, and T. J. Robinson, "Chromosomal polymorphism in mammals: an evolutionary perspective," *Biological Reviews*, vol. 92, no. 1, pp. 1–21, 2017.

[12] P. H. Sudmant, T. Rausch, E. J. Gardner, R. E. Handsaker, A. Abyzov, J. Huddleston, Y. Zhang, K. Ye, G. Jun, M. H.-Y. Fritz, *et al.*, "An integrated map of structural variation in 2,504 human genomes," *Nature*, vol. 526, no. 7571, p. 75, 2015.

[13] A. Korte and A. Farlow, "The advantages and limitations of trait analysis with gwas: a review," *Plant methods*, vol. 9, no. 1, p. 29, 2013.

[14] P. M. Visscher, M. A. Brown, M. I. McCarthy, and J. Yang, "Five years of gwas discovery," *The American Journal of Human Genetics*, vol. 90, no. 1, pp. 7–24, 2012.

[15] F. Antonacci, J. M. Kidd, T. Marques-Bonet, M. Ventura, P. Siswara, Z. Jiang, and E. E. Eichler, "Characterization of six human disease-associated inversion polymorphisms," *Human molecular genetics*, vol. 18, no. 14, pp. 2555–2566, 2009.

[16] P. K. Han, K. L. Umstead, B. A. Bernhardt, R. C. Green, S. Joffe, B. Koenig, I. Krantz, L. B. Waterston, L. G. Biesecker, and B. B. Biesecker, "A taxonomy of medical uncertainties in clinical genome sequencing," *Genetics in Medicine*, vol. 19, no. 8, p. 918, 2017.

[17] P. M. Visscher, N. R. Wray, Q. Zhang, P. Sklar, M. I. McCarthy, M. A. Brown, and J. Yang, "10 years of gwas discovery: biology, function, and translation," *The American Journal of Human Genetics*, vol. 101, no. 1, pp. 5–22, 2017.

[18] L. Chin, J. N. Andersen, and P. A. Futreal, "Cancer genomics: from discovery science to personalized medicine," *Nature medicine*, vol. 17, no. 3, p. 297, 2011.

[19] L. Ding, M. C. Wendl, D. C. Koboldt, and E. R. Mardis, "Analysis of next-generation genomic data in cancer: accomplishments and challenges," *Human molecular genetics*, vol. 19, no. R2, pp. R188–R196, 2010.

[20] D. E. Pritchard, F. Moeckel, M. S. Villa, L. T. Housman, C. A. McCarty, and H. L. McLeod, "Strategies for integrating personalized medicine into healthcare practice," *Personalized Medicine*, vol. 14, no. 2, pp. 141–152, 2017.

[21] E. D. Esplin, L. Oei, and M. P. Snyder, "Personalized sequencing and the future of medicine: discovery, diagnosis and defeat of disease," *Pharmacogenomics*, vol. 15, no. 14, pp. 1771–1790, 2014.

[22] M. L. Metzker, "Sequencing technologies—the next generation," *Nature reviews genetics*, vol. 11, no. 1, p. 31, 2010.

[23] M. A. Hamburg and F. S. Collins, "The path to personalized medicine," *New England Journal of Medicine*, vol. 363, no. 4, pp. 301–304, 2010.

[24] G. S. Ginsburg and J. J. McCarthy, "Personalized medicine: revolutionizing drug discovery and patient care," *TRENDS in Biotechnology*, vol. 19, no. 12, pp. 491–496, 2001.

[25] J. A. Reuter, D. V. Spacek, and M. P. Snyder, "High-throughput sequencing technologies," *Molecular cell*, vol. 58, no. 4, pp. 586–597, 2015.

[26] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Molecular Biology*, vol. 147, pp. 195–197, 1981.

[27] H. Cheng, H. Jiang, J. Yang, Y. Xu, and Y. Shang, "Bitmapper: an efficient all-mapper based on bit-vector computing," *BMC bioinformatics*, vol. 16, no. 1, p. 192, 2015.

[28] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating read mapping with fasthash," in *BMC genomics*, vol. 14, p. S13, BioMed Central, 2013.

[29] F. Hach, F. Hormozdiari, C. Alkan, F. Hormozdiari, I. Birol, E. E. Eichler, and S. C. Sahinalp, "mrsfast: a cache-oblivious algorithm for short-read mapping," *Nature methods*, vol. 7, no. 8, p. 576, 2010.

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

[31] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature methods*, vol. 9, no. 4, p. 357, 2012.

[32] R. Luo, T. Wong, J. Zhu, C.-M. Liu, X. Zhu, E. Wu, L.-K. Lee, H. Lin, W. Zhu, D. W. Cheung, *et al.*, "Soap3-dp: fast, accurate and sensitive gpu-based short read aligner," *PloS one*, vol. 8, no. 5, p. e65632, 2013.

[33] G. Navarro, "A guided tour to approximate string matching," *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

[34] J. S. Kim, D. S. Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "Grim-filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies," *BMC Genomics*, vol. 19, no. 2, p. 89, 2018.

[35] S. Marco-Sola, M. Sammeth, R. Guigó, and P. Ribeca, "The gem mapper: fast, accurate and versatile alignment by filtration," *Nature methods*, vol. 9, no. 12, p. 1185, 2012.

[36] E. J. Fox, K. S. Reid-Bayliss, M. J. Emond, and L. A. Loeb, "Accuracy of next generation sequencing platforms," *Next generation, sequencing & applications*, vol. 1, 2014.

[37] K. J. McKernan, H. E. Peckham, G. L. Costa, S. F. McLaughlin, Y. Fu, E. F. Tsung, C. R. Clouser, C. Duncan, J. K. Ichikawa, C. C. Lee, *et al.*, "Sequence and structural variation in a human genome uncovered by short-read, massively parallel ligation sequencing using two-base encoding," *Genome research*, vol. 19, no. 9, pp. 1527–1541, 2009.

[38] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, pp. 707–710, 1966.

[39] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[40] S. Canzar and S. L. Salzberg, "Short read mapping: an algorithmic tour," *Proceedings of the IEEE*, vol. 105, no. 3, pp. 436–458, 2017.

[41] M. Escalona, S. Rocha, and D. Posada, "A comparison of tools for the simulation of genomic next-generation sequencing data," *Nature Reviews Genetics*, vol. 17, no. 8, p. 459, 2016.

[42] M. A. Eberle, E. Fritzilas, P. Krusche, M. Källberg, B. L. Moore, M. A. Bekritsky, Z. Iqbal, H.-Y. Chuang, S. J. Humphray, A. L. Halpern, *et al.*, "A reference data set of 5.4 million phased human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree," *Genome research*, vol. 27, no. 1, pp. 157–164, 2017.

[43] G. X. Zheng, B. T. Lau, M. Schnall-Levin, M. Jarosz, J. M. Bell, C. M. Hindson, S. Kyriazopoulou-Panagiotopoulou, D. A. Masquelier, L. Merrill,

J. M. Terry, *et al.*, "Haplotyping germline and cancer genomes with high-throughput linked-read sequencing," *Nature biotechnology*, vol. 34, no. 3, p. 303, 2016.

[44] M. Griffith, C. A. Miller, O. L. Griffith, K. Krysiak, Z. L. Skidmore, A. Ramu, J. R. Walker, H. X. Dang, L. Trani, D. E. Larson, *et al.*, "Optimizing cancer genome sequencing and analysis," *Cell systems*, vol. 1, no. 3, pp. 210–223, 2015.

[45] I. Iossifov, B. J. O'roak, S. J. Sanders, M. Ronemus, N. Krumm, D. Levy, H. A. Stessman, K. T. Witherspoon, L. Vives, K. E. Patterson, *et al.*, "The contribution of de novo coding mutations to autism spectrum disorder," *Nature*, vol. 515, no. 7526, p. 216, 2014.

[46] M. Meyerson, S. Gabriel, and G. Getz, "Advances in understanding cancer genomes through second-generation sequencing," *Nature Reviews Genetics*, vol. 11, no. 10, p. 685, 2010.

[47] S. K. Delaney, M. L. Hultner, H. J. Jacob, D. H. Ledbetter, J. J. McCarthy, M. Ball, K. B. Beckman, J. W. Belmont, C. S. Bloss, M. F. Christman, *et al.*, "Toward clinical genomics in everyday medicine: perspectives and recommendations," *Expert review of molecular diagnostics*, vol. 16, no. 5, pp. 521–532, 2016.

[48] L. Pérez-Lago, M. Martínez-Lirola, S. García, M. Herranz, I. Mokrousov, I. Comas, L. Martínez-Priego, E. Bouza, and D. García-de Viedma, "Urgent implementation in a hospital setting of a strategy to rule out secondary cases caused by imported extensively drug-resistant mycobacterium tuberculosis strains at diagnosis," *Journal of clinical microbiology*, vol. 54, no. 12, pp. 2969–2974, 2016.

[49] S. F. Kingsmore, J. Petrikin, L. K. Willig, and E. Guest, "Emergency medical genomes: a breakthrough application of precision medicine," *Genome medicine*, vol. 7, no. 1, p. 82, 2015.

[50] L. K. Willig, J. E. Petrikin, L. D. Smith, C. J. Saunders, I. Thiffault, N. A. Miller, S. E. Soden, J. A. Cakici, S. M. Herd, G. Twist, *et al.*, "Whole-genome sequencing for identification of mendelian disorders in critically ill infants: a retrospective analysis of diagnostic and clinical findings," *The Lancet Respiratory Medicine*, vol. 3, no. 5, pp. 377–387, 2015.

[51] N. M. Sweeney, S. A. Nahas, S. Chowdhury, M. Del Campo, M. C. Jones, D. P. Dimmock, S. F. Kingsmore, R. Investigators, *et al.*, "The case for early use of rapid whole genome sequencing in management of critically ill infants: Late diagnosis of coffin-siris syndrome in an infant with left congenital diaphragmatic hernia, congenital heart disease and recurrent infections," *Molecular Case Studies*, pp. mcs–a002469, 2018.

[52] L. Farnaes, A. Hildreth, N. M. Sweeney, M. M. Clark, S. Chowdhury, S. Nahas, J. A. Cakici, W. Benson, R. H. Kaplan, R. Kronick, *et al.*, "Rapid whole-genome sequencing decreases infant morbidity and cost of hospitalization," *NPJ genomic medicine*, vol. 3, no. 1, p. 10, 2018.

[53] J. S. Berg, L. M. Amendola, C. Eng, E. Van Allen, S. W. Gray, N. Wagle, H. L. Rehm, E. T. DeChene, M. C. Dulik, F. M. Hisama, *et al.*, "Processes and preliminary outputs for identification of actionable genes as incidental findings in genomic sequence data in the clinical sequencing exploratory research consortium," *Genetics in Medicine*, vol. 15, no. 11, p. 860, 2013.

[54] C. R. Ferreira, D. S. Regier, D. W. Hadley, P. S. Hart, and M. Muenke, "Medical genetics and genomic medicine in the united states of america. part 1: history, demographics, legislation, and burden of disease," *Molecular genetics & genomic medicine*, vol. 5, no. 4, pp. 307–316, 2017.

[55] S. S. Banerjee, M. El-Hadedy, J. B. Lim, Z. T. Kalbarczyk, D. Chen, S. Lumetta, and R. K. Iyer, "Asap: Accelerated short-read alignment on programmable hardware," *arXiv preprint arXiv:1803.02657*, 2018.

[56] X. Fei, Z. Dan, L. Lina, M. Xin, and Z. Chunlei, "Fpgasw: Accelerating large-scale smith–waterman sequence alignment application with backtracking on fpga linear systolic array," *Interdisciplinary Sciences: Computational Life Sciences*, vol. 10, no. 1, pp. 176–188, 2018.

[57] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "meraligner: A fully parallel sequence aligner," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pp. 561–570, IEEE, 2015.

[58] Y. Liu and B. Schmidt, "Gswabe: faster gpu-accelerated sequence alignment with optimal alignment retrieval for short dna sequences," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 4, pp. 958–972, 2015.

[59] Y. Liu, A. Wirawan, and B. Schmidt, "Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions," *BMC bioinformatics*, vol. 14, no. 1, p. 117, 2013.

[60] J. Arram, K. H. Tsoi, W. Luk, and P. Jiang, "Hardware acceleration of genetic sequence alignment," in *International Symposium on Applied Reconfigurable Computing*, pp. 13–24, Springer, 2013.

[61] D. Weese, M. Holtgrewe, and K. Reinert, "Razers 3: faster, fully sensitive read mapping," *Bioinformatics*, vol. 28, no. 20, pp. 2592–2599, 2012.

[62] A. Ahmadi, A. Behm, N. Honnalli, C. Li, L. Weng, and X. Xie, "Hobbes: optimized gram-based methods for efficient read alignment," *Nucleic acids research*, vol. 40, no. 6, pp. e41–e41, 2011.

[63] G. Rizk and D. Lavenier, "Gassst: global alignment short sequence search tool," *Bioinformatics*, vol. 26, no. 20, pp. 2534–2540, 2010.

[64] D. Weese, A.-K. Emde, T. Rausch, A. Döring, and K. Reinert, "Razers—fast read mapping with sensitivity control," *Genome research*, vol. 19, no. 9, pp. 1646–1654, 2009.

[65] T. Nishimura, J. L. Bordim, Y. Ito, and K. Nakano, "Accelerating the smith-waterman algorithm using bitwise parallel bulk computation technique on gpu," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pp. 932–941, IEEE, 2017.

[66] P. Chen, C. Wang, X. Li, and X. Zhou, "Accelerating the next generation long read mapping with the fpga-based system," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 11, no. 5, pp. 840–852, 2014.

[67] R. C. Edgar and K. Sjölander, "A comparison of scoring functions for protein sequence profile alignment," *Bioinformatics*, vol. 20, no. 8, pp. 1301–1308, 2004.

[68] H. Xin, S. Nahar, R. Zhu, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu, "Optimal seed solver: optimizing seed selection in read mapping," *Bioinformatics*, vol. 32, no. 11, pp. 1632–1642, 2015.

[69] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "Gatekeeper: a new hardware architecture for accelerating pre-alignment in dna short read mapping," *Bioinformatics*, vol. 33, no. 21, pp. 3355–3363, 2017.

[70] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "Gatekeeper: A new hardware architecture for accelerating pre-alignment in dna short read mapping," *arXiv preprint arXiv:1604.01789*, 2016.

[71] M. Alser, O. Mutlu, and C. Alkan, "Magnet: Understanding and improving the accuracy of genome pre-alignment filtering," *Transactions on Internet Research*, vol. 13, no. 2, pp. 33–42, 2017.

[72] M. Alser, O. Mutlu, and C. Alkan, "Magnet: Understanding and improving the accuracy of genome pre-alignment filtering," *arXiv preprint arXiv:1707.01631*, 2017.

[73] M. Alser, H. Hassan, A. Kumar, O. Mutlu, and C. Alkan, "Exploring speed/accuracy trade-offs in hardware accelerated pre-alignment in genome analysis," in *HPCA2018 Workshop on Accelerator Architecture in Computational Biology and Bioinformatics (AACBB), , Vienna, Austria, February 24th, 2018. https://aacbb-workshop.github.io/*, 2018.

[74] P. Weiner, "Linear pattern matching algorithms," in *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pp. 1–11, IEEE, 1973.

[75] M. T. Özsu and P. Valduriez, *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[76] M. BURROWS, "A block-sorting lossless data compression algorithm," *SRC Research Report, 124*, 1994.

[77] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pp. 390–398, IEEE, 2000.

[78] S. M. Rumble, P. Lacroute, A. V. Dalca, M. Fiume, A. Sidow, and M. Brudno, "Shrimp: accurate mapping of short color-space reads," *PLoS computational biology*, vol. 5, no. 5, p. e1000386, 2009.

[79] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[80] H. Li and R. Durbin, "Fast and accurate long-read alignment with burrows–wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, 2010.

[81] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome biology*, vol. 10, no. 3, p. R25, 2009.

[82] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, "Soap2: an improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 25, no. 15, pp. 1966–1967, 2009.

[83] C.-M. Liu, T. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, *et al.*, "Soap3: ultra-fast gpu-based parallel alignment tool for short reads," *Bioinformatics*, vol. 28, no. 6, pp. 878–879, 2012.

[84] C. Firtina and C. Alkan, "On genomic repeats and reproducibility," *Bioinformatics*, vol. 32, no. 15, pp. 2243–2247, 2016.

[85] I. Medina, J. Tárraga, H. Martínez, S. Barrachina, M. Castillo, J. Paschall, J. Salavert-Torres, I. Blanquer-Espert, V. Hernández-García, E. Quintana-Ortí, *et al.*, "Highly sensitive and ultrafast read mapping for rna-seq analysis," *DNA Research*, vol. 23, no. 2, pp. 93–100, 2016.

[86] R. Li, Y. Li, K. Kristiansen, and J. Wang, "Soap: short oligonucleotide alignment program," *Bioinformatics*, vol. 24, no. 5, pp. 713–714, 2008.

[87] H. Li, J. Ruan, and R. Durbin, "Mapping short dna sequencing reads and calling variants using mapping quality scores," *Genome research*, vol. 18, no. 11, pp. 1851–1858, 2008.

[88] A. D. Smith, Z. Xuan, and M. Q. Zhang, "Using quality scores and longer reads improves accuracy of solexa read mapping," *BMC bioinformatics*, vol. 9, no. 1, p. 128, 2008.

[89] H. Lin, Z. Zhang, M. Q. Zhang, B. Ma, and M. Li, "Zoom! zillions of oligos mapped," *Bioinformatics*, vol. 24, no. 21, pp. 2431–2437, 2008.

[90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[91] N. Homer, B. Merriman, and S. F. Nelson, "Bfast: an alignment tool for large scale genome resequencing," *PloS one*, vol. 4, no. 11, p. e7767, 2009.

[92] M. David, M. Dzamba, D. Lister, L. Ilie, and M. Brudno, "Shrimp2: sensitive yet practical short read mapping," *Bioinformatics*, vol. 27, no. 7, pp. 1011–1012, 2011.

[93] F. Hormozdiari, F. Hach, S. C. Sahinalp, E. E. Eichler, and C. Alkan, "Sensitive and fast mapping of di-base encoded reads," *Bioinformatics*, vol. 27, no. 14, pp. 1915–1921, 2011.

[94] W.-P. Lee, M. P. Stromberg, A. Ward, C. Stewart, E. P. Garrison, and G. T. Marth, "Mosaik: a hash-based algorithm for accurate next-generation sequencing short-read mapping," *PloS one*, vol. 9, no. 3, p. e90581, 2014.

[95] J. Blom, T. Jakobi, D. Doppmeier, S. Jaenicke, J. Kalinowski, J. Stoye, and A. Goesmann, "Exact and complete short-read alignment to microbial genomes using graphics processing unit programming," *Bioinformatics*, vol. 27, no. 10, pp. 1351–1358, 2011.

[96] E. Ukkonen, "Algorithms for approximate string matching," *Information and control*, vol. 64, no. 1-3, pp. 100–118, 1985.

[97] D. Yorukoglu, Y. W. Yu, J. Peng, and B. Berger, "Compressive mapping for next-generation sequencing," *Nature biotechnology*, vol. 34, no. 4, p. 374, 2016.

[98] S. M. Kiełbasa, R. Wan, K. Sato, P. Horton, and M. C. Frith, "Adaptive seeds tame genomic sequence comparison," *Genome research*, vol. 21, no. 3, pp. 487–493, 2011.

[99] L. Egidi and G. Manzini, "Better spaced seeds using quadratic residues," *Journal of Computer and System Sciences*, vol. 79, no. 7, pp. 1144–1155, 2013.

[100] R. Lederman, "A random-permutations-based approach to fast read alignment," in *BMC bioinformatics*, vol. 14, p. S8, BioMed Central, 2013.

[101] C. Calude, K. Salomaa, and S. Yu, "Additive distances and quasi-distances between words," *Journal of Universal Computer Science*, vol. 8, no. 2, pp. 141–152, 2002.

[102] W. J. Masek and M. S. Paterson, "A faster algorithm computing string edit distances," *Journal of Computer and System Sciences*, vol. 20, no. 1, pp. 18–31, 1980.

[103] A. Backurs and P. Indyk, "Edit distance cannot be computed in strongly subquadratic time (unless seth is false)," *arXiv preprint arXiv:1412.0348v4*, 2017.

[104] A. Al Kawam, S. Khatri, and A. Datta, "A survey of software and hardware approaches to performing read alignment in next generation sequencing,"

IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB), vol. 14, no. 6, pp. 1202–1213, 2017.

[105] S. Aluru and N. Jammula, "A review of hardware acceleration for computational genomics," *IEEE Design & Test*, vol. 31, no. 1, pp. 19–30, 2014.

[106] H.-C. Ng, S. Liu, and W. Luk, "Reconfigurable acceleration of genetic sequence alignment: A survey of two decades of efforts," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pp. 1–8, IEEE, 2017.

[107] E. F. D. O. Sandes, A. Boukerche, and A. C. M. A. D. Melo, "Parallel optimal pairwise biological sequence comparison: Algorithms, platforms, and classification," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, p. 63, 2016.

[108] H. Kung, "Why systolic architectures?," *Computer*, vol. 15, no. 1, pp. 37–46, 1982.

[109] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, and P. Wei, "When apache spark meets fpgas: a case study for next-generation dna sequencing acceleration," in *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, pp. 64–70, USENIX Association, 2016.

[110] Y.-T. Chen and J. Cong, "Cs-bwamem: A fast and scalable read aligner at the cloud scale for whole genome sequencing," in *Proceedings of High Throughput Sequencing Algorithms and Applications (HITSEQ)*, 2015.

[111] C. Wang, R.-X. Yan, X.-F. Wang, J.-N. Si, and Z. Zhang, "Comparison of linear gap penalties and profile-based variable gap penalties in profile–profile alignments," *Computational biology and chemistry*, vol. 35, no. 5, pp. 308–318, 2011.

[112] S. Henikoff and J. G. Henikoff, "Amino acid substitution matrices from protein blocks," *Proceedings of the National Academy of Sciences*, vol. 89, no. 22, pp. 10915–10919, 1992.

[113] A. Hatem, D. Bozdağ, A. E. Toland, and Ü. V. Çatalyürek, "Benchmarking short sequence mapping tools," *BMC bioinformatics*, vol. 14, no. 1, p. 184, 2013.

[114] M. C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello, "Achieving high performance with fpga-based computing," *Computer*, vol. 40, no. 3, 2007.

[115] S. M. Trimberger, "Three ages of fpgas: A retrospective on the first thirty years of fpga technology," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318–331, 2015.

[116] N. McVICAR, A. Hoshino, A. La Torre, T. A. Reh, W. L. Ruzzo, and S. Hauck, "Fpga acceleration of short read alignment," *arXiv preprint arXiv:1805.00106*, 2018.

[117] N. Alachiotis, D. Theodoropoulos, and D. Pnevmatikatos, "Versatile deployment of fpga accelerators in disaggregated data centers: A bioinformatics case study," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pp. 1–4, IEEE, 2017.

[118] X. Virtex, "Fpga vc709 connectivity kit," *Xilinx: All Programmable*, 7.

[119] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "Riffa 2.1: A reusable integration framework for fpga accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 4, p. 22, 2015.

[120] U. Guide, "Series fpgas configurable logic block," *Xilinx, San Jose, CA*, vol. 1, 7.

[121] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 273–287, ACM, 2017.

[122] D. J. Lipman and W. R. Pearson, "Rapid and sensitive protein similarity searches," *Science*, vol. 227, no. 4693, pp. 1435–1441, 1985.

[123] J. W. J. Williams, "Algorithm 232: heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, 1964.

[124] M. McNamara *et al.*, "Ieee standard verilog hardware description language. the institute of electrical and electronics engineers," *Inc. IEEE Std*, pp. 1364–2001, 2001.

[125] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[126] G. Dimitrakopoulos, K. Galanopoulos, C. Mavrokefalidis, and D. Nikolos, "Low-power leading-zero counting and anticipation logic for high-speed floating point units," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 16, no. 7, pp. 837–850, 2008.

[127] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, *et al.*, "Personalized copy number and segmental duplication maps using next-generation sequencing," *Nature genetics*, vol. 41, no. 10, p. 1061, 2009.

[128] . G. P. Consortium *et al.*, "An integrated map of genetic variation from 1,092 human genomes," *Nature*, vol. 491, no. 7422, p. 56, 2012.

[129] R. W. Hamming, "Error detecting and error correcting codes," *Bell Labs Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[130] M. S. Lindner, B. Strauch, J. M. Schulze, S. H. Tausch, P. W. Dabrowski, A. Nitsche, and B. Y. Renard, "Hilive: real-time mapping of illumina reads while sequencing," *Bioinformatics*, vol. 33, no. 6, pp. 917–319, 2016.

[131] S. Salinas and P. Li, "Secure cloud computing for pairwise sequence alignment," in *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pp. 178–183, ACM, 2017.

Appendix A

Data

In this chapter, we use Edlib [8] (edit distance mode) to assess the number of accepted (i.e., having edits less or equal to the edit distance threshold) and rejected (i.e., having more edits than the edit distance threshold) pairs for each of the 12 datasets. We also provide the time, in seconds, that is needed for Edlib to complete clustering the pairs. We provide these details for set_1, set_2, set_3, and set_4 in Table A.1. We also provide the same details for set_5, set_6, set_7, and set_8 in Table A.2 and for set_9, set_10, set_11, and set_12 in Table A.3.

Next, we provide the effect of reducing the search space of our SneakySnake algorithm and the best exact edit distance algorithm, Edlib [8]. We use the following datasets set_1, set_2, set_3, and set_4 in Figure A.4. We also use these datasets set_5, set_6, set_7, and set_8 in Figure A.5. Finally, in Figure A.6, we use the following datasets set_9, set_10, set_11, and set_12.

Table A.1: Details of our first four datasets (set_1, set_2, set_3, and set_4). We use Edlib [8] (edit distance mode) to benchmark the accepted and the rejected pairs for edit distance thresholds of 0 up to 10 edits.

Dataset	*E*	Time (sec)	Accepted	Rejected	Dataset	*E*	Time (sec)	Accepted	Rejected
ERR240727_1, mrFAST -e=2, 30 million pairs	0	110.61	381,901	29,618,099	ERR240727_1, mrFAST -e=3, 30 million pairs	0	120.84	124,531	29,875,469
	1	135.30	1,345,842	28,654,158		1	121.67	441,927	29,558,073
	2	126.87	3,266,455	26,733,545		2	121.33	1,073,808	28,926,192
	3	125.46	5,595,596	24,404,404		3	117.88	2,053,181	27,946,819
	4	122.09	7,825,272	22,174,728		4	113.79	3,235,057	26,764,943
	5	121.73	9,821,308	20,178,692		5	115.00	4,481,341	25,518,659
	6	126.26	11,650,490	18,349,510		6	115.26	5,756,432	24,243,568
	7	124.86	13,407,801	16,592,199		7	116.10	7,091,373	22,908,627
	8	126.83	15,152,501	14,847,499		8	116.71	8,531,811	21,468,189
	9	122.94	16,894,680	13,105,320		9	117.47	10,102,726	19,897,274
	10	122.60	18,610,897	11,389,103		10	118.49	11,807,488	18,192,512
Dataset	*E*	Time (sec)	Accepted	Rejected	Dataset	*E*	Time (sec)	Accepted	Rejected
ERR240727_1, mrFAST -e=5, 30 million pairs	0	141.01	11,989	29,988,011	ERR240727_1, mrFAST -e=40, 30 million pairs	0	124.32	11	29,999,989
	1	130.52	44,565	29,955,435		1	115.18	18	29,999,982
	2	123.52	108,979	29,891,021		2	110.56	24	29,999,976
	3	125.81	206,903	29,793,097		3	111.59	27	29,999,973
	4	120.18	334,712	29,665,288		4	114.13	29	29,999,971
	5	120.08	490,670	29,509,330		5	113.80	34	29,999,966
	6	119.60	675,357	29,324,643		6	115.74	83	29,999,917
	7	124.48	891,447	29,108,553		7	116.70	177	29,999,823
	8	125.64	1,151,447	28,848,553		8	118.12	333	29,999,667
	9	123.18	1,469,996	28,530,004		9	120.11	711	29,999,289
	10	131.24	1,868,827	28,131,173		10	123.42	1,627	29,998,373

Table A.2: Details of our second four datasets (set_5, set_6, set_7, and set_8). We use Edlib [8] (edit distance mode) to benchmark the accepted and the rejected pairs for edit distance thresholds of 0 up to 15 edits.

Dataset	*E*	Time (sec)	Accepted	Rejected	Dataset	*E*	Time (sec)	Accepted	Rejected
SRR826460_1, mrFAST -e=4, 30 million pairs	0	202.82	1,440,497	28,559,503	SRR826460_1, mrFAST -e=6, 30 million pairs	0	185.55	248,920	29,751,080
	1	170.93	1,868,909	28,131,091		1	187.66	324,056	29,675,944
	3	179.84	2,734,841	27,265,159		3	190.54	481,724	29,518,276
	4	185.39	3,457,975	26,542,025		4	195.78	612,747	29,387,253
	6	194.29	5,320,713	24,679,287		6	208.64	991,606	29,008,394
	7	200.69	6,261,628	23,738,372		7	216.06	1,226,695	28,773,305
	9	207.89	7,916,882	22,083,118		9	225.75	1,740,067	28,259,933
	10	208.18	8,658,021	21,341,979		10	227.97	2,009,835	27,990,165
	12	208.92	10,131,849	19,868,151		12	237.14	2,591,299	27,408,701
	13	208.01	10,917,472	19,082,528		13	238.31	2,923,699	27,076,301
	15	208.96	12,646,165	17,353,835		15	233.91	3,730,089	26,269,911
Dataset	*E*	Time (sec)	Accepted	Rejected	Dataset	*E*	Time (sec)	Accepted	Rejected
SRR826460_1, mrFAST -e=10, 30 million pairs	0	293.88	444	29,999,556	SRR826460_1, mrFAST -e=70, 30 million pairs	0	244.79	201	29,999,799
	1	273.39	695	29,999,305		1	248.65	327	29,999,673
	3	269.15	927	29,999,073		3	253.53	444	29,999,556
	4	264.98	994	29,999,006		4	258.86	475	29,999,525
	6	281.14	1,097	29,998,903		6	280.37	529	29,999,471
	7	282.71	1,136	29,998,864		7	286.30	546	29,999,454
	9	287.50	1,221	29,998,779		9	292.03	587	29,999,413
	10	285.02	1,274	29,998,726		10	293.44	612	29,999,388
	12	290.85	1,701	29,998,299		12	301.29	710	29,999,290
	13	291.93	2,146	29,997,854		13	302.68	796	29,999,204
	15	287.24	3,921	29,996,079		15	299.55	1,153	29,998,847

Table A.3: Details of our last four datasets (set_9, set_10, set_11, and set_12). We use Edlib [8] (edit distance mode) to benchmark the accepted and the rejected pairs for edit distance thresholds of 0 up to 25 edits.

Dataset	*E*	Time (sec)	Accepted	Rejected	Dataset	*E*	Time (sec)	Accepted	Rejected
SRR826471_1, mrFAST -e=8, 30 million pairs	0	337.06	707,517	29,292,483	SRR826471_1, mrFAST -e=12, 30 million pairs	0	375.35	43,565	29,956,435
	2	319.08	1,462,242	28,537,758		2	390.71	88,141	29,911,859
	5	340.47	1,973,835	28,026,165		5	449.10	119,100	29,880,900
	7	349.22	2,361,418	27,638,582		7	457.05	145,290	29,854,710
	10	345.73	3,183,271	26,816,729		10	462.65	205,536	29,794,464
	12	339.16	3,862,776	26,137,224		12	464.26	257,360	29,742,640
	15	340.40	4,915,346	25,084,654		15	528.12	346,809	29,653,191
	17	340.23	5,550,869	24,449,131		17	572.97	409,978	29,590,022
	20	341.09	6,404,832	23,595,168		20	521.83	507,177	29,492,823
	22	341.57	6,959,616	23,040,384		22	468.11	572,769	29,427,231
	25	342.30	7,857,750	22,142,250		25	467.72	673,254	29,326,746
Dataset	*E*	Time (sec)	Accepted	Rejected	Dataset	*E*	Time (sec)	Accepted	Rejected
SRR826471_1, mrFAST -e=15, 30 million pairs	0	502.22	4,389	29,995,611	SRR826471_1, mrFAST -e=100, 30 million pairs	0	449.02	49	29,999,951
	2	469.97	8,970	29,991,030		2	467.78	163	29,999,837
	5	505.53	12,420	29,987,580		5	524.47	301	29,999,699
	7	490.44	15,405	29,984,595		7	533.67	375	29,999,625
	10	488.32	22,014	29,977,986		10	612.16	472	29,999,528
	12	482.28	27,817	29,972,183		12	655.45	520	29,999,480
	15	483.46	37,710	29,962,290		15	657.91	575	29,999,425
	17	483.10	44,225	29,955,775		17	547.26	623	29,999,377
	20	484.07	54,650	29,945,350		20	542.69	718	29,999,282
	22	483.88	62,255	29,937,745		22	541.08	842	29,999,158
	25	482.39	74,761	29,925,239		25	544.22	1,133	29,998,867

Table A.4: Details of evaluating the feasibility of reducing the search space for SneakySnake and Edlib, evaluated using set_1, set_2, set_3, and set_4 datasets.

	E	Edlib-100	Edlib-50	Edlib-25	Edlib-10	Edlib-5	S.Snake-100	S.Snake-5		Edlib-100	Edlib-50	Edlib-25	Edlib-10	Edlib-5	S.Snake-100	S.Snake-5
set_1	0	110.61	83.78	93.25	130.88	217.25	12.63	5.78	set_2	120.84	84	85	103.2908	159.0398	13.6411	5.6883
	1	135.30	99.55	130.35	218.46	386.72	25.83	9.76		121.67	88	103	151.2703	255.4318	28.7489	9.0634
	2	126.87	112.30	162.10	299.17	545.09	38.92	15.56		121.33	102	133	219.9173	388.1105	44.2272	14.6858
	3	125.46	128.70	196.53	380.76	704.22	52.76	22.30		117.88	136	188	333.0511	601.9558	59.834	21.9874
	4	122.09	142.29	225.22	449.91	839.38	66.54	30.42		113.79	153	222	410.9619	753.2721	75.7436	31.2187
	5	121.73	155.53	252.52	514.75	964.96	81.36	39.31		115.00	166	249	475.8178	877.5177	92.1749	42.4613
	6	126.26	176.43	292.32	606.17	1,140.25	98.40	48.79		115.26	175	270	531.0017	982.9451	108.7952	54.7952
	7	124.86	174.30	294.39	618.41	1,167.09	114.14	58.23		116.10	178	281	563.1201	1046.0609	124.699	67.3507
	8	126.83	177.64	304.85	647.42	1,225.22	126.22	67.73		116.71	180	290	591.1801	1103.1341	141.5191	80.7123
	9	122.94	185.93	323.59	694.37	1,318.02	142.33	76.61		117.47	198	326	673.6352	1263.0262	159.1982	93.2317
	10	122.60	225.97	397.26	861.54	1,638.95	157.16	84.52		118.49	195	328	686.6502	1292.8982	177.6106	105.0066
	E	Edlib-100	Edlib-50	Edlib-25	Edlib-10	Edlib-5	S.Snake-100	S.Snake-5		Edlib-100	Edlib-50	Edlib-25	Edlib-10	Edlib-5	S.Snake-100	S.Snake-5
set_3	0	141.01	85.57	75.75	72.72	91.05	16.54	5.51	set_4	124.32	65	56	75.2024	109.909	17.7314	5.7039
	1	130.52	88.28	81.68	86.08	120.37	38.08	8.27		115.18	65	57	78.5792	125.3585	44.6246	8.7136
	2	123.52	88.41	86.93	103.20	160.08	60.34	13.72		110.56	65	59	85.1783	143.1387	70.9374	14.2037
	3	125.81	88.03	92.36	118.98	198.59	81.59	21.08		111.59	64	61	93.3215	158.0324	94.9017	21.6675
	4	120.18	75.31	83.86	116.32	203.35	103.47	31.42		114.13	64	65	103.7998	172.8054	119.7748	31.441
	5	120.08	75.23	88.22	131.75	233.25	124.23	43.80		113.80	64	70	114.6424	188.6127	142.9827	43.4782
	6	119.60	78.06	96.21	154.06	271.37	145.29	59.32		115.74	64	76	123.8703	203.6602	166.2669	57.8799
	7	124.48	83.51	107.02	180.35	317.71	166.84	76.68		116.70	63	85	131.1515	218.7594	188.4381	74.6849
	8	125.64	88.65	118.18	205.98	366.12	189.24	96.48		118.12	63	94	138.6343	235.5282	209.9816	94.8629
	9	123.18	93.96	129.70	231.06	416.06	208.91	117.34		120.11	63	101	147.0508	252.9756	232.2404	117.9441
	10	131.24	100.69	142.64	259.35	471.42	230.11	140.24		123.42	63	106	156.874	270.9877	252.5516	143.9834

Table A.5: Details of evaluating the feasibility of reducing the search space for SneakySnake and Edlib, evaluated using set_5, set_6, set_7, and set_8 datasets.

	E	Edlib-150	Edlib-50	Edlib-25	Edlib-10	Edlib-5	S.Snake-150	S.Snake-5		Edlib-150	Edlib-50	Edlib-25	Edlib-10	Edlib-5	S.Snake-150	S.Snake-5
set_5	0	202.82	101.23	121.06	187.98	324.91	17.88	6.16	set_6	185.6	95.1	86.6	98.6	139.4	21.0	5.5
	1	170.93	111.10	147.07	252.10	451.86	42.47	11.29		187.7	97.0	95.9	124.2	193.3	53.4	9.1
	3	179.84	145.72	214.95	408.65	756.09	86.87	23.20		190.5	107.5	123.3	191.8	337.3	112.5	21.9
	4	185.39	166.56	254.30	498.66	930.11	112.19	33.18		195.8	118.0	143.6	238.2	431.5	142.7	32.4
	6	194.29	202.91	323.24	661.09	1,239.52	162.21	59.21		208.6	144.9	191.6	351.1	637.4	203.7	61.0
	7	200.69	238.14	383.74	796.84	1,496.58	187.97	74.22		216.1	153.7	209.9	397.8	722.6	233.4	78.9
	9	207.89	260.26	427.73	904.87	1,706.34	245.30	110.20		225.8	181.8	261.0	512.2	942.3	296.1	123.4
	10	208.18	282.60	467.43	996.83	1,882.48	268.31	130.29		228.0	166.9	243.8	484.7	896.0	329.4	149.4
	12	208.92	319.23	534.98	1,156.46	2,188.61	320.65	167.11		237.1	174.8	263.5	538.7	997.7	385.9	204.6
	13	208.01	367.10	617.98	1,343.92	2,545.30	346.42	185.56		238.3	185.8	284.6	587.0	1,087.7	417.8	234.2
	15	208.96	383.92	656.38	1,441.52	2,736.52	393.62	217.05		233.9	206.6	328.6	681.3	1,267.7	475.2	291.5
	E	Edlib-150	Edlib-50	Edlib-25	Edlib-10	Edlib-5	S.Snake-150	S.Snake-5		Edlib-150	Edlib-50	Edlib-25	Edlib-10	Edlib-5	S.Snake-150	S.Snake-5
set_7	0	293.88	75.77	63.79	73.95	98.33	24.41	5.59	set_8	244.8	88.4	72.7	95.1	133.4	24.2	5.8
	1	273.39	75.30	65.12	77.67	111.52	65.63	8.85		248.7	81.1	67.6	91.8	140.7	66.7	9.9
	3	269.15	72.76	68.23	88.33	140.70	140.52	21.36		253.5	82.4	74.1	111.4	182.4	143.8	21.5
	4	264.98	72.77	72.47	97.78	161.34	178.85	32.01		258.9	81.9	78.8	122.9	201.2	182.6	31.4
	6	281.14	73.09	82.92	117.09	195.48	248.45	60.51		280.4	76.4	89.2	138.1	227.6	253.2	58.4
	7	282.71	74.89	91.64	131.61	220.59	282.43	78.38		286.3	85.0	110.5	164.5	274.8	288.0	75.5
	9	287.50	74.07	103.59	154.10	263.53	347.99	122.58		292.0	76.2	118.2	170.7	293.0	355.8	118.4
	10	285.02	77.82	114.34	173.21	300.71	380.44	149.68		293.4	65.8	106.5	157.9	273.0	389.0	144.2
	12	290.85	77.24	117.97	191.91	338.64	443.12	212.30		301.3	84.5	140.7	224.9	391.4	453.8	205.7
	13	291.93	79.00	121.57	205.33	363.53	473.68	248.05		302.7	92.0	153.3	255.1	446.4	485.7	241.0
	15	287.24	92.59	143.43	255.94	453.83	533.90	328.17		299.5	90.1	149.4	267.1	471.6	548.3	320.7

Table A.6: Details of evaluating the feasibility of reducing the search space for SneakySnake and Edlib, evaluated using set_9, set_10, set_11, and set_12 datasets.

	E	Edlib-150	Edlib-50	Edlib-25	Edlib-10	Edlib-5	S.Snake-150	S.Snake-5		Edlib-150	Edlib-50	Edlib-25	Edlib-10	Edlib-5	S.Snake-150	S.Snake-5
set_9	0	337.1	119.2	149.2	239.5	420.9	30.4	6.0	set_10	375.3	85.1	75.3	73.9	91.3	37.0	5.4
	2	319.1	146.6	212.2	392.6	727.8	131.4	16.1		390.7	86.4	82.1	94.9	142.2	168.0	13.8
	5	340.5	198.7	311.2	618.5	1,170.4	281.2	49.4		449.1	95.0	102.4	149.6	268.1	350.3	45.5
	7	349.2	245.2	394.3	811.5	1,533.4	383.7	85.4		457.1	104.1	121.4	208.3	365.0	465.1	80.4
	10	345.7	322.4	531.5	1,125.9	2,140.4	539.4	160.3		462.7	123.4	158.2	296.2	539.4	635.1	155.7
	12	339.2	358.9	599.9	1,287.6	2,451.3	640.4	219.8		464.3	139.9	189.3	370.2	677.3	742.6	221.5
	15	340.4	449.8	767.4	1,663.8	3,175.2	787.2	326.3		528.1	175.8	266.8	520.2	958.5	903.9	342.0
	17	340.2	534.6	918.8	2,004.7	3,831.2	882.1	405.3		573.0	182.3	292.6	568.1	1,053.4	1,007.1	438.5
	20	341.1	559.4	968.9	2,133.6	4,083.8	1,025.7	531.1		521.8	219.2	358.9	722.2	1,343.9	1,162.2	603.8
	22	341.6	547.4	948.5	2,102.7	4,024.3	1,122.1	613.7		468.1	257.4	419.8	862.2	1,611.6	1,263.2	724.9
	25	342.3	555.7	967.9	2,160.1	4,139.9	1,257.8	730.1		467.7	281.3	453.7	948.9	1,778.4	1,420.9	924.0
	E	Edlib-150	Edlib-50	Edlib-25	Edlib-10	Edlib-5	S.Snake-150	S.Snake-5		Edlib-150	Edlib-50	Edlib-25	Edlib-10	Edlib-5	S.Snake-150	S.Snake-5
set_11	0	502.2	81.3	68.0	61.1	66.4	37.4	5.3	set_12	449.0	81.9	67.1	80.3	106.1	37.8	5.6
	2	470.0	80.4	69.4	70.4	96.6	172.5	13.3		467.8	77.2	65.3	84.6	129.5	181.1	13.7
	5	505.5	82.1	80.6	103.6	177.3	359.9	45.7		524.5	77.9	78.3	114.4	189.4	374.9	43.2
	7	490.4	86.1	92.3	141.0	237.3	477.3	80.9		533.7	79.8	96.2	139.5	233.0	493.6	76.2
	10	488.3	99.5	120.1	208.1	367.2	651.6	156.5		612.2	81.5	117.0	175.6	302.9	668.5	146.9
	12	482.3	112.2	144.7	263.7	474.2	760.0	222.2		655.4	83.2	123.0	199.4	350.2	780.0	208.1
	15	483.5	130.5	188.6	352.2	638.3	919.7	344.7		657.9	89.6	137.9	244.0	434.5	944.1	317.4
	17	483.1	158.8	243.8	454.5	830.4	1,025.3	443.1		547.3	106.2	166.1	295.1	531.3	1,053.9	410.4
	20	484.1	167.1	265.4	509.9	938.4	1,182.5	612.5		542.7	123.8	187.1	328.0	594.3	1,230.4	569.6
	22	483.9	200.1	317.0	623.7	1,152.2	1,283.6	737.9		541.1	143.2	205.2	351.8	638.7	1,315.5	685.2
	25	482.4	250.7	393.4	798.9	1,484.1	1,439.7	947.8		544.2	177.0	242.5	422.0	769.2	1,467.6	884.3